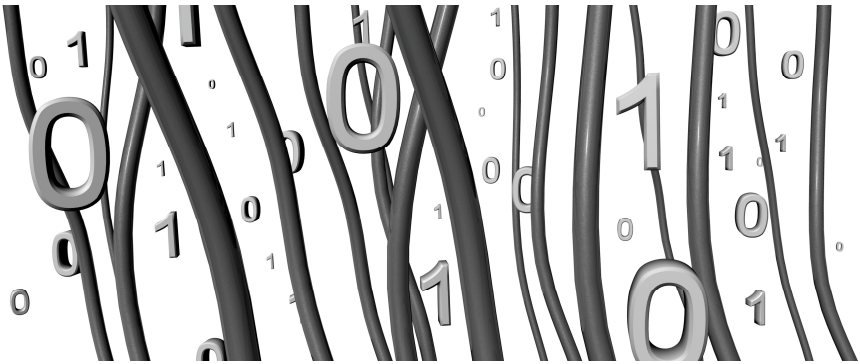
Nacionalinė Moksleivių Akademija

Informatika



Informatikos olimpiados: algoritmai ir taikymo pavyzdžiai

Autoriai: L.Petrauskas, J.Skūpienė

Vilnius, 2006

Viršelio autorius:

Karolis Narkevičius

Iliustracijos:

Linas Petrauskas

Knygos autoriai:

© Linas Petrauskas
© Jūratė Skūpienė, Matematikos ir informatikos institutas

Redaktorė:

Julija Klimkienė

Recenzavo:

Doc.dr. Vladas Tumasonis, Vilniaus universitetas

ISBN 9955-9894-0-8

ĮVADAS

Čia reikia bėgti kiek įkabini, kad išsilaikytum toje pačioje vietoje.
O jei nori patekti kur nors kitur, reikia bėgti dvigubai greičiau.
Luisas Kerolis, „Alisa veidrodžių karalystėje“

Ši knygelė skirta moksleiviams, kurie ruošiasi dalyvauti informatikos olimpiadose. Galima svarstyti ir ginčytis, ką reiškia žodis *informatika*, tačiau pasaulinių olimpiadų kontekste informatikos olimpiada – tai pirmiausia įdomūs algoritmavimo uždaviniai, kuriems išspręsti reikia ne tik išmanyti įvairius algoritmus, bet ir rasti įdomių, netradicinių sprendimų bei sugebėti per keletą valandų „materializuoti“ savo idėjas veikiančiomis programomis.

Informatikos olimpiados jau beveik dvidešimt metų kasmet rengiamos Lietuvoje ir beveik tiek pat metų geriausi Nacionalinės informatikos olimpiados dalyviai atstovauja Lietuvai Baltijos bei pasaulinėse informatikos olimpiadose. Knygelėje aprašyti svarbūs algoritmai ir metodai, kuriuos turėtų žinoti kiekvienas norintis sėkmingai dalyvauti šiose olimpiadose. Tiesa, čia nėra griežtų algoritmų teisingumo įrodymų, kurie pateikiami aukštųjų mokyklų vadovėliuose. Algoritmų teisingumas dažniausiai grindžiamas intuityviai, o teorija iliustruojama konkrečiais uždaviniais, paimtais iš realių olimpiadų. Leidinyje pateikėme sutrumpintas uždavinių sąlygas, atmetę daug detalių, susijusių su automatiniu programų testavimu. Dauguma sprendimų pateikti kaip Paskalio kalbos[1] procedūros ar funkcijos, stengiantis kuo mažiau prisirišti prie konkrečios

[1] Knygelėje pateiktos programos kompiliuotos *Free Pascal* kompiliatoriumi, kuris naudojamas ir tarptautinėse olimpiadose (www.freepascal.org).

programavimo kalbos subtilumų. Tad leidinys puikiausiai tiks ir programuojantiems kitomis programavimo kalbomis.

Tikimės, kad knygelė bus pirmasis rengimosi informatikos olimpiadoms etapas. Jį įveikus dar bus ilgas kelias, kuriame teks spręsti daugybę įdomių uždavinių ir kuriuo reikės bėgti dvigubai greičiau, norint pasiekti daugiau.

Už vertingas pastabas ir pagalbą rengiant šį leidinį nuoširdžiai dėkojame Justui Kranauskui, Martynui Kriaučiūnui bei Mindaugui Plukui.

Autoriai

TURINYS

1 INTUITYVIOS ALGORITMO IR ALGORITMO SUDĖTINGUMO SĄVOKOS

1.1 Algoritmas

Languages come and go, but algorithms stand the test of time.
Programavimo kalbos atsiranda ir išnyksta,
bet algoritmai išlieka ilgam.
Donaldas Knutas (Donald Knuth)

Algoritmo sąvoka atsirado daugiau kaip prieš tūkstantį metų, o pats žodis kilo iš IX a. persų matematiko Mohamedo ibn Musos al Chorezmio (*Al-Khwarizmi*) vardo. **Algoritmu sutarta vadinti seką elementarių veiksmų, pradinius duomenis paverčiančių rezultatais.** Algoritmas yra teisingas, jei su visais pradiniais duomenimis baigia darbą ir gaunami teisingi rezultatai.

1 pav. Mohamedas ibn Musa al Chorezmis

Koks algoritmas yra geras? Į šį klausimą puikiai atsakyta jau klasika tapusioje knygoje „Įvadas į algoritmus“ [2]:

Geras algoritmas – kaip aštrus peilis – atlieka tiksliai tai, ką turi atlikti, su minimaliomis pastangomis. Netinkamo algoritmo naudojimas problemai spręsti primena bandymą perpjauti kepsnį atsuktuvu: anksčiau ar vėliau gali pavykti pasiekti patenkinamą rezultatą, tačiau teks išnaudoti daug daugiau pastangų negu būtina, ir pats rezultatas nekels estetinio pasigėrėjimo.

Geras algoritmas turi būti teisingas ir efektyvus laiko ir atminties požiūriu. Jis taip pat turi būti lengvai realizuojamas, t. y. užrašomas realia programavimo kalba. Dažniausiai visų tikslų pasiekti nepavyksta ir tenka nusileisti iki kompromiso. Mokslininkai teoretikai

linkę skirti dėmesį teisingumui ir efektyvumui, nes jie retai patys programuoja savo algoritmus. Tuo tarpu industrija renkasi vadinamąjį *greitą ir purviną* (angl. *Quick and Dirty*) darbo principą: bet kokia programa, kuri pateikia priimtinus rezultatus ir pernelyg nesulėtina darbo, yra tinkama, nepaisant to, kad gali būti ir geresnis algoritmas.

Atskirai paminėsime euristinius algoritmus. Ne visiems uždaviniams spręsti sugalvoti efektyvūs algoritmai, o teisingi, bet neefektyvūs algoritmai praktiškai nepritaikomi, nes jų vykdymas užtruktų per ilgai, pavyzdžiui, kelis šimtmečius. Jei uždavinį vis dėlto reikia spręsti, galvojami spartūs algoritmai, kurie nebūtinai suranda tikslų sprendinį, tačiau rastasis sprendinys *dažniausiai* yra *artimas* ieškomajam. Tokie optimizmu grįsti algoritmai vadinami **euristiniais algoritmais** arba tiesiog **euristikomis**.

Iš tiesų skyrelio pradžioje pateiktas algoritmo apibrėžimas tėra intuityvi ir matematiškai netiksli algoritmo sąvoka[1]. Tačiau mums jos pakaks.

1.2 Algoritmo sudėtingumas

Kaip jau minėjome, yra uždavinių, kurių kompiuteris negali išspręsti per priimtiną laiką, ir būtų neišmintinga viltis, kad kompiuteriai gali

[1] Tikslios algoritmo sąvokos prireikė matematikams, panorusiems įrodyti, kad nėra algoritmo, sprendžiančio duotąjį uždavinį. 20-ajame amžiuje daug matematikų ieškojo būdo tiksliai apibrėžti algoritmo sąvoką. Galima sakyti, jog jiems pavyko. 1936 m. amerikiečių matematikas A. Čerčas (A. Church) paskelbė tezę, teigiančią, jog jo apibrėžta dalinių rekursyviųjų funkcijų (DRF) klasė sutampa su algoritmiškai apskaičiuojamų funkcijų klase. Tačiau tezės įrodyti negalima, kadangi neįmanoma palyginti matematiškai tikslios ir intuityviai suprantamos funkcijų klasių. Kita vertus, niekam nepavyko rasti algoritmo (intuityviąja prasme), kurio nebūtų galima realizuoti kaip DRF, o visos DRF apskaičiuojamos algoritmais intuityviąja prasme, todėl Čerčo tezė visuotinai laikoma teisinga.

greitai atlikti *bet kokius* skaičiavimus. Todėl svarbu mokėti įvertinti **algoritmo sudėtingumą**, t. y. nustatyti, kiek laiko ir atminties išteklių prireiks algoritmo vykdymui.

Kas gi yra spartus algoritmas? Kuo didesnis pradinių duomenų kiekis (arba dydis), tuo ilgiau veikia programos, apdorojančios šiuos duomenis. Taigi algoritmas yra spartus, jei ganėtinai greitai apdoroja *didelius duomenų kiekius*. Negalime sakyti, kad vienas rikiavimo algoritmas spartesnis už kitą, jei pirmasis 10 skaičių išrikiavo greičiau nei antrasis. Kas kita, jei tenka rikiuoti labai daug skaičių. Apskritai konkretūs laiko įverčiai dažniausiai neteikia naudos, kadangi priklauso nuo daugybės veiksnių – techninių kompiuterio parametrų, algoritmo realizacijos, kompiliatoriaus nustatymų ir pan.

Daug svarbiau žinoti, kaip algoritmo vykdymui reikalingi laiko ir atminties ištekliai priklauso nuo pradinių duomenų kiekio. Žinodami, kad rikiavimo algoritmo atliekamų veiksmų skaičius didėja proporcingai rikiuojamos sekos ilgio kvadratui, galėsime nuspręsti, ar toks efektyvumas priimtinas.

Algoritmo sudėtingumas **laiko atžvilgiu** vertinamas funkcija, apibrėžiančia atliekamų veiksmų skaičiaus priklausomybę nuo pradinių duomenų dydžio. Algoritmo sudėtingumas **atminties atžvilgiu** vertinamas funkcija, apibrėžiančia reikalingos atminties kiekio priklausomybę nuo pradinių duomenų dydžio.

Kas yra pradinių duomenų dydis? Tai priklauso nuo paties algoritmo. Pavyzdžiui, dažnam rikiavimo algoritmui duomenų dydį apibrėžia rikiuojamų skaičių kiekis, bet ne patys skaičiai. Tačiau yra rikiavimo algoritmų, kurių efektyvumas priklauso ir nuo pačių rikiuojamų skaičių, todėl šiuo atveju duomenų dydis papildomas ir maksimalia rikiuojamų skaičių reikšme.

1.3 Kaip įvertinti algoritmo sudėtingumą

Natūralus būdas įvertinti algoritmo sudėtingumą – apskaičiuoti, kiek elementarių veiksmų (aritmetinių operacijų, kreipimųsi į atmintį) jis atlieka. Susitarsime, kad visi elementarūs veiksmai įvykdomi vienodai greitai[2]. Žinodami, kiek vidutiniškai elementarių veiksmų per sekundę atlieka kompiuteris, galėsime įvertinti vykdymui reikalingą laiką. Panagrinėkime programos fragmentą, randantį kvadratinėje $n \times n$ lentelėje surašytų skaičių sumą, ir suskaičiuokime atliekamų elementarių veiksmų skaičių.

```
suma := 0;                          // atliekamas vieną kartą
read(n);                            //          vieną kartą
for i := 1 to n do                  //          n kartų
    for j := 1 to n do begin        //          n² kartų
        read(a);                    //          n² kartų
        suma := suma + a;           //          n² kartų
                                    // (priskyrimas ir sumavimas)
    end;
writeln(suma);                      //          vieną kartą
```

Elementarių veiksmų skaičius lygus $1 + 1 + n + n^2 + n^2 + 2n^2 + 1 = 4n^2 + n + 3$. Jį nusako funkcija $f(n) = 4n^2 + n + 3$. Tai ir yra šio fragmento sudėtingumas laiko atžvilgiu.

Jei paimtumėte kurią nors savo programą ir pabandytumėte pakartoti šiuos žingsnius, tikriausiai susiimtumėte už galvos! Kaip skaičiuoti, jei programoje yra ciklas **`while`** ar naudojama rekursija,

[2] Toks modelis kartais kritikuojamas, nes vieni elementarūs veiksmai įvykdomi greičiau negu kiti. Pavyzdžiui, skaičių perskaityti iš failo trunka ilgiau nei tą patį skaičių perskaityti iš operatyviosios atminties. Kelių knygų apie algoritmus autorius prof. S. Skienna drąsiai atremia tokią kritiką: *Visi žinome, kad žemė yra apvali, tačiau statydami namą laikome ją plokščia ir toks modelis mums puikiausiai tinka. Tas pats galioja ir šiuo atveju.* [6]

jei priklausomai nuo įvairių sąlygų vieną kartą atliekami vieni, o kitą – kiti veiksmai.

Panagrinėkime kurį nors rikiavimo algoritmą. Jei pradiniai duomenys sudaro surikiuotą seką, tikriausiai bus atliekama mažiau veiksmų, negu rikiuojant atsitiktinę seką. Tad atliekamų elementarių veiksmų skaičius gali priklausyti ne tik nuo pradinių duomenų kiekio, bet ir nuo pačių duomenų.

Dėl šių priežasčių dažnai skaičiuojama, kiek veiksmų bus atliekama **blogiausiu atveju**, t. y. kiek *daugiausiai* elementarių veiksmų gali tekti atlikti vykdant algoritmą.

Kiekvienos programos veikimą nusakys vis kitokia funkcija. Tiksliai suskaičiuoti elementarių veiksmų kiekį didesnėms programoms būtų sudėtinga. Laimei, to daryti neteks! Panagrinėkime, kaip didėjant n auga kiekvienas iš dėmenų. Kai $n = 1$, dėmenys lygūs 4, 1 ir 3, kai $n = 10$, jie atitinkamai lygūs 400, 10 ir 3, kai $n = 1000$, gauname 4 000 000, 1000 ir 3. Matome, kad didėjant n labiausiai didėja tik pirmasis dėmuo, o kiti dėmenys – labai nežymiai. Kadangi kiekvienas dėmuo tiesiogiai reiškia elementarių veiksmų skaičių, du mažesniuosius dėmenis galime atmesti. Laikas, sugaištas atlikti 1003 veiksmams, yra nereikšmingas palyginti su laiku, reikalingu atlikti keturiems milijonams veiksmų.

Taigi, augant pradiniams duomenims (n), algoritmo atliekamų elementarių veiksmų skaičius vis labiau priklausys nuo greičiausiai augančio funkcijos dėmens, t.y. nuo $4n^2$. Natūralu vietoj funkcijos $f(n) = 4n^2 + n + 3$ toliau nagrinėti paprastesnę funkciją $g(n) = 4n^2$.

Tai dar ne viskas. Padidinus n dešimt kartų, vykdymo laikas padidės šimtąkart. Palyginus su tuo, vykdymo laiko padidėjimas keturis kartus yra neesminis. Taigi galime atmesti konstantą prie n^2 ir tarti,

kad elementarių veiksmų skaičių pakankamai gerai nusako dar paprastesnė funkcija $h(n) = n^2$.

Mokslininkai rašytų, kad nagrinėto programos fragmento sudėtingumas yra $O(n^2)$. Mat visur, kur kalbama apie algoritmų sudėtingumą, naudojamas *didžiosios O žymėjimas*.

1.4 Didžiosios O žymėjimas

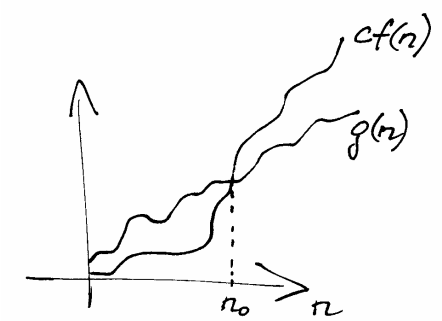
Formaliai **algoritmo sudėtingumas** apibrėžiamas taip:

> Tarkime, pradinių duomenų dydis yra n, o algoritmo atliekamų elementarių veiksmų skaičius – $g(n)$. Sakysime, jog algoritmo sudėtingumas yra $O(f(n))$ (rašome $g(n) = O(f(n))$), jei egzistuoja tokie skaičiai c ir n_0, su kuriais visiems $n > n_0$ galioja nelygybės: $0 \le g(n) \le cf(n)$.

Geriau suprasti šį apibrėžimą padės 2 paveiksle pateikti funkcijų f ir g grafikai.

Šis formalus apibrėžimas reiškia, kad, augant n, funkcija $g(n)$ auga ne sparčiau nei funkcija $f(n)$.

Sutartiniu didžiosios O žymėjimu paprastai parodoma, kaip elgsis algoritmas didėjant pradiniams duomenims, t. y. kaip augs algoritmui reikalingos atminties dydis arba vykdymo laikas.



2 pav. g(n) = O(f(n))

Panagrinėkime dar keletą pavyzdžių:

$3n^2 + 2n + 20 = O(n^2)$,

$n + 10\,000 = O(n)$,

$n + 10\,000 = O(n^2)$ (pagal apibrėžimą teisingas teiginys, tačiau parankesnė praeita lygybė),

$2^n + n^{10} = O(2^n)$.

Jei algoritmo sudėtingumas nepriklauso nuo duomenų kiekio (t. y. jis pastovus, konstantinis), tai jį žymėsime O(1). Pavyzdžiui, atminties, kurią naudoja nagrinėtas programos fragmentas, dydis lygus O(1).

Pradinių duomenų dydį gali nusakyti ne vienas, o keli kintamieji. Tokiu atveju didžiosios O žymėjimas aprašo sudėtingumo augimą didėjant visiems parametrams. Pavyzdžiui, galimi tokie algoritmo sudėtingumo variantai: $O(2^{n+m})$, $O(L^2W + W^2L)$.

Nusakant algoritmų sudėtingumą dažnai teks susidurti su šiomis funkcijomis:

O(1) (konstantinis), O(log *n*) (logaritminis[3]), $O(\sqrt{n})$ (šakninis[4]), O(*n*) (tiesinis), O(*n*log *n*), $O(n^2)$ (kvadratinis), $O(n^3)$ (kubinis), $O(2^n)$ (eksponentinis), O(*n*!) (faktorialinis[5]).

---

[3] Logaritmas yra funkcija, atvirkščia kėlimui laipsniu. $\log_a b$ (a, $b > 0$; $a \neq 1$; a vadinamas logaritmo pagrindu) atsako į klausimą: kokiu laipsniu reikia pakelti a, kad gautume b? Pavyzdžiui, $\log_2 8 = 3$, $\log_5 625 = 4$, $\log_2 32768 = 15$. Logaritmas – vienodai lėtai auganti funkcija, nesvarbu koks logaritmo pagrindas. Taigi logaritminis algoritmo sudėtingumas yra labai palankus. Didžiosios O žymėjime logaritmo pagrindas dažnai nerašomas.

[4] Kvadratinė šaknis iš skaičiaus n yra toks skaičius r, kad $r^2 = n$, $r \geq 0$.

1.5 Kaip tai pritaikyti olimpiadoje

Olimpiadose ribojamas programų veikimo laikas ir naudojamoji atmintis. Taigi apmąstant įvairius sprendimo būdus reikia mokėti įvertinti, ar programa bus pakankamai efektyvi (ar suspės įveikti uždavinį su visais pradiniais duomenimis per leistiną laiką). Tačiau kiek gi veiksmų gali atlikti kompiuteris per, pavyzdžiui, vieną sekundę? Tai priklauso nuo daugelio dalykų: nuo procesoriaus, kompiliatoriaus, pačių veiksmų, kuriuos programa atlieka. Atliekamų veiksmų skaičių mums padės įvertinti paprasta programa:

```
uses windows;

var pradžia, veiksmųSk : longint;

begin
    veiksmųSk := 0;
    pradžia := GetTickCount;
    while GetTickCount - pradžia < 1000 do
        inc(veiksmųSk);
    writeln(veiksmųSk);
end.
```

Ši programa suskaičiuoja, kiek elementarių veiksmų kompiuteris gali atlikti per vieną sekundę (suprantama, jei programą pradėjote ir baigėte vykdyti tą pačią parą). Be abejo, matavimai nėra visiškai tikslūs, tačiau jų pakanka įvertinti kompiuterio spartai.

Taigi tarkime, kad duomenų dydis yra n, $O(f(n))$ sudėtingumo algoritmas atlieka lygiai $f(n)$ elementarių veiksmų, o atlikę pateiktą programą įvertinome, kad kompiuteris per 1 sekundę atlieka 10^9 tokių veiksmų. Sudarykime lentelę, atspindinčią, kiek laiko trunka

[5] Teigiamo skaičiaus n faktorialu vadinama visų skaičių nuo 1 iki n sandauga ($n! = 1 \cdot 2 \cdot \ldots \cdot n$).

įvairaus sudėtingumo algoritmų vykdymas su įvairiais pradiniais duomenimis.

n	**10**	**20**	**30**	**100**	**1 000**	**10^6**	**10^9**
O(1)	~0	~0	~0	~0	~0	~0	~0
$O(\log_2 n)$	~0	~0	~0	~0	~0	~0	~0
$O(\sqrt{n})$	~0	~0	~0	~0	~0	~0	~0,03 ms
O(n)	~0	~0	~0	~0	~0	~1 ms	~1 s
$O(n\log_2 n)$	~0	~0	~0	~0	~0	~20 ms	~30 s
$O(n^2)$	~0	~0	~0	~0	~1 ms	~17 min	~32 metai
$O(n^3)$	~0	~0	~0.03 ms	~1 ms	~1 s	~32 metai	$\sim 32\times 10^9$ metų
$O(2^n)$	~0	~1 ms	~1 s	$\sim 4\times 10^{13}$ metų	–	–	–
O(n!)	~4 ms	~77 metai	$\sim 8\times 10^{15}$ metų	–	–	–	–

Sunku patikėti, bet tai tiesa: naivus skaičių rikiavimo algoritmas, kuris bando visus įmanomus skaičių išdėstymo būdus (tokių yra $n!$), ir tikrina, ar gautoji skaičių seka yra didėjanti, dvidešimt skaičių „rikiuotų“ daug metų. Toks algoritmas, žinoma, yra neefektyvus.

Efektyviais laikomi **polinominio sudėtingumo algoritmai**, t. y. tokie, kurių sudėtingumo funkcija yra polinomas – $O(n^k)$. Pirmieji septyni lentelėje pateikti sudėtingumai yra polinominiai, taigi laikomi efektyviais. Algoritmai, kurių sudėtingumas nepolinominis,

laikomi neefektyviais. Tokie yra eksponentinio (pavyzdžiui, $O(2^n)$) ir faktorialinio ($O(n!)$) sudėtingumo algoritmai.

Šią lentelę verta įsidėmėti. Olimpiados metu, sugalvoję uždavinio sprendimą, galime įvertinti jo sudėtingumą ir patikrinti, ar to užteks pradiniams duomenims įveikti per leistiną laiką. Įgijus patirties, algoritmo sudėtingumą dažnai nesunku įvertinti pažvelgus į algoritmo struktūrą: kokie jame yra ciklai, kokie rekursiniai kreipiniai ir pan.

Dar daugiau: matydami, jog uždavinio pradiniai duomenys labai maži, žinome, kad pakaks ir neefektyvaus algoritmo uždaviniui spręsti. Ir atvirkščiai: jei uždavinio pradiniai duomenys yra dideli, o leistinas programos veikimo laikas – mažas, reikia ieškoti efektyvaus būdo, kaip spręsti šį uždavinį.

Beje, beveik visose programose 90% laiko sugaištama vykdant 10% kodo. Ir likusių 90% kodo optimizavimas, deja, neturės didelės įtakos programos efektyvumui. Tad prieš imantis optimizuoti kurią nors algoritmo dalį reikia įsitikinti, ar verta tai daryti.

1.6 Uždavinys *Posekio suma*

Pabandykime pritaikyti įgytas žinias spręsdami konkretų uždavinį:

> *Duotas sveikasis skaičius k bei n neneigiamų skaičių seka $a_1, a_2, \ldots, a_n$.*
>
> ***Užduotis.*** *Reikia nustatyti, ar egzistuoja tokie indeksai i ir j ($1 \le i \le j \le n$), kad sekos narių nuo a_i iki a_j suma būtų lygi skaičiui k.*

Galioja ribojimai:
$1 \le k \le 100\,000\,000$; $1 \le n \le 100\,000$; $0 \le a_i \le 1\,000$.
Vykdymo laikas: 1 s.

Aptarkime kelis galimus uždavinio sprendimo būdus bei jų sudėtingumą. Pats paprasčiausias būdas – perrinkti visas galimas indeksų *i* ir *j* poras, kiekvienąkart suskaičiuojant sekos narių nuo *i*-ojo iki *j*-ojo sumą:

```
rasta := false;
i := 0;
repeat
    j := i;
    i := i + 1;
    repeat
        j := j + 1;
        suma  := 0;
        for l := i to j do
            suma := suma + a[l];
            { ši operacija vykdoma daugiausiai kartų }
        rasta := (suma = k);
    until (j = n) or rasta;
until (i = n) or rasta;
```

Jei algoritmui baigus darbą kintamojo `rasta` reikšmė bus lygi `true`, tai *i* ir *j* bus ieškomi indeksai. Suskaičiavę, kiek elementarių veiksmų blogiausiu atveju atlieka algoritmas, pamatytume, kad greičiausiai augantis gautojo reiškinio dėmuo yra $n^3/6$, taigi šio algoritmo sudėtingumas – $O(n^3)$. Tai atsispindi ir algoritmo struktūroje: jį sudaro trys ciklai, įdėti vienas į kitą, ir kiekvieno šių ciklų trukmė tiesiogiai priklauso nuo *n*.

Tai nėra geriausias uždavinio sprendimo būdas. Pasižiūrėjus į 1.5 skyrelyje pateiktą lentelę[6], matyti, kad per leistiną laiką algoritmas

[6] Turima omenyje, jog uždavinio sprendimą testuojančio kompiuterio spartą atitinka minėta lentelė.

įveiktų testus, kur $n \leq \sim 1000$. Atkreipę dėmesį į tai, kad sekos nariai yra tik neneigiami skaičiai, galime sudaryti gudresnį algoritmą.

Tegul ieškomasis indeksas i lygus i_1 (t. y. kažkokiam konkrečiam skaičiui). Priskyrę indeksui j pradinę reikšmę i_1, jį didinsime tol, kol sekos narių nuo i iki j suma taps lygi arba viršys k (arba kol indeksas j pasieks sekos pabaigą). Sumos neperskaičiuosime iš naujo kiekvieną kartą, o, padidinę indeksą j, prie sumos tiesiog pridėsime sekos narį a_j.

```
rasta := false;
i := 0;
repeat
    j := i;
    i := i + 1;
    suma := 0;
    repeat
        j := j + 1;
        suma := suma + a[j];
    until (j = n) or (suma >= k);
    rasta := (suma = k);
until (i = n) or rasta;
```

Šį algoritmą sudaro du ciklai, antrasis jų pirmojo viduje, ir abiejų ilgis tiesiogiai priklauso nuo n. Blogiausiu atveju abiejuose cikluose bus vykdoma n žingsnių (pavyzdžiui, jei visi sekos nariai – nuliai, tuomet suma niekada netaps lygi arba didesnė už k), taigi šio algoritmo sudėtingumas yra $O(n^2)$. Tai daug geresnis algoritmas, jis gali įveikti testus, kur $n \leq \sim 30\,000$. Tačiau to nepakanka.

Kritiškai įvertinkime savo algoritmą. Tarkime, $n = 100\,000$, $i = 1$, $j = 90\,000$, ir $suma < k$. Kas atsitiks, jei, padidinus j dar vienetu, *suma* taps didesnė už k? Indeksas i bus padidintas vienetu, j priskirta i reikšmė ir iš naujo skaičiuojamos sumos. Tačiau jei sekos narių nuo 1 iki 90 000 suma buvo mažesnė už k, tai tuo labiau tokia bus ir

narių nuo 2 iki 90 000 suma. Šio (milžiniško) intervalo būtų galima netikrinti!

Tai apibendrinę, galime sudaryti dar geresnį algoritmą. Priskirkime indeksams reikšmes $i = j = 1$, o sumai reikšmę a_1. Tai bus pradinis intervalas. Veiksmus kartosime, kol *suma* nelygi k ir j mažesnis už n. Kiekvienu žingsniu vykdysime vieną iš šių veiksmų: jei *suma* mažesnė už k, intervalą praplėsime – padidinsime indeksą j ir prie sumos pridėsime a_j; jei *suma* didesnė už k (tai tokia ji tapo po paskutinio žingsnio), intervalą siaurinsime – iš sumos atimsime a_i ir padidinsime indekso i reikšmę. Jei po kurio nors žingsnio *suma* taps lygi k, algoritmas iškart nutrauks darbą.

```
suma := a[1];
i := 1;
j := 1;
while (suma <> k) and (j < n) do
    if suma < k then begin
        j := j + 1;
        suma := suma + a[j];
    end else begin
        suma := suma - a[i];
        i := i + 1;
    end;
rasta := (suma = k);
```

Kadangi vienu žingsniu padidinamas tik vienas iš indeksų ir kiekvienas iš indeksų gali būti padidintas ne daugiau kaip n kartų, daugių daugiausia gali tekti įvykdyti $2n$ žingsnių. Algoritmo sudėtingumas yra $O(n)$, taigi jo visiškai pakaks uždaviniui įveikti ir kai $n = 100\,000$.

Aptarėme kelis uždavinio *Posekio suma* sprendimus ir skirtingą jų efektyvumą. Atsiminkime, jog geras algoritmas atlieka tik tai, kas būtina. Ieškodami, kaip galime pagerinti algoritmą, galvokime, kokius nereikalingus arba pakartotinius veiksmus jis atlieka.

1.7 NP sudėtingumas

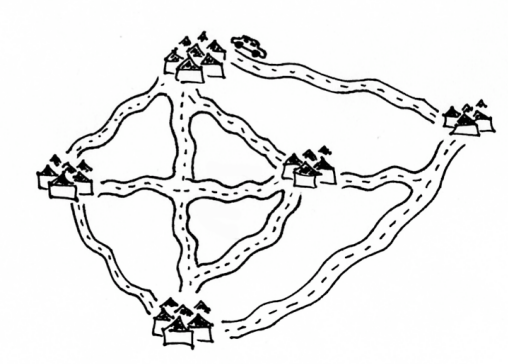
Skaitydami knygas apie algoritmus ir uždavinių sprendimus, ne kartą sutiksite mistiškai skambančią frazę **uždavinys yra NP pilnas**.

Uždavinys priklauso **NP** (*nondeterministic polynomial time*) **sudėtingumo klasei**, jei, žinodami šio uždavinio sprendinį, per polinominį laiką galime patikrinti, ar sprendinys teisingas. NP uždavinį galima išspręsti perrinkimu per eksponentinį laiką generuojant visus galimus sprendinius, ir kiekvieną sprendinį patikrinant per polinominį laiką.

NP klasei priklauso daug labai gerai žinomų ir plačiai nagrinėtų kombinatorinių optimizavimo uždavinių. Vieni jų yra paprastesni (išsprendžiami per polinominį laiką), kitiems, sudėtingesniems, uždaviniams, žinomi tik perrenkantys visus sprendinius algoritmai.

NP pilnas uždavinys yra toks uždavinys, kuris yra ne lengvesnis už visus kitus NP uždavinius. Taigi frazę „uždavinys yra NP pilnas“ „išvertus“ į suprantamesnę kalbą, reikštų: *niekam iki šiol nepavyko rasti efektyvaus uždavinį sprendžiančio algoritmo;* ***tikėtina****, kad toks algoritmas apskritai neegzistuoja.*

Nepaisant sudėtingumo, šie uždaviniai gali turėti labai paprastą formuluotę, pavyzdžiui, tokią. *Žinomi atstumai tarp n miestų; pirklys nori pradėti savo kelionę viename iš jų, apsilankyti kiekviename mieste tik po vieną kartą ir sugrįžti į pradinį miestą; užduotis – iš visų tokių maršrutų surasti trumpiausią.*



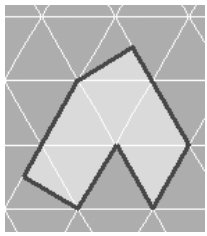
3 pav. Keliaujančio pirklio uždavinys

Šis uždavinys dar yra vadinamas *Keliaujančio pirklio uždaviniu*, o ieškomasis kelias – *optimaliu Hamiltono ciklu*.

Neįtikėtina, bet iki šiol niekas nesurado tikslaus ir efektyvaus algoritmo, sprendžiančio šį uždavinį. Vienintelis žinomas būdas rasti optimalų sprendinį bendru atveju – perrinkti visus įmanomus maršrutus O(n!) sudėtingumo (t.y. labai neefektyviu) algoritmu.

Ką gi daryti, jei olimpiadoje tenka spręsti uždavinį, kuris, jūsų žiniomis, yra NP pilnas? Tikrai neverta pulti į paniką. Svarbiausia, kad jūs tai jau žinote! Nereikia ieškoti tikslaus ir efektyvaus uždavinį sprendžiančio algoritmo manant, kad kiti jau tokį surado, o nesiseka tik jums. Verčiau skirkite savo laiką ir energiją kurti euristiniam algoritmui, kuris bendru atveju pateiktų kuo geresnius rezultatus (pavyzdžiui, kuo trumpesnius maršrutus), arba, jei pradiniai duomenys tikrai labai maži, – spręsti uždavinį perrinkimu.

1.8 *Amžinybės dėlionė*



4 pav. Amžinybės dėlionės *daugiakampio pavyzdys*

Kristoferis Montonas (*Christopher Monton)* sukūrė geometrinę dėlionę, kurią pavadino *Amžinybės dėlione* (angl. *Eternity puzzle*). Ji buvo sudaryta iš 209 įvairios formos netaisyklingų daugiakampių, iš kurių reikėjo sudėti dvylikakampį. Dauguma daugiakampių buvo skirtingi, o juos visaip sukiojant buvo galima pasiekti labai daug pozicijų (t.y. iš dalies daugiakampių sudėliotų geometrinių figūrų), kurios nenuvesdavo prie sprendinio.

K. Montonas užsakė pagaminti šią dėlionę, ir 1999 metų liepą ji atsidūrė parduotuvių lentynose. Jis taip pat pažadėjo, kad sumokės milijoną svarų tam, kuris pirmasis sudės šią dėlionę iki 2000 metų rugsėjo. Kilo visuotinis susidomėjimas dėlione, prekyba vyko labai sėkmingai: netgi Grenlandijoje buvo parduodami rekordiniai kiekiai dėlionių. Žmonės pirko, bandė sudėlioti dėlionę ir laimėti milijoną.

Prieš pažadėdamas milijoną, K. Montonas be abejo, konsultavosi su matematikais, ir šie užtikrino, kad uždavinio neįmanoma išspręsti per duotą laiką net ir su kompiuterio pagalba, nes tai NP pilnas uždavinys. Vieni ekspertai spėjo, kad geriausiu atveju uždavinio sprendimas užtruks apie ketverius metus. Kiti mokslininkai netgi teigė, kad uždavinio sprendimas užtruks ilgiau nei gyvuos Visata. Nors visada lieka atsitiktinio sudėliojimo tikimybė, buvo apskaičiuota, kad tikimybė vienu bandymu atsitiktinai sudėlioti šią dėlionę yra 1 iš 10^{500} (palyginimui: tikimybė išlošti Didžiosios Britanijos nacionalinėje loterijoje yra 1 iš $14 \cdot 10^{6}$).

K. Montonas buvo tikras, kad jo milijonas yra saugus. Už pinigus, gautus pardavus dėliones, jis tikėjosi suremontuoti jam priklausiusį 1825 metais pastatytą dvarą, turintį 67 kambarius ir 200 akrų žemės.

Tačiau du Kembridžo matematikai A. Serbis (*Alex Serby*) ir O. Riordanas (*Oliver Riordan*) sugebėjo sudėti dėlionę iki nurodytos datos. Jie pastebėjo, kad sudėlioti dėlionę iki tokios būsenos, kai likę daugiakampiai nebetelpa, yra gana paprasta. Tolesnė sėkmė priklauso nuo nepanaudotų daugiakampių rinkinio – kuo parankesnių formų jie yra ir kuo daugiau įvairios formos daugiakampių galima iš jų sudėti, tuo šis rinkinys parankesnis tolimesniems bandymams. Tokiu būdu jie atrinko „blogus“ gabalėlius ir optimizuotoje perrinkimo programoje stengėsi juos padėti pirmiausia. Ši strategija pasitvirtino ir keletas jų asmeninių kompiuterių per porą savaičių surado sprendinį. K. Montonui teko parduoti savo dvarą ir išmokėti milijoną...

2 PIRMASIS ALGORITMAS – EUKLIDO ALGORITMAS DIDŽIAUSIAM BENDRAJAM DALIKLIUI RASTI

Kartą mokinys, išmokęs savo pirmąją geometrijos teoremą, paklausė Euklido: „Kokia man nauda, kad šitai išmoksiu?“. Euklidas pakvietė savo vergą ir tarė: „Duok šiam žmogui drachmą, nes jis turi turėti naudos iš to, ką išmoksta.“
J. Stobijus (Joannes Stobaeus), V a. pr. Kr.

Šiame skyrelyje susipažinsime su seniausiu netrivialiu algoritmu, išlikusiu iki šių dienų. Tai algoritmas didžiausiam bendrajam dalikliui rasti. Nėra žinoma, kas šį algoritmą sugalvojo (ir ar tai buvo vienas žmogus). Dar prieš Euklidui (graikų k. Ευκλειδηξ, *Eukleides*) aprašant šį algoritmą, jį savo veikale cituoja Aristotelis. Euklidas algoritmą kruopščiai aprašė garsiajame veikale „*Pradmenys*“ (apie 300 m. pr. Kr.), todėl algoritmui ir prigijo Euklido vardas.

5 pav. Euklido portretas (sena graviūra)

2.1 Didžiausias bendrasis daliklis ir mažiausias bendrasis kartotinis

Prisiminkime *didžiausio bendrojo daliklio* (DBD) ir *mažiausio bendrojo kartotinio* (MBK) sąvokas.

Sakome, kad **skaičius *a* dalija skaičių *b***, jei egzistuoja toks sveikasis skaičius k, kad $b = ka$; žymime $a \mid b$.

Pavyzdžiui, $5 \mid 30$, nes $30 = 6 \cdot 5$. Skaičius 1 dalija visus skaičius ($1 \mid a$, visiems sveikiesiems a), o skaičių 0 dalija visi skaičiai, išskyrus patį 0 ($a \mid 0$, visiems sveikiesiems a, $a \neq 0$).

Dviejų neneigiamų skaičių a ir b **didžiausiu bendruoju dalikliu** (DBD) vadinamas didžiausias skaičius, dalijantis a ir b. Pavyzdžiui, $\mathrm{DBD}(12, 8) = 4$, $\mathrm{DBD}(3, 6) = 3$, $\mathrm{DBD}(7, 9) = 1$.

Neneigiamų skaičių a ir b **mažiausiu bendruoju kartotiniu** (MBK) vadinamas mažiausias teigiamas skaičius, kurį dalija a ir b. Pavyzdžiui, $\mathrm{MBK}(12, 8) = 24$, $\mathrm{MBK}(3, 6) = 6$, $\mathrm{MBK}(7, 9) = 63$.

Natūralus būdas rasti $\mathrm{DBD}(a, b)$ – išskaidyti skaičius a ir b pirminiais daugikliais ir išrinkti visus *bendruosius* šių skaičių pirminius daugiklius. Pavyzdžiui, $12 = \mathbf{2 \cdot 2} \cdot 3$, $8 = \mathbf{2 \cdot 2} \cdot 2$, bendrieji daugikliai yra $\mathbf{2 \cdot 2}$, taigi $\mathrm{DBD}(12, 8) = 4$. Šiuo būdu tarsi konstruojame DBD, stengdamiesi jį padaryti kuo didesnį (rinkdami kuo daugiau skaičiaus a pirminių daugiklių), tačiau žiūrėdami, kad DBD dalytų ir skaičių b.

$\mathrm{MBK}(a, b)$ taip pat galime rasti išskaidę skaičius a ir b į pirminius daugiklius. Kadangi $a \mid \mathrm{MBK}(a, b)$, tai MBK turi priklausyti visi a pirminiai daugikliai. Tačiau ir $b \mid \mathrm{MBK}(a, b)$, todėl pridedame skaičiaus b pirminius daugiklius, kurių trūksta (būtent, daugiklius, kurie *nėra bendrieji* skaičiams a ir b). Pavyzdžiui, $\mathrm{MBK}(12, 8) = 2 \cdot 2 \cdot 3 \cdot 2 = 24$.

Šie DBD ir MBK konstravimo būdai paaiškina ir šiuos skaičius siejančią lygybę: $\mathrm{DBD}(a, b) \cdot \mathrm{MBK}(a, b) = a \cdot b$.

2.2 Euklido algoritmas

An algorithm must be seen to be believed.
Algoritmą reikia pamatyti, kad juo patikėtum.
Donaldas Knutas (Donald Knuth)

Tarkime, reikia rasti skaičių a ir b didžiausiąjį bendrą daliklį. Atliekame tokius veiksmus:

- jei $b = 0$, tai DBD(a, b) lygus a, priešingu atveju $a_2 = b$, $b_2 = a \bmod b$ (lygus liekanai, gautai padalijus a iš b)
- jei $b_2 = 0$, tai DBD(a, b) lygus a_2, priešingu atveju $a_3 = b_2$, $b_3 = a_2 \bmod b_2$
- ...
- jei $b_k = 0$, tai DBD(a, b) lygus a_k, priešingu atveju $a_{k+1} = b_k$, $b_{k+1} = a_k \bmod b_k$.

Kadangi dalydami iš skaičiaus n galime gauti liekaną nuo 0 iki $n-1$, tai $b > b_2 > \ldots > b_k > \ldots > b_n = 0$, ir algoritmas atliks baigtinį skaičių veiksmų (anksčiau ar vėliau b_i taps lygus 0, tad algoritmas baigs darbą).

Raskime skaičių $a = 12$ ir $b = 8$ DBD naudodamiesi Euklido algoritmu:

- $b > 0$, taigi skaičiuojame $a_2 = b = 8$, $b_2 = a \bmod b = 12 \bmod 8 = 4$.
- $b_2 > 0$, taigi skaičiuojame $a_3 = b_2 = 4$, $b_3 = a_2 \bmod b_2 = 8 \bmod 4 = 0$.
- $b_3 = 0$, taigi DBD(a, b) $= a_3 = 4$.

Gavome DBD(12, 8) = 4. Užrašysime Euklido algoritmą Paskalio kalba.

```
function DBD(a, b : longint) : longint;
var c : longint;
begin
    while b > 0 do begin
        c := a;
        a := b;
        b := c mod b;
    end;
    DBD := a;
end;
```

Jei reikia rasti dviejų skaičių DBD, tačiau nežinome, ar jie teigiami, funkciją iškviečiame perduodami skaičių modulius: DBD(abs(a), abs(b)).

Euklido algoritmas yra **teisingas**, nes remiasi sąryšiu: DBD(a, b) = DBD(b, a mod b). Šio sąryšio teisingumu nesunku įsitikinti pasinaudojus lygybe:

$$a = (a \text{ div } b) \cdot b + a \bmod b.$$

Du skaičiai turi vieną ir tik vieną didžiausiąjį bendrą daliklį. Tarkime, DBD (a, b) = d. Daliklis d dalija skaičių a ir taip pat dalija jo dalį (a div b) · b, todėl turi dalyti ir likusią skaičiaus a dalį — a mod b. Taigi skaičių a ir b didžiausias bendrasis daliklis yra ir (mažesnių) skaičių poros b ir a mod b didžiausias bendrasis daliklis, t. y. DBD(a, b) = d = DBD(b, a mod b).

Pamėginkime įvertinti Euklido algoritmo **sudėtingumą**. Pasiremsime nelygybe $n \bmod m < n/2$, kur n ir m — sveikieji neneigiami skaičiai ir $n \geq m$.

Nelygybė teisinga, nes:

- jei $m \le n/2$, tuomet $n \bmod m < m \le n/2$;

- jei $m > n/2$, tuomet $n \operatorname{div} m = 1$; tada lygybę $n = (n \operatorname{div} m)\, m + n \bmod m$ perrašome: $n = m + n \bmod m$; gauname $n \bmod m = n - m < n - n/2 = n/2$.

Tarkime, kad $a > b$ (jei taip nėra, tai atliekant ciklą pirmąjį kartą, šie skaičiai bus sukeisti vietomis). Ciklo viduje atliekamas operacijas galime laikyti elementariomis, tad Euklido algoritmo sudėtingumas tiesiog proporcingas tam, kiek kartų bus atliekamas ciklas `while`.

Panagrinėkime, kaip keičiasi kintamųjų a ir b reikšmės vykdant `while` ciklą. Sakykime, pradinės šių kintamųjų reikšmės yra a_0 ir b_0. Po pirmos ciklo iteracijos $a_1 = b_0$, o $b_1 = a_0 \bmod b_0 < a_0/2$. Po antros iteracijos $a_2 = b_1 < a_0/2$, o $b_2 = a_1 \bmod b_1 < a_2$. Gavome, kad atlikus dvi ciklo iteracijas, pirmojo kintamojo reikšmė sumažėja daugiau negu dvigubai ir dar vis galioja $a \ge b$. Po keturių iteracijų pirmojo kintamojo reikšmė bus daugiau nei keturis kartus mažesnė už pradinę ir t. t. Taigi matyti, kad ciklas bus vykdomas ne daugiau kaip $2\log a$ kartų. Dabar jau nesunku įvertinti, kad Euklido algoritmo sudėtingumas yra $O(\log a)$.

Kadangi Euklido algoritmas apibrėžiamas rekurentiniais sąryšiais:

$\mathrm{DBD}(a, b) = a$, jei $b = 0$
$\mathrm{DBD}(a, b) = \mathrm{DBD}(b, a \bmod b)$, jei $b > 0$

tai Euklido algoritmą nesunku užrašyti rekursyvia[7] funkcija:

```
function DBD(a, b : longint) : longint;
begin
    if b = 0 then
        DBD := a
    else
        DBD := DBD(b, a mod b);
end;
```

Pastebėkime, kad jei $a < b$, algoritmas pirmu žingsniu šiuos skaičius sukeičia vietomis, pavyzdžiui, DBD(24, 54) = DBD(54, 24) = DBD(24, 6) = DBD(6, 0) = 6.

Beje, pats Euklidas šį algoritmą aprašė kiek kitaip. Mat graikų matematikai nelaikė, kad vienetas dalija kitą teigiamą skaičių. Buvo galimi trys variantai: arba du teigiami sveikieji skaičiai yra abu lygūs vienetui, arba tarpusavyje pirminiai, arba turi bendrą didžiausią daliklį. Vienetas netgi nebuvo laikomas skaičiumi, o nulis apskritai neegzistavo.

2.3 Euklido algoritmo taikymas, mažiausio bendrojo kartotinio (MBK) radimas

Didžiausiojo bendrojo daliklio gali prireikti sprendžiant įvairius skaičiavimo uždavinius. Vienas iš pavyzdžių – prastinant trupmenas, skaitiklį ir vardiklį reikia padalyti iš didžiausio jų bendrojo daliklio.

[7] Su rekursija išsamiai susipažinsime 4 skyriuje.

Euklido algoritmas leidžia efektyviai apskaičiuoti ir mažiausią bendrąjį kartotinį:

```
function MBK(a, b : longint[8]) : longint;
begin
    MBK := a * b div DBD(a, b);
end;
```

Naudodamiesi Euklido algoritmu galime rasti ne tik dviejų, bet ir keleto skaičių DBD bei MBK. Kadangi DBD(a, b, c) = DBD(DBD(a, b), c), ir MBK(a, b, c) = MBK(MBK(a, b), c). Šias lygybes suprasti ir įrodyti nesunku įsivaizduojant, kaip konstruotume DBD ir MBK iš skaičių a, b ir c pirminių daugiklių.

Tarkime, masyve m yra k sveikųjų skaičių. Pateiksime fragmentą, randantį visų k skaičių DBD ir MBK:

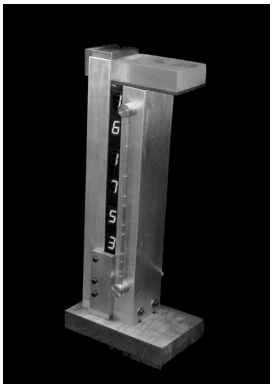
```
visųDBD := 0; { po pirmo žingsnio taps lygiu m[1] }
for i := 1 to k do
    visųDBD := DBD(abs(m[i]), visųDBD);

visųMBK := 1; { po pirmo žingsnio taps lygiu m[1] }
for i := 1 to k do
    visųMBK := MBK(abs(m[i]), visųMBK);
```

[8] Svarbu nepamiršti, kad *longint* tipo kintamieji gali saugoti reikšmes, ne didesnes negu $2^{31} - 1$. Taigi MBK bus skaičiuojamas teisingai tik tuo atveju, kai skaičių a ir b sandauga neviršija šio skaičiaus.

3 PIRMINIAI SKAIČIAI

Matematikai veltui bandė atrasti kokį nors dėsningumą pirminių skaičių sekoje, ir yra priežasčių manyti, kad šios paslapties žmogaus protas neperpras niekada.
Leonardas Oileris (Leonhard Euler)

Manoma, kad pirminiai skaičiai buvo žinomi jau Babilonijos civilizacijoje. Nuo seniausių laikų jie domino matematikus. XX a. pabaigoje pirminiai skaičiai buvo sėkmingai pritaikyti kriptografijoje: kelios populiarios viešojo rakto kriptoschemos paremtos faktu, jog sudauginti du skaičius lengva, o didelį skaičių išskaidyti pirminiais daugikliais – labai sudėtinga. Žinių apie pirminius skaičius gali prireikti ir sprendžiant įvairius skaičiavimo uždavinius.



6 pav. Laikrodis, rodantis tik pirminį laiką (valandas, minutes, sekundes). Šis laikrodis per parą 7669 kartus teisingai rodo laiką

3.1 Pirminiai skaičiai ir pagrindinė aritmetikos teorema

Pirminiais vadinami natūralieji skaičiai, kurie dalijasi tik iš vieneto ir savęs. Štai dešimt pirmųjų pirminių skaičių: 2, 3, 5, 7, 11, 13, 17, 19, 23, 29. Pirminiai skaičiai matematikoje yra svarbūs dėl **pagrindinės aritmetikos teoremos**, teigiančios, kad kiekvieną skaičių vieninteliu (unikaliu) būdu galima išreikšti pirminių skaičių sandauga, nekreipiant dėmesio į jų tvarką. Didelė šios teoremos svarba yra viena priežasčių, kodėl skaičius *vienas* nelaikomas pirminiu: tuomet teoremą reikėtų papildyti dar viena nereikalinga sąlyga.

3.2 Kiek jų yra?

Pirminių skaičių yra be galo daug. Tai žmonės žinojo jau labai seniai. Euklidas savo veikale „Pradmenys“ pateikė grakštų įrodymą:

> Tarkime, kad pirminių skaičių yra baigtinis kiekis – k. Pažymėkime šiuos k pirminių skaičių p_1, p_2, ..., p_{k-1}, p_k, ir panagrinėkime skaičių $m = p_1 \cdot p_2 \cdot p_3 \cdot \ldots \cdot p_{k-1} \cdot p_k + 1$. Dalydami m iš bet kurio p_i $(1 \le i \le k)$ gausime liekaną 1, t. y. nė vienas pirminis skaičius nedalija m. Tai reiškia, kad arba m pats yra pirminis, arba išrašėme ne visus pirminius skaičius. Bet kuriuo atveju yra bent $k + 1$ pirminių skaičių – gavome prieštarą. Taigi pradžioje padaryta prielaida, kad pirminių skaičių yra baigtinis kiekis, buvo neteisinga. Vadinasi, pirminių skaičių yra be galo daug. Tai ir reikėjo įrodyti.

Kiek yra pirminių skaičių, ne didesnių už n? Šis klausimas buvo užduodamas taip dažnai, kad atsakymas turi net specialų vardą – $\pi(n)$. *Pi funkcijos* reikšmė lygi pirminių skaičių, mažesnių arba lygių n, skaičiui (ši funkcija neturi nieko bendra su skaičiumi π). Pavyzdžiui, $\pi(20) = 8$, nes yra aštuoni pirminiai skaičiai, mažesni arba lygūs 20. Iš tiesų nėra jokio paprasto ir efektyvaus būdo, kaip šią funkciją apskaičiuoti, kai argumentas didelis[9].

3.3 Ar skaičius 234234743 pirminis?

Pats paprasčiausias būdas nustatyti, ar skaičius n pirminis – patikrinti, ar jis tenkina pirminio skaičiaus apibrėžimą, t. y. ar

[9] Tačiau įrodyta, jog teisingas šis funkcijos vertinimas: $0{,}89\dfrac{n}{\ln n} < \pi(n) < 1{,}11\dfrac{n}{\ln n}$.

Taigi funkcijos $\pi(n)$ priklausomybė nuo argumento nedaug skiriasi nuo tiesinės.

neatsiras tokio skaičiaus d ($1 < d < n$), kuris dalytų n. Algoritmo, tikrinančio visus potencialius daliklius nuo 2 iki n–1, sudėtingumas yra O(n).

Veiksmų skaičių nesunku sumažinti dvigubai: iš pradžių patikrinę, ar n nelyginis, vėliau galime tikrinti dalumą tik iš nelyginių skaičių. Nors veiksmų teks atlikti beveik dvigubai mažiau, algoritmo sudėtingumas taip pat yra O(n), nes veiksmų skaičius tiesiškai priklauso nuo n. Įrodysime, kad pakanka tikrinti potencialius daliklius nuo 2 iki $\sqrt{n}$.

> Tarkime, $n = d_1 \cdot d_2$. Jei $d_1 > \sqrt{n}$ ir $d_2 > \sqrt{n}$, tuomet $d_1 \cdot d_2 > n$, taigi arba $d_1 \le \sqrt{n}$, arba $d_2 \le \sqrt{n}$. Todėl, nuosekliai ieškodami daliklių nuo 2, negalime tikėtis rasti daliklį $d_1 > \sqrt{n}$, nes $d_2 = (n / d_1) < \sqrt{n}$ taip pat turi būti skaičiaus n daliklis, ir jį mes būtume aptikę anksčiau.

Apibendrinę šiuos pastebėjimus, galime parašyti pakankamai spartų (O($\sqrt{n}$) sudėtingumo) algoritmą, tikrinantį, ar skaičius $n > 1$ pirminis.

```
function pirminis(n : longint) : boolean;
var d,              { potencialus daliklis }
    sn : longint;   { riba, iki kurios ieškosime daliklių }
begin
    pirminis := (n mod 2 <> 0) or (n = 2);
    sn := round(sqrt(n) + 1);
    d := 3;         { tikrinsime dalumą iš nelyginių skaičių }
    while pirminis and (d < sn) do
        if n mod d = 0 then pirminis := false
        else d := d + 2;
end;
```

Įvykdę funkciją `pirminis` galime atsakyti į skyrelio pradžioje pateiktą klausimą – skaičius 234234743 tikrai pirminis.

7 pav. Graikų matematikas Eratostenas 276 – 194 m. pr. Kr.

Jei skaičių reikšmės būtų per didelės standartiniams sveikųjų skaičių tipams, tai su jais atliekamos aritmetinės operacijos nebegalėtų būti prilyginamos elementariems veiksmams, o joms atlikti tektų rašyti specialias procedūras. Tai keistų ir algoritmo sudėtingumą.

Ilgą laiką buvo nežinomas polinominis algoritmas, tikrinantis, ar didelis skaičius yra pirminis[10], ir tik 2002 metais Indijos mokslininkų grupė įrodė, kad tai nėra NP pilnas uždavinys. Beje, jei būtų atrastas būdas efektyviai išskaidyti didelį skaičių pirminiais dauginamaisiais, tai kai kurios svarbios saugumo sistemos taptų nesaugios.

3.4 Eratosteno rėtis

Jei norėtume surasti visus pirminius skaičius, mažesnius arba lygius n, galėtume tikrinti kiekvieną iš jų ką tik aprašytuoju būdu. Tokio algoritmo sudėtingumas – $O(n\sqrt{n})$. Tačiau šitaip ieškodami pirminių skaičių mes nepasinaudotume svarbiu faktu: tikrinant, ar skaičius n_0 pirminis, jau rasti visi pirminiai skaičiai, mažesni už n_0.

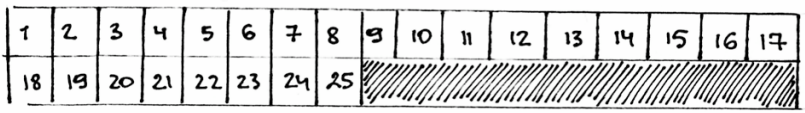
[10] Operacijų su dideliais skaičiais sudėtingumas matuojamas aritmetinių bitų operacijų skaičiumi. Tokiu atveju pradinių duomenų dydis yra skaitmenų (bitų) skaičius, taigi skaičiui n pradinių duomenų dydis yra $m = \log n$. O algoritmas, skaičiui n atliekantis n veiksmų, iš tiesų atliks eksponentinį veiksmų skaičių, kaip funkciją nuo pradinių duomenų dydžio: $n = 2^m$.

Geresnį pirminių skaičių paieškos algoritmą prieš kelis tūkstančius metų sugalvojo graikų matematikas *Eratostenas* (graikų k. Ερατοσθενης). Graikijoje tuo metu buvo rašoma ant papiruso arba odos, o vykdant šį algoritmą, sudėtinis skaičius buvo išbraukiamas jį perduriant aštria lazdele. Pabaigus vykdyti algoritmą, lentelė primindavo rėtį, todėl šis algoritmas vadinamas **Eratosteno rėčiu**.

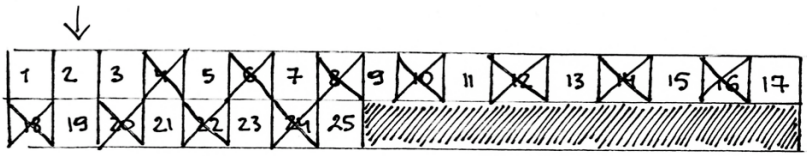
Surašykime visus skaičius nuo 1 iki n į eilę. Skaičių „sijojimas“ vyksta labai paprastai: eile keliaujama nuo 2 iki $\sqrt{n}$, ir, sutikus neišbrauktą skaičių k, išbraukiami visi k kartotiniai iki n (išskyrus patį skaičių k). Tokiu būdu „atsijojami“ sudėtiniai skaičiai, o visi likę yra pirminiai (išskyrus, žinoma, vienetą).

Naudodamiesi Eratosteno rėčiu raskime visus pirminius skaičius, ne didesnius kaip $n = 25$.

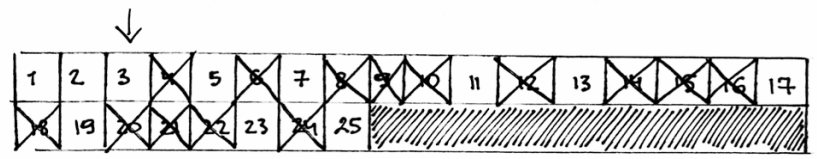
Į eilę surašome skaičius nuo 1 iki 25, o eile keliausime iki $\sqrt{25} = 5$.



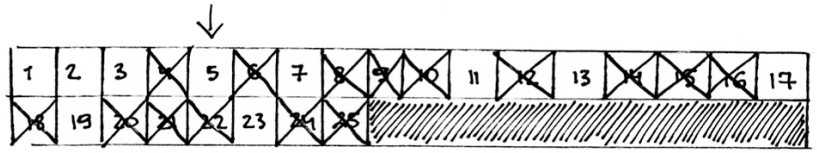
Pradedame nuo skaičiaus 2 – patį skaičių paliekame, o visus jo kartotinius išbraukiame.



Paeiname eile per vieną skaičių į dešinę (nuo 2 pereiname prie 3). 3 neišbrauktas, tad 3 paliekame, o visus kartotinius išbraukiame.



Vėl pereiname per vieną skaičių į dešinę. Skaičius 4 jau išbrauktas, tačiau 5 – ne. Išbraukiame visus skaičiaus 5 kartotinius:



Pasiekėme $5 = \sqrt{25}$, taigi darbą baigiame. Eilėje liko pirminiai skaičiai, ne didesni už 25, ir vienetas.

Dabar užrašykime algoritmą Paskalio kalba. Skaičių eilę vaizduosime loginiu masyvu `pirm`.

```
for k := 2 to n do
    pirm[k] := true;
for k := 2 to round(sqrt(n) + 1) do
    if pirm[k] then begin
        j := 2 * k;
        while (j <= n) do begin
            pirm[j] := false;
            j := j + k;
        end;
    end;
```

Šis algoritmas reikalauja O(n) atminties (loginiam masyvui). Turbūt ne taip akivaizdu, kad algoritmas reikalauja O(n·log(log n)) laiko –

šio fakto neįrodinėsime. Iš tiesų algoritmo sudėtingumas beveik tiesinis.

Kartą įvykdę Eratosteno rėčio algoritmą, galime per konstantinį (O(1)) laiką patikrinti, ar skaičius iš intervalo 1..*n* yra pirminis, – tereikia patikrinti atitinkamą masyvo elementą.

Abu aptartus algoritmus galima naudoti kartu. Įsivaizduokime, jog tenka tikrinti, ar dideli skaičiai (iki 2^{31}) yra pirminiai. Tiek atminties skirti negalime, todėl negalime naudoti Eratosteno rėčio algoritmo. Tačiau Eratosteno rėčiu suradę visus pirminius skaičius iki $\sqrt{2^{31}} \approx 46341$ ir perkėlę į atskirą masyvą, juos galime naudoti kaip potencialius daliklius vietoj visų skaičių iš intervalo $2 .. \sqrt{n}$. Tarkime, visi pirminiai skaičiai iki $\sqrt{2^{31}}$ iš eilės surašyti masyve `p`. Tuomet ankstesnę patikrinimo, ar skaičius pirminis, funkciją galime pakeisti spartesne:

```
function pirminis(n : longint) : boolean;
var i,              { masyvo p indeksas }
    sn : longint;   { riba, iki kurios ieškosime daliklių }
begin
    pirminis := true;
    sn := round(sqrt(n) + 1);
    i  := 1;
    while pirminis and (p[i] < sn) do
        if n mod p[i] = 0 then
            pirminis := false
        else
            i := i + 1;
end;
```

3.5 Pirminių skaičių paieška tęsiasi

8 pav.
Marinas Mersenas
(1588–1648)

Pirminių skaičių yra be galo daug, tad didžiausio jų ir negali būti. Nuo senų laikų lenktyniaujama, kas atras didesnį pirminį skaičių. XVII amžiuje matematikai ėmė intensyviai ieškoti dėsningumų pirminių skaičių sekoje. Tuo metu gyvenęs filosofas ir matematikas vienuolis Marinas Mersenas (*Marin Mersenne*) pastebėjo, kad daug skaičių, užrašomų pavidalu 2^p-1, kur p – pirminis skaičius, taip pat yra pirminiai. Tokie pirminiai skaičiai dabar vadinami Merseno pirminiais. Atsiradus kompiuteriams, šie iš karto buvo pasitelkti pirminių skaičių paieškai. 1997 metais pirminių skaičių paieškai buvo sukurtas GIMPS (angl. *The Great Internet Mersenne Prime Search*) paskirstytų skaičiavimų projektas. Visi norintys dalyvauti šiame projekte gali atsisiųsti į savo kompiuterį programinę įrangą, kuri išnaudos laisvą jūsų kompiuterio procesoriaus darbo laiką: parsisiųs ir atliks tam tikrą užduočių paketą, o rezultatus perduos į centrinį serverį. Šio projekto vykdytojai jau rado net 9 didžiausius (tuo metu) Merseno pirminius skaičius. 1999 m. EFF (*Electronic Frontier Foundation*) paskelbė šimtatūkstantines premijas pirmiesiems, atradusiems pirminius skaičius, turinčius labai daug

9 pav. 1963 m. didžiausio tuo metu žinomo pirminio skaičiaus garbei buvo skirtas pašto ženklas

(nuo 1 000 000) skaitmenų. Pirmoji 50 000 dolerių premija jau buvo išmokėta 2000 metais GIMPS projekto dalyviui, atradusiam Merseno pirminį, sudarytą iš 2 098 960 skaitmenų. 2005 gruodžio 15 dieną buvo atrastas 43-iasis Merseno pirminis skaičius $2^{30\,402\,457}-1$, sudarytas iš 9 152 052 skaitmenų. Tad iki antrosios, dvigubai didesnės, premijos už iš ne mažiau kaip 10 000 000 skaitmenų sudarytą pirminį skaičių laukti lieka neilgai.

4 REKURSIJA

In order to understand recursion, one must first understand recursion.
Norint suprasti rekursiją, pirma reikia suprasti rekursiją.
Populiarus humoristinis posakis

Šis humoristinis posakis gana gerai nusako rekursijos esmę: algoritmas yra **rekursyvus**, jei bent vienas iš jo žingsnių yra to paties algoritmo atlikimas su kitais (dažniausiai mažesniais) duomenimis.

Pažintį su rekursija pradėkime nuo geometrinės figūros, vadinamos Mengerio kempine[11] (ji yra fraktalinis[12] kūnas), konstravimo algoritmo:

1 žingsnis Paimamas kubas.

2 žingsnis Kubas suskaidomas į 27 vienodo dydžio kubelius.

3 žingsnis Pašalinamas kubo viduryje esantis kubelis, taip pat dar 6 kiekvienos sienos viduryje esantys kubeliai.

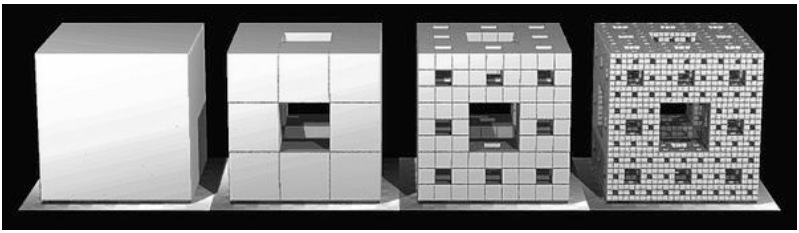
4 žingsnis Toliau su kiekvienu likusiu kubeliu veiksmai kartojami nuo antro žingsnio.

---

[11] Mengerio kempinės iliustracija paimta iš http://en.wikipedia.org/wiki/Menger_sponge.

[12] Terminą „fraktalas“ (išvertus iš lotynų kalbos tai reiškia *sudužęs*, *suskilęs*) pasiūlė B. Mandelbrotas. Jis norėjo viena sąvoka aprašyti tokius gamtoje pasitaikančius darinius kaip debesys, kalnai, žaibai arba tam tikrus geometrinius objektus. Pasirodo, visi šie objektai yra fraktalai ir turi tam tikrų bendrų savybių. Fraktalų geometrijos atradimas yra vienas didžiausių XX amžiaus matematikos pasiekimų, ši geometrija plačiai taikoma įvairiose srityse, pavyzdžiui, kuriant fantastinius gamtą imituojančius peizažus filmuose.

Pateiktas konstravimo algoritmas yra rekursyvus, nes ketvirtame žingsnyje nurodoma tą patį algoritmą taikyti kitiems duomenims.



10 pav. Menger io kempinė

4.1 Rekursyvios funkcijos

Aukšto lygio programavimo kalbos suteikia galimybę aprašyti rekursyvias funkcijas, t. y. funkcijas, kurios iškviečia pačios save. Kiekvieną kartą kreipiantis į funkciją, įsimenamas grįžimo adresas, padaromos parametrų kopijos ir sukuriami nauji lokalūs funkcijos kintamieji.

Tai organizuojama **dėklo** (angl. *stack*) duomenų struktūra. Ši struktūra veikia LIFO (angl. *Last in First out*) principu: nauji duomenys dedami į dėklo „viršų“ ir imami nuo „viršaus“, t. y. imant duomenis visada paimamas paskutinis padėtas duomuo. Taigi kiekvieną rekursyvų algoritmą galima realizuoti nenaudojant rekursijos, o suprogramuojant ir panaudojant savo dėklo duomenų struktūrą.

11 pav. Lėkščių krovimas į stirtą primena dėklą — paskutinė padėta lėkštė bus paimta pirmoji

Rekursyvi funkcija su ją iškvietusia funkcija (savo pačios „kopija“) gali bendrauti tik parametrais bei globaliais kintamaisiais.

Panagrinėkime keletą paprastų rekursyvių funkcijų. Vieną, beje, jau matėme – Euklido algoritmas DBD rasti gali būti užrašomas rekursyviai.

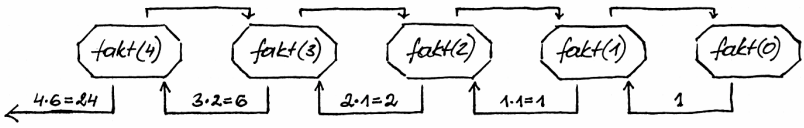
Kitas pavyzdys – skaičiaus faktorialas:

$$0! = 1$$
$$n! = n(n-1)!, \text{ jei } n > 0$$

Galime parašyti skaičiaus faktorialą skaičiuojančią funkciją:

```
function fakt(n : integer) : longint;
begin
    if n = 0 then
        fakt := 1
    else
        fakt := n * fakt(n - 1);
end;
```

Kreipinio `fakt(4)` vykdymą iliustruoja žemiau pateiktas paveikslas:



Atlikus kreipinį `fakt(n)`, iš viso bus įvykdyta $(n + 1)$ funkcijų kvietimų, taigi šios funkcijos sudėtingumas yra $O(n)$. Šis būdas yra lėtesnis už faktorialo skaičiavimą ciklu, kadangi funkcijos iškvietimas yra kur kas sudėtingesnis procesas už ciklo iteraciją.

Kitas rekursyvios funkcijos pavyzdys – Fibonačio skaičiai. 1202 metais italų matematikas Leonardo Pisano, vadinamas Fibonačiu

(*Fibonacci*), sugalvojo uždavinį: triušių pora kas mėnesį atsiveda po du triušiukus (patinėlį ir patelę), o iš atvestųjų triušiukų po dviejų mėnesių jau gaunamas naujas prieauglis. Kiek triušių bus po metų, jei metų pradžioje buvo viena jauniklių pora? Triušių skaičių kiekvieną mėnesį nusakys seka 1, 1, 2, 3, 5, 8, 13, 21, 34..., o šie skaičiai yra vadinami Fibonačio skaičiais. Juos taip pat galima skaičiuoti rekursyviai:

$F_1 = F_2 = 1$
$F_n = F_{n-1} + F_{n-2}$, jei $n > 2$

```
function F(n : integer) : longint;
begin
    if n <= 2 then
        F := 1
    else
        F := F(n - 1) + F(n - 2);
end;
```

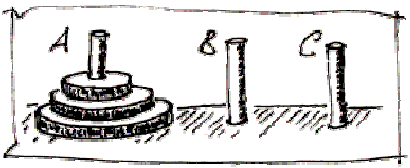
Nors ši funkcija atrodo tokia pat paprasta, kaip ir faktorialo, jos sudėtingumas yra eksponentinis[13]. Taip yra todėl, kad kiekviena funkcija iškviečia net dvi kitas, antrines funkcijas, o joms perduodami argumentai sumažinami tik pastoviu dydžiu. Iškvietus `F(45)`, atsakymo tektų palaukti.

Pastebėkime, kad visi minėti uždaviniai pasižymi viena bendra savybe: spręsdami uždavinį, turime išspręsti analogiškus, bet mažesnius uždavinius. Pavyzdžiui, jei norime suskaičiuoti $n!$, turime išspręsti mažesnį uždavinį – suskaičiuoti $(n-1)!$, o jei norime rasti DBD(25, 15) (pagal Euklido algoritmą), turime rasti DBD(15, 10).

[13] Fibonačio skaičius galima skaičiuoti efektyviai (per tiesinį laiką), masyve įsimenant jau apskaičiuotas reikšmes; apie tai skaitykite 12.2 skyrelyje.

4.2 *Hanojaus bokštų uždavinys*

Išspręsime klasikinį *Hanojaus bokštų uždavinį*, kurį 1883 metais suformulavo prancūzų matematikas Eduardas Lukas (*Edouard Lucas*).



12 pav. Pavyzdys su trimis diskais

Duoti trys stiebai ir aštuoni skirtingo dydžio diskai. Iš pradžių visi šie diskai sumauti ant pirmojo stiebo: apačioje pats didžiausias diskas, ant jo – mažesnis ir t. t. Viršuje užmautas pats mažiausias iš diskų.

Užduotis: *reikia perkelti visus diskus nuo pirmojo stiebo ant paskutinio laikantis šių taisyklių:*

- *Vienu ėjimu galima kelti tik vieną diską.*
- *Diską galima užmauti tik ant tuščio stiebo arba uždėti ant didesnio už jį disko.*
- *Atliekamų perkėlimų skaičius turi būti minimalus.*

Praplėsime standartinę uždavinio formuluotę: vietoj aštuonių diskų reikia perkelti n diskų. Stiebai pavadinti raidėmis A, B ir C. Parašykite programą, kuri atspausdintų, kaip perkelti visus diskus, laikantis minėtų taisyklių.

Panagrinėkime paprasčiausius atvejus[14]. Kai $n = 1$, diską perkeliame (ir uždavinį išsprendžiame) vienu žingsniu. Nesunku jį išspręsti, kai

[14] Kelių paprastų uždavinio atvejų sprendimas ranka įtraukia mus į užduotį, suteikia intuicijos ir dažnai privilioja geras idėjas! Taigi tai naudinga daryti olimpiadose.

$n = 2$, tam reikia trijų perkėlimų. Šiek tiek pagalvoję suvokiame, kad pakanka 7 perkėlimų uždaviniui išspręsti, kai $n = 3$.

Atkreipkite dėmesį, kad niekas nepasikeistų, jei uždavinyje būtų reikalaujama diskus perkelti ne ant dešiniojo, o ant vidurinio disko: atliktume tuos pačius ėjimus, tik diskus keltume ne ant dešiniojo, o ant vidurinio ir atvirkščiai.

Ko gi reikia, kad galėtume pagal taisykles perkelti *n*-ąjį (patį didžiausią) diską? Visų pirma, ant jo neturi būti jokių kitų diskų. Be to, dešinysis stiebas taip pat turi būti tuščias. Vadinasi, visi likę diskai turi būti jau perkelti ant vidurinio stiebo! Tik tuomet galėsime perkelti *n*-ąjį (didžiausią) diską.

Bandydami $(n - 1)$ mažesnių diskų perkelti ant vidurinio stiebo, galime visiškai nekreipti dėmesio į *n*-ąjį diską: jis nesutrukdys, kadangi yra didesnis už visus likusius diskus. Taigi $(n - 1)$ diskų perkėlimas yra visiškai tas pats, tik sumažintas, uždavinys. Taip pradedame įžvelgti rekursyvų uždavinio sprendimą, kurio bendra schema tokia:

Jei norime perkelti $n > 0$ diskų:

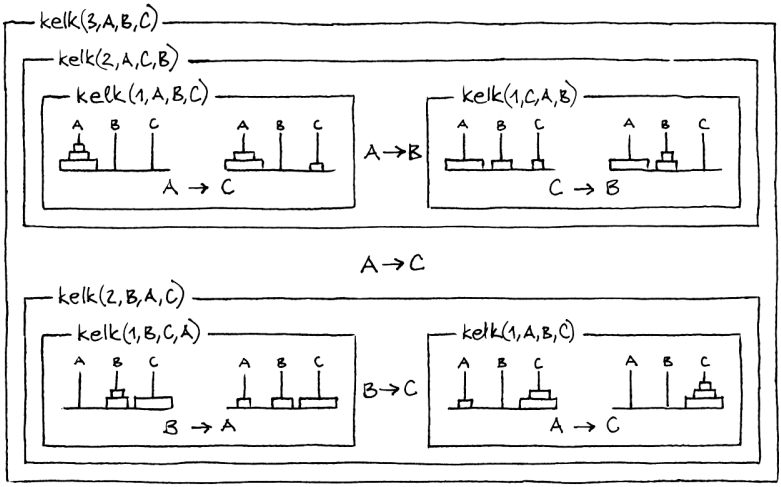
- *Visus mažesnius diskus perkeliame ant tarpinio stiebo.*
- *Perkeliame n–ąjį diską.*
- *Visus mažesnius diskus perkeliame ant galinio stiebo.*

Tegul `kelk` yra diskų perkėlinėjimo funkcija. Ji turi priklausyti nuo diskų, kuriuos reikia perkelti, skaičiaus. Be to, ji turi žinoti, nuo kurio ir ant kurio stiebo norima perkelti diskus. Tai nebus visada tie patys stiebai A ir C. Pavyzdžiui, jei norėsime *n* diskų perkelti nuo stiebo A ant stiebo C, turime $(n - 1)$ diską perkelti nuo stiebo A ant stiebo B (ta pati užduotis, tik kitas diskų skaičius ir stiebų vardai), o vėliau – nuo B ant C. Kintamuosius žymėsime `nuo`, `ant` ir `tarp`

(tarpiniam stiebui). Jei $n > 0$, diskus perkeliame remdamiesi aukščiau aprašyta taisykle, o jei $n = 0$, nereikia atlikti nieko – rekursija baigiama.

```
procedure kelk(n : integer; nuo, tarp, ant : char);
begin
    if n > 0 then begin
        kelk(n - 1, nuo, ant, tarp); { nuo –> tarp }
        { perkeliamas n-tasis diskas }
        writeln(nuo, ' -> ', ant);
        kelk(n - 1, tarp, nuo, ant)  { tarp –> ant }
    end
end;
```

Jei norime perkelti n diskų nuo stiebo A ant stiebo C, iškviečiame `kelk(n, 'A', 'B', 'C')`. Žemiau iliustruojamas procedūros veikimas, iškvietus `kelk(3, 'A', 'B', 'C')`:



Taigi procedūra atspausdins:

```
A -> C
A -> B
C -> B
A -> C
B -> A
B -> C
A -> C
```

Nuostabu, kad šiam, iš pirmo žvilgsnio sudėtingam, uždaviniui egzistuoja toks elegantiškas sprendimas.

Parodysime, jog aprašytuoju būdu kilnojant diskus perkėlimų skaičius yra mažiausias. Pažymėkime T_n mažiausią perkėlimų skaičių, reikalingą perkelti n diskų nuo vieno stiebo ant kito. Žinome, kad $T_0 = 0$, $T_1 = 1$, $T_2 = 3$ ir $T_3 = 7$.

Be to, iš ankstesnių samprotavimų seka, kad n diskų galima perkelti $T_{n-1} + 1 + T_{n-1} = 2T_{n-1} + 1$ perkėlimais, t. y.:

$$T_n \leq 2T_{n-1} + 1 \quad (1)$$

Kita vertus, ar galime ką nors atlikti geriau? Anksčiau ar vėliau būtinai teks perkelti n-tąjį (didžiausią) diską. Prieš tai likusieji $n - 1$ diskų privalės atsidurti ant vidurinio stiebo, o tam reikės bent T_{n-1} (minimalaus skaičiaus) perkėlimų. Vieno perkėlimo reikės n-ajam diskui, ir pagaliau dar bent T_{n-1} perkėlimų mažesniems diskams perkelti ant viršaus. Todėl:

$$T_n \geq 2T_{n-1} + 1 \quad (2)$$

Iš (1) ir (2) nelygybių gauname, kad $T_n = 2T_{n-1} + 1$.

Taigi T_n galime apskaičiuoti pagal rekurentinį sąryšį:

$T_0 = 0$
$T_n = 2T_{n-1} + 1$, jei $n > 0$ (3)

Pavyzdžiui, $T_4 = 2T_3 + 1 = 15$.

Tačiau rekurentinis sąryšis neatsako į klausimą, koks procedūros `kelk` sudėtingumas. Matyti, kad, diskų skaičių padidinus vienetu, ėjimų skaičius maždaug padvigubėja. Norėdami būti tikri, išspręsime rekurentinį sąryšį.

Pažymėkime U_n skaičių, vienetu didesnį už T_n: t. y. $U_n = T_n + 1$.

Pridėję prie (3) lygybių po vienetą, gauname:

$T_0 + 1 = 1$
$T_n + 1 = 2T_{n-1} + 2 = 2(T_{n-1} + 1)$, jei $n > 0$

Taigi:

$U_0 = 1$
$U_n = 2U_{n-1}$, jei $n > 0$

Iš čia matyti, kad $U_n = 2U_{n-1} = 2^kU_{n-k} = 2^n$, vadinasi, $T_n = U_n - 1 = 2^n - 1$.

Procedūros `kelk`, perkeliančios *n* diskų, atliekamų žingsnių skaičius proporcingas T_n, taigi šios procedūros sudėtingumas yra $O(2^n)$. Palyginkime procedūrą `kelk` su Fibonačio skaičių skaičiavimo funkcija `F` – kiekviena jų atlieka du rekursyvius kreipinius, argumentą sumažindamos tik pastoviu dydžiu. Tai lemia eksponentinį sudėtingumą.

4.3 Rekursijos užbaigimas

Yra jūroj paskandinta dėžė, toj dėžėj yra zuikis,
tam zuiky – karvelis, tam karvely – kiaušinis,
tam kiaušiny – adata, ją perlaužus raganius mirs.
Lietuvių liaudies pasaka

Kiekvienoje rekursinėje procedūroje turi būti numatyti visi ribiniai atvejai, kuriuos pasiekus rekursija nutraukiama. Ribinis atvejis – randama ir sulaužoma adata – numatytas netgi pasakoje, tuo labiau jo nereiktų pamiršti programuojant.

Panagrinėkime analizuotų pavyzdžių ribinius atvejus. Skaičiuojant skaičiaus n faktorialą, ribinis atvejis yra $n = 0$ ($0! = 1$), ieškant n-ojo Fibonačio skaičiaus – $n \leq 2$ ($F_1 = F_2 = 1$). Ieškant didžiausiojo bendro skaičių a ir b daliklio – rekursija baigiama, kai $b = 0$, keliant diskus Hanojaus bokštų uždavinyje – kai reikia perkelti 0 (t. y. nebereikia kelti nė vieno) diskų.

Viena vertus, būtina užtikrinti, kad rekursiniame procese *būtinai* bus *pasiekiamas* kuris nors ribinis atvejis, kita vertus – reikia nepamiršti numatyti *visų* ribinių atvejų. Jei karalaitis kiaušinyje rastų ne adatą, o obuolį, jis atsidurtų keblioje padėtyje...

5 PERRINKIMAS IR GRĮŽIMO METODAS

Modern computers are so fast that brute force can be an effective and honourable way to solve problems.
Šiuolaikiniai kompiuteriai yra tokie spartūs, kad perrinkimas gali būti efektyvus ir garbingas būdas uždaviniams spręsti.
Steven S. Skienna, Miguel A. Revilla, „Programming Challenges“

1852 metais matematikas F. Gatris (*Francis Guthrie*) paskelbė hipotezę, teigiančią, jog kiekvienam žemėlapiui nuspalvinti taip, kad jokios dvi gretimos valstybės nebūtų nuspalvintos ta pačia spalva, pakanka keturių spalvų. Daugelis matematikų siūlė šios hipotezės įrodymus, tačiau vis išaiškėdavo, kad jie neteisingi[15]. Hipotezė pagaliau tapo teorema (buvo įrodyta) 1976 metais, o dalis įrodymo rėmėsi kompiuteriu išnagrinėtomis 1476 situacijomis. Kompiuterio programa veikė šimtus valandų, o žmonėms nepakako ir šimto metų. Taigi nors perrinkimo metodai dažnai būna neefektyvūs, spartėjant kompiuteriams šis sprendimo būdas (visų galimų sprendinių išbandymas) tam tikrais atvejais gali būti priimtinas, ypač jei perrinkimą pavyksta optimizuoti.

Formaliai **perrinkimą** galima apibrėžti kaip uždavinių sprendimo metodą, kai išbandomi visi galimi sprendiniai.

Šiame skyrelyje susipažinsime su patogiu metodu perrinkimui realizuoti – grįžimo metodu. **Grįžimo metodas** (angl. *Backtracking*) – tai sistemingas būdas spręsti uždaviniams, kurių sprendinys yra kintamųjų p_1, p_2, ..., p_n reikšmių rinkinys, tenkinantis kokius nors reikalavimus. Prisiminkime, pavyzdžiui, *Keliaujančio pirklio uždavinį*[16]: šiuo atveju kintamųjų p_1, p_2, ..., p_n reikšmėms

15 Keletas įrodymų buvo paneigta praėjus tik 11 metų po jų paskelbimo.

16 Žr. 1.7 skyrelį.

reikia priskirti skirtingus miestų numerius taip, kad ši miestų aplankymo tvarka ir būtų pageidaujamas maršrutas.

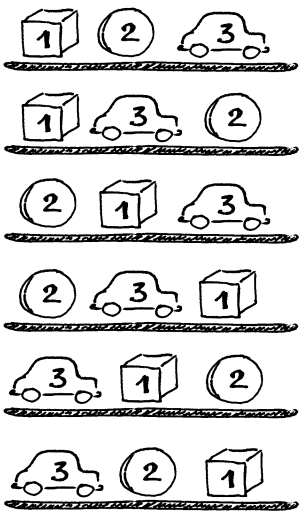
Pagrindinė grįžimo metodo idėja tokia: paeiliui renkamos visų galimų kintamųjų reikšmės ir tikrinama, ar tenkinami reikalavimai, o radus sprendinį arba situaciją, kai reikalavimai netenkinami, grįžtama per vieną žingsnį atgal ir parenkama nauja atitinkamo kintamojo reikšmė.

Programuojant grįžimo metodą dažnai naudojama rekursija. Panagrinėsime abstrakčių kombinatorinių uždavinių sprendimų schemas bei porą konkrečių uždavinių.

5.1 Kėlinių generavimas

Sakykime, parduotuvės lentynoje vienoje eilėje reikia išdėlioti *n* skirtingų prekių. Raskime visus skirtingus būdus, kaip tai padaryti. Uždavinys yra ekvivalentus visų *n* ilgio kėlinių be pasikartojimų generavimo uždaviniui.

Kas gi tas kėlinys be pasikartojimų? Tarkime, turime aibę iš *n* elementų. Kiekviena visų (skirtingų) *n* elementų seka vadinama **kėliniu be pasikartojimų**. Taigi kėliniai vienas nuo kito skiriasi tik elementų išsidėstymu vienas kito atžvilgiu.



13 pav. Visi galimi trijų prekių išdėstymo lentynoje būdai

Parašykime algoritmą, kuris išspausdintų visus prekių išdėstymo būdus. Prekes laikysime sunumeruotomis nuo 1 iki n.

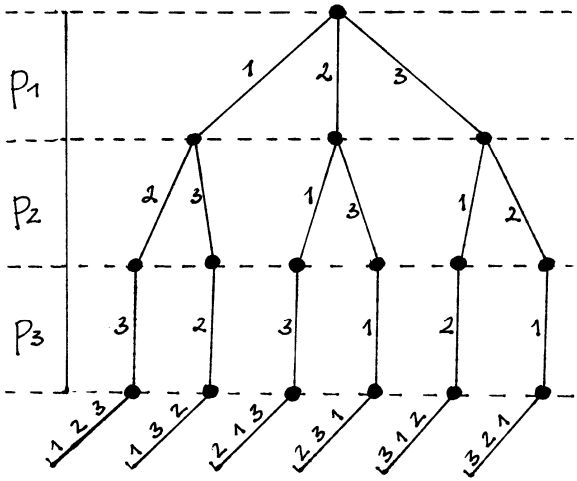
Atkreipkite dėmesį – šis uždavinys priskiriamas įžangoje minėtai uždavinių klasei: m-ojoje vietoje padėtą prekę (prekės numerį) pažymėjus $p_{m,}$ reikia rasti kintamųjų p_1, ..., p_n reikšmių (kintančių nuo 1 iki n) rinkinius, tenkinančius vieną reikalavimą – visos reikšmės turi būti skirtingos; tuos rinkinius atspausdinti.

Uždavinį galima išreikšti rekursyviai, t. y. suskaidyti į tokius pat, tik mažesnius, uždavinius. Tegu procedūra `generuok(m, n)` priskirs reikšmes kintamiesiems nuo p_m-ojo iki p_n-ojo. Tuomet jos veikimas galėtų būti toks:

- Jei $m \leq n$, imti po vieną visas dar lentynoje nepadėtas prekes ir su kiekviena atlikti tokius veiksmus:
    - Prekę padėti į m-ąją poziciją lentynoje (p_m := prekės numeris).
    - Sudėti į lentyną likusias prekes (priskirti reikšmes kintamiesiems nuo p_{m+1} iki p_n) – toks pat uždavinys, taigi iškviesti `generuok(m + 1)`.
    - Prekę, esančią m-ojoje pozicijoje, paimti nuo lentynos.
- Jei $m > n$, tai ši procedūra iškviesta jau išdėliojus visas prekes lentynoje, todėl atspausdiname kintamųjų p_1, p_2, ..., p_n reikšmes.

Norint patikrinti, kurios prekės jau sudėtos į lentyną, galima peržiūrėti jau priskirtas reikšmes. Tačiau paprasčiau ir efektyviau paskirti globalų loginį masyvą `panaudotas`, ir, padėjus į lentyną prekę su numeriu `i`, pažymėti, jog šis numeris jau panaudotas (`panaudotas[i] := true`), o paėmus prekę nuo lentynos – atstatyti buvusią reikšmę (`panaudotas[i] := false`).

```
const MAXN = 20;          { didžiausia n reikšmė }

var p : array [1..MAXN] of integer;
    panaudotas : array [1..MAXN] of boolean;

procedure spausdink(m: integer);
var i : integer;
begin
    for i := 1 to m do
        write(p[i], ' ');
    writeln;
end;

procedure generuok(m, { parenkamas elementas m-ajai pozicijai }
                  n : integer);
var i : integer;
begin
    { jei m > n, tai ši procedūra iškviesta jau sugeneravus visą kėlinį }
    if m > n then
       spausdink(n)
    else
        for i := 1 to n do
            if not panaudotas[i] then begin
                panaudotas[i] := true;
                p[m] := i;
                generuok(m + 1, n);
                panaudotas[i] := false;
            end;
end;
```



14 pav. Procedūros `generuok` *vykdymą vaizduojantis medis (n = 3)*

Kad galėtume išspausdinti visas trijų prekių išdėliojimo lentynoje tvarkas, įvykdome:

```
n := 3;
for i := 1 to n do
    panaudotas[i] := false;
generuok(1, n);
```

Parašytą procedūrą nesunku pritaikyti kitiems uždaviniams – vietoj spausdinimo galima atlikti kokius nors kitus veiksmus. Spausdinimą iškėlėme į atskirą procedūrą norėdami paryškinti sprendimo struktūrą.

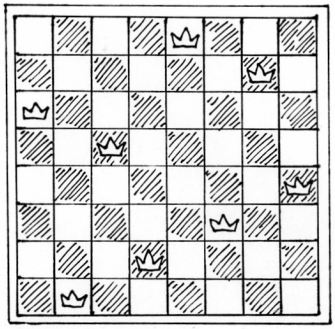
Koks gi parašytos programos sudėtingumas, t. y. kaip atliekamų veiksmų skaičius priklauso nuo n? Algoritmas generuoja visus įmanomus skaičių nuo 1 iki n išdėstymo į eilę būdus. Kiek jų yra? Pirmąjį skaičių galima parinkti n būdų, antrąjį skaičių – $(n-1)$

būdu (kadangi vienas skaičius jau pasirinktas), trečiąjį skaičių – $(n-2)$ būdais (du skaičiai jau parinkti) ir t. t. Gauname, kad yra $n(n-1)(n-2)\cdot\ldots\cdot 2\cdot 1 = n!$ skirtingų būdų išdėstyti n skaičių į eilę. Taigi procedūros `generuok` sudėtingumas yra $O(n!)$. Pavyzdžiui, kai $n = 13$, tai vieną atspausdintą eilutę sudaro apie 30 simbolių, o eilučių yra $13! = 6227020800$ ir programa spausdintų daugiau nei 150 gigabaitų teksto... (jei, žinoma, sulauktume veikimo pabaigos).

5.2 *Aštuonių valdovių uždavinys*

Išspręsime klasikinį *aštuonių valdovių* uždavinį.

Užduotis. *8×8 dydžio šachmatų lentoje reikia išdėlioti 8 valdoves taip, kad jokiu būdu neatsidurtų dvi vienoje eilutėje, stulpelyje arba įstrižainėje (t. y. nė viena negalėtų nukirsti kitos tolesniu ėjimu). Uždavinio formuluotę išplėsime ir ieškosime, kaip n valdovių surikiuoti n × n dydžio lentoje.*



15 pav. Aštuonių valdovių išdėstymo pavyzdys

Šį uždavinį taip pat spręsime grįžimo metodu. Pavyzdžiui, lentos langelius sunumeravę nuo 1 iki n^2, kiekvienai valdovei galime skirti po vieną langelį (numerį) taip, kad būtų tenkinama uždavinio sąlyga. Tačiau spręsdami uždavinį šiuo būdu, turėtume išnagrinėti labai didelį variantų skaičių. Variantų skaičius, kuriuo aštuonioms valdovėms galima paskirstyti langelių numerius nuo 1 iki 64 yra $64\cdot 63\cdot 62\cdot 61\cdot 60\cdot 59\cdot 58\cdot 57 = 178\,462\,987\,637\,760$ būdų.

Be abejo, didžioji dalis šių variantų visiškai neįdomūs, nes labai tikėtina, kad kurios nors dvi valdovės atsidurs toje pačioje eilutėje, stulpelyje arba įstrižainėje. Atkreipkime dėmesį – kiekviename stulpelyje turės atsidurti lygiai viena valdovė; stulpelių yra tiek, kiek ir valdovių, o viename stulpelyje dvi valdovės stovėti negali.

Taigi galima šiek tiek kitaip vykdyti perrinkimą. Tegu p_k yra valdovės, stovinčios k-ajame stulpelyje, eilutės numeris. Kintamiesiems $p_1, p_2, ..., p_n$ reikia priskirti reikšmes nuo 1 iki n taip, kad jokios dvi valdovės neatsidurtų vienoje eilutėje arba įstrižainėje.

Šitaip atliekant perrinkimą, net nepaisant įstrižainių apribojimo, nagrinėjamų variantų bus tik $n!$. Palyginkite – aštuonių valdovių atveju teks išnagrinėti $8! = 40\,320$ variantų vietoj 178 462 987 637 760.

Perrenkant valdovių rikiavimo būdus, visai nesudėtinga sekti, kuriose eilutėse valdovės jau pastatytos – tam galima skirti loginį masyvą.

Tačiau kaip elgtis su įstrižainėmis? Patikrinti, ar dvi valdovės nestovi vienoje įstrižainėje, galima sustačius visas valdoves. Tačiau išsisuksime paprasčiau (ir efektyviau) pastebėję, kad įstrižaines taip pat nesunku sunumeruoti: vienoje įstrižainėje esančių langelių eilutės ir stulpelio numerių suma arba skirtumas yra pastovus.

Taigi žinodami langelio koordinates (stulpelio ir eilutės numerius), galime pasakyti, kuriai įstrižainei priklauso šis langelis. Įstrižainėms skiriame du loginius masyvus su indeksais atitinkamai $[2..2n]$ ir $[-n + 1..n - 1]$, kuriuose žymėsime, ar įstrižainės jau užimtos.

	1	**2**	**3**	**4**	**5**
1	2	3	4	5	6
2	3	4	5	6	7
3	4	5	6	7	8
4	5	6	7	8	9
5	6	7	8	9	10

	1	**2**	**3**	**4**	**5**
1	0	-1	-2	-3	-4
2	1	0	-1	-2	-3
3	2	1	0	-1	-2
4	3	2	1	0	-1
5	4	3	2	1	0

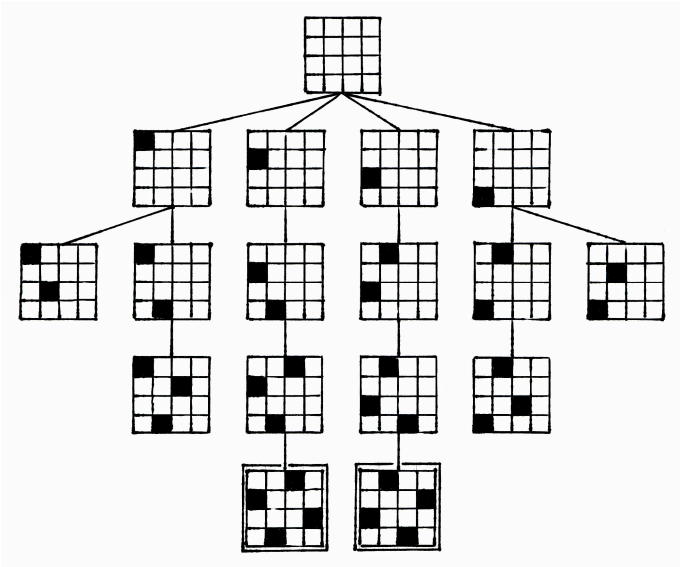
16 pav. Kairėje pavaizduotos įstrižainės numeruojamos eilutės ir stulpelio numerių suma, dešinėje – skirtumu

Parašysime procedūrą `statyk(k, n)`, perrenkančią sprendinius grįžimo metodu, kuri visais įmanomais būdais sudėlios lentoje valdoves nuo *k*-osios iki *n*-osios. *k*-oji valdovė bus statoma *k*-ajame stulpelyje. Taigi procedūra turi bandyti pastatyti *k*-ąją valdovę nepažeisdama apribojimų, o pastačius – pažymėti užimtas eilutę ir įstrižaines, ir iškviesti `statyk(k + 1, n)`.

Jei iškvietus procedūrą parametro *k* reikšmė viršija *n* ($k > n$), tai reiškia, kad ši procedūra buvo iškviesta sudėliojus visas *n* valdovių, taigi radus sprendinį. Viena vertus, sudėliojus visas *n* valdovių, procedūros `statyk` būtų galima nebekviesti, tačiau dėl šio papildomo iškvietimo programa tampa paprastesnė ir aiškesnė. Tai dažnai naudojama rekursyviose procedūrose.

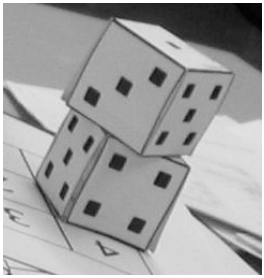
Procedūroje skaičiuosime, kiek yra sprendinių, t. y. būdų išdėlioti valdoves lentoje. Tačiau nesunku modifikuoti procedūrą taip, kad ši rastus sprendinius išspausdintų – tuomet dar reikėtų saugoti, kur lentoje statomos valdovės.

```
const MAXN = 12;

var eilutė : array [1..MAXN] of boolean;
    įstr1 : array [2..2 * MAXN] of boolean;
    įstr2 : array [-MAXN + 1..MAXN - 1] of boolean;
    sprendinių_sk : longint;

procedure statyk(k, { valdovė statoma k-ajame stulpelyje }
                 n : integer { reikia pastatyti n valdovių });
var i : integer;
begin
    if k > n then { rastas sprendinys }
        sprendinių_sk := sprendinių_sk + 1
    else
        for i := 1 to n do
            if not (eilutė[i] or
                    įstr1[i + k] or
                    įstr2[i - k])
            then begin
                eilutė[i] := true;
                įstr1[i + k] := true;
                įstr2[i - k] := true;
                { bandoma pastatyti likusias valdoves }
                statyk(k + 1, n);
                eilutė[i] := false;
                įstr1[i + k] := false;
                įstr2[i - k] := false;
            end;
end;
```



17 pav. Valdovių uždavinio rekursijos medis, kai n=4

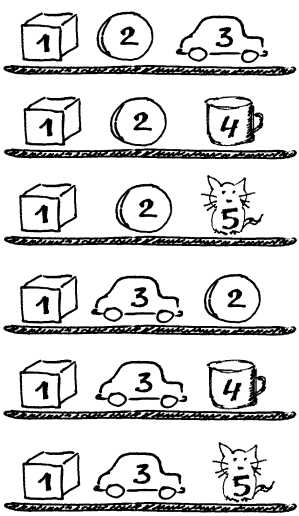
5.3 Gretiniai, deriniai ir poaibiai



18 pav. Azartiniai žaidimai buvo dingstis atsirasti kombinatorikai

Ankstesniame skyrelyje (5.1) nagrinėjome, kiek ir kokių kombinacijų galima sudaryti iš įvairių objektų, kad būtų tenkinamos vienokios ar kitokios sąlygos. Šitai nagrinėja matematikos šaka, vadinama *kombinatorika*, kuri atsirado XVI amžiuje išpopuliarėjus azartiniams žaidimams. Pirmieji kombinatorikos uždaviniai ir buvo susiję su šiais žaidimais, pavyzdžiui, buvo tiriama, keliais būdais galima išmesti kokį nors taškų skaičių, žaidžiant dviem arba trimis kauliukais.

Kombinatorikos žinių prireikia sprendžiant įvairius olimpiadinius uždavinius. Šiame skyrelyje glaustai išdėstysime, kaip generuoti kitus junginius rekursiniais algoritmais[17].



19 pav. Keletas gretinių iš penkių prekių po tris

Gretiniai. Grįžkime prie pavyzdžio su parduotuve. Sakykime, turime n skirtingų prekių, kurias reikia išdėlioti lentynoje; deja, lentynoje telpa tik k prekių ir visų prekių iš karto parodyti pirkėjams nepavyks. Reikia rasti visus būdus, kuriais galima išdėlioti prekes lentynoje. Tuščių vietų likti lentynoje negali.

Kitaip sakant, reikia rasti visus **gretinius be pasikartojimų** iš n elementų po k. Uždavinys labai panašus į jau nagrinėtą kėlinių be pasikartojimų generavimo uždavinį, tiesiog iš n elementų renkame tik k ($k \le n$).

```
const MAX = 20;          { didžiausia n ir k reikšmė }

var p : array [1..MAX] of integer;
    panaudotas : array [1..MAX] of boolean;
```

[17] Yra efektyvesnių (nerekursinių) kombinatorinius objektus generuojančių algoritmų, tačiau rekursiniai algoritmai yra intuityvesni ir lengviau realizuojami.

```
procedure generuok(m, { parenkamas elementas m-ajai pozicijai }
                      n, k : integer);
var i : integer;

begin
    { jei m > k,
      tai ši procedūra iškviesta jau sugeneravus visą gretinį }
    if m > k then
       spausdink[18](k)
    else
        for i := 1 to n do
            if not panaudotas[i] then begin
                panaudotas[i] := true;
                p[m] := i;
                generuok(m + 1, n, k);
                panaudotas[i] := false;
            end;
end;
```

Norėdami gauti visus gretinius iš 5 po 3, į procedūrą kreipiamės:

```
n := 5;
k := 3;
for i := 1 to n do
    panaudotas[i] := false;
generuok(1, n, k);
```

Suskaičiuosime, kiek gali būti skirtingų gretinių be pasikartojimų, tuo pačiu įvertinsime ir algoritmo sudėtingumą. Pirmąją prekę galime rinktis iš visų n prekių, antrąją prekę – iš $(n-1)$ prekės ir t. t. k-ąją prekę galime rinktis iš $(n-k+1)$ prekių.

Gretinių be pasikartojimų iš n elementų po k skaičius žymimas A_n^k ir lygus

$$A_n^k = n\,(n-1)(n-2)....(n-k+1).$$

[18] Procedūros `spausdink` tekstą rasite 5.1 skyrelyje.

Deriniai. Generuodami gretinius atsižvelgėme į prekių išdėstymą lentynose. Pamėginkime rasti visus būdus, kuriais galima išdėstyti *n* skirtingų prekių lentynoje, kurioje telpa tik *k* prekių (lentynoje neturi likti tuščių vietų) nekreipiant dėmesio į prekių išdėstymą, t. y. kai rūpi tik tai, kokios prekės yra lentynoje, tačiau nesvarbu, kokia tvarka jos ten išdėliotos. Kitaip sakant, reikia sugeneruoti visus **derinius be pasikartojimų** iš *n* elementų po *k*.

Derinius galima generuoti kaip gretinius, laikantis vienos papildomos taisyklės: prekės dėliojamos taip, kad jų numeriai sudarytų didėjančią seką, t. y. $p_1 < p_2 < p_3 < \ldots < p_k$. Derinius generuojančiai rekursinei procedūrai prireiks vieno papildomo parametro, kuris rodytų, nuo kurio elemento galime rinkti tolesnius elementus.

20 pav. Keletas derinių iš penkių prekių po tris (tvarka deriniuose nesvarbi)

```
const MAX = 20;          { didžiausia n ir k reikšmė }
var p : array [1..MAX] of integer;
    panaudotas : array [1..MAX] of boolean;
procedure generuok(nuo,  { bus renkamasi tik iš elementų,
                           didesnių arba lygių „nuo“ }
                   m,  { parenkamas elementas m-ajai pozicijai }
                   n, k: integer);
var i : integer;
begin
    { jei m > k, tai ši procedūra iškviesta jau sugeneravus visą derinį }
    if m > k then spausdink[19](k)
```

[19] Procedūros `spausdink` tekstą galite rasti 5.1 skyrelyje.

```
    else
        for i := nuo to n do
            if not panaudotas[i] then begin
                panaudotas[i] := true;
                p[m] := i;
                generuok(i + 1, m + 1, n, k);
                panaudotas[i] := false;
            end;
end;
```

Norėdami gauti visus skirtingus derinius iš 5 elementų po 3, į procedūrą kreipiamės:

```
n := 5;
k := 3;
for i := 1 to n do
    panaudotas[i] := false;
generuok(1, 1, n, k);
```

Beliko apskaičiuoti, kiek gali būti skirtingų derinių be pasikartojimų iš n po k. Šį skaičių pažymėkime C_n^k.

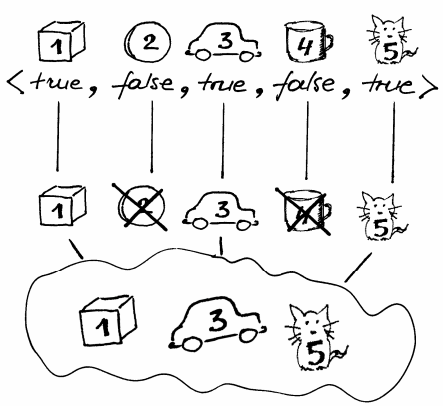
Sakykime, turime konkretų derinį. Jei paimtume visus jo perstatymus, gautume visus kėlinius be pasikartojimų iš tų k derinį sudarančių elementų. Tokių kėlinių gali būti $k!$

O jei kartu paimtume visus kiekvieno galimo derinio perstatymus, gautume visus gretinius be pasikartojimų iš n elementų po k. Žinome, kad jų gali būti $A_n^k = n(n-1)(n-2)....(n-k+1)$. Gauname:

$$k!C_n^k = A_n^k \text{ arba } C_n^k = \frac{A_n^k}{k!} = \frac{n(n-1)(n-2)...(n-k+1)}{k!} = \frac{n!}{k!(n-k)!}$$

Pavyzdžiui, jei turime 10 prekių, o lentynoje telpa 7 prekės, tai nepaisydami prekių išdėstymo tvarkos šias prekes galime išdėlioti lentynoje $C_{10}^7 = \frac{10!}{7!(10-7)!} = \frac{8 \cdot 9 \cdot 10}{3 \cdot 2 \cdot 1} = 1080$ būdų.

Poaibiai. Visus galimus *n* elementų aibės poaibius galime gauti generuodami iš eilės 0, 1, 2, ..., *n* ilgio derinius be pasikartojimų. Galimas ir dar paprastesnis būdas: pakanka sugeneruoti visus įmanomus žodžius, kurių ilgis *n* iš abėcėlės {true, false}.



21 pav Abėcėlės {true, false} žodžių transformavimo į poaibius pavyzdys

```
const MAXN = 20;          { didžiausia n reikšmė }

var parinktas : array [1..MAXN] of boolean;

procedure spausdink (m: integer);
var i : integer;
begin
    write('{ ');
    for i := 1 to m do
        if parinktas[i] then
            write(i, ' ');
    writeln('}');
end;
```

```
procedure generuok(k, n : integer);
{ nagrinėjamas k-asis n elementų aibės narys }
var log : boolean;
begin
    { jei k > n, tai ši procedūra iškviesta jau sugeneravus visą poaibį }
    if k > n then
        spausdink (k)
    else
        for log := false to true do begin
            parinktas[k] := log;
            generuok(k + 1, n);
        end;
end;
```

Norėdami gauti visus poaibius iš 4 elementų, į procedūrą `generuok` kreipiamės:

```
n := 4;
generuok(1, n);
```

Suskaičiuosime, kiek skirtingų poaibių turės aibė iš n elementų, o tuo pačiu ir algoritmo sudėtingumą. Poaibių skaičius lygus visų įmanomų n ilgio žodžių iš abėcėlės {true, false} skaičiui. Kadangi kiekvieną tokio žodžio raidę galime parinkti dviem būdais (atitinkamas elementas arba įtraukiamas į poaibį, arba ne), tai tokių žodžių (ir galimų poaibių) skaičius lygus 2^n.

5.4 Uždavinys *Pakyla*[20]

Panagrinėsime vieną uždavinį, kurio sprendimui reikia taikyti kombinatorikos žinias ir perrinkti visus įmanomus variantus.

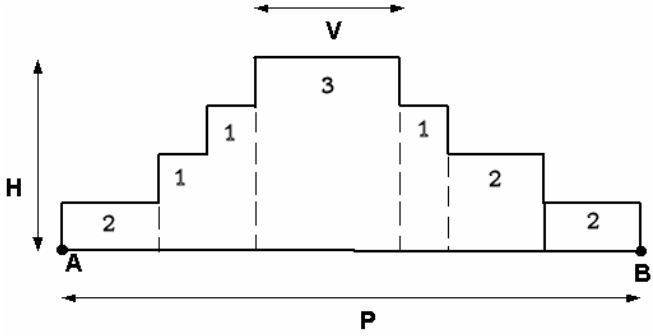
[20] Šis uždavinys buvo pateiktas Lietuvos moksleivių informatikos olimpiadoje III etape 2004 metais.

Tarp dviejų taškų A ir B norime pastatyti pakylą, kurios aukštis H metrų. Į pakylos viršų tiek iš taško A, tiek iš taško B turi vesti kylantys laiptai. Laiptų pakopos aukštis yra 1 metras. Nesunku apskaičiuoti, kad pakylą turi sudaryti (2H-1) pakopų – po (H – 1) iš kiekvienos pusės bei viršutinė. Pirmoji laiptų, kylančių iš taško A (taško B), pakopa turi prasidėti taške A (atitinkamai taške B).

Atstumas tarp taškų A ir B lygus P metrų. O kiekvienos pakopos plotis turi būti lygus sveikajam metrų skaičiui. Aukščiausioje dalyje esančios pakopos plotis turi būti lygus V metrų.

Užduotis. *Reikia rasti visus galimus skirtingus būdus pakylai įrengti. Dvi pakylos laikomos skirtingomis, jei jų aukštis skiriasi bent vienoje pozicijoje tarp taškų A ir B.*

Galioja ribojimai: pradiniai duomenys tokie, kad galimų variantų skaičius pakylai įrengti neviršija 20 000.



22 pav. Pakylos pavyzdys. H=4; P=12; V=3;
Ši pakyla apibūdinama seka: 0 2 3 4 7 8 10 12

Prieš sprendžiant uždavinį, svarbu tiksliai apibrėžti, ko iš tiesų ieškome. Kiekvieną galimą pakylą atitinka didėjanti skaičių nuo 0 iki P seka $\{S_i\}$, kurią sudaro lygiai $2H$ skaičių ir kuri tenkina papildomus ribojimus:

- $S_1 = 0$;
- $S_{2H} = P$;
- $S_{H+1} - S_H = V$.

Kiekvienas šios sekos elementas rodo vietą (koordinatę x), kurioje keičiasi pakopos aukštis. Pavyzdžiui, paveiksle pavaizduotą pakylą atitinka skaičių seka $< 0, 2, 3, 4, 7, 8, 10, 12 >$.

Taigi pirmojo ir paskutinio nario reikšmės yra fiksuotos, o $(H+1)$-ojo nario reikšmė priklauso nuo H-ojo nario: $S_{H+1} = S_H + V$. Nesunku apriboti k-ojo nario reikšmę:

$S_{k-1} < S_k \le P - (V - 1) - (2H - k)$, jei $2 \le k \le H$;
$S_{k-1} < S_k \le P - (2H - k)$, jei $H+2 \le k \le 2H - 1$.

Apatinis ribojimas išplaukia iš to, kad seka yra didėjanti, o viršutinis – kad nepritrūktų skaičių sekai užbaigti.

Gavome derinių generavimo uždavinį, tik tam tikrais ribojimais maksimalioms pozicijų reikšmėms.

Pasinaudosime jau žinomu derinių generavimo algoritmu, kurį pritaikysime šio uždavinio sprendimui. Beje, sutarsime, kad sprendinys egzistuoja.

```
const MAXH = 100; { maksimalus pakylos aukštis }

var s : array [1..2*MAXH] of integer;
    P, H, V : integer;
```

```
procedure generuok(k : integer);
{ generuoja sekos narį, kurio numeris k }
var i, max : integer;
begin
    if k = 2*H then
        { sugeneruoti visi nariai (paskutinis žinomas iš anksto) }
        spausdink(2*H)[21]
    else if k = H+1 then begin
        { (H+1)-osios pakopos viršūnės plotis fiksuotas }
        s[k] := s[k-1]+V;
        generuok(k+1);
    end
    else begin
        { nagrinėjamos visos galimos k-ojo nario reikšmės }
        if k <= H then
            max := P-(2*H-k)-(V-1)
        else
            max := P-(2*H-k);
        for i := s[k-1]+1 to max do begin
            s[k] := i;
            generuok(k+1);
        end;
    end;
end;
```

Į procedūrą `generuok` turi būti kreipiamasi tokiu būdu:

```
S[1] := 0;
S[2*H] := P;
generuok(2);
```

5.5 Perrinkimo optimizavimas

Panagrinėkime dar vieną pavyzdį. Sakykime, saugos kodą, kurį reikia surinkti įeinant į laiptinę, sudaro 3 skaitmenys. Norint jį atspėti, reikia išbandyti $10^3=1000$ variantų. Jei vieną kodą galima

[21] Procedūra `spausdink` analogiška spausdinimo procedūrai 5.1 skyrelyje.

surinkti ir pabandyti atidaryti duris per 3 sekundes, tai visus variantus pavyks išbandyti per 50 minučių. Tačiau jei saugos kodą sudarytų 4 skaitmenys, tai visiems $10^4 = 10\,000$ variantams išbandyti prireiktų daugiau nei 8 valandų. Matome, kad pradiniams duomenims (t. y. skaitmenų skaičiui) padidėjus 33%, galimų sprendinių skaičius padidėja 900%. Toks staigus sprendinių skaičiaus augimas vadinamas **kombinatoriniu sprogimu**.

Vienas didžiausių perrinkimo trūkumų yra tai, kad susiduriama su kombinatoriniu sprogimu. Generuojant kombinatorinius objektus kitaip ir negali būti: reikia rasti visus objektus, o jų yra daug, taigi ir algoritmų sudėtingumas turi būti didelis. Tačiau dažniau tenka ieškoti tam tikros kombinacijos, t. y. sprendinio, tenkinančio konkrečias sąlygas.

Todėl daugelyje tokių uždavinių stengiamasi **optimizuoti paiešką**. Vienas galimų optimizavimo būdų – paanalizuoti sprendinio struktūrą ir **sumažinti galimų sprendinių paieškos erdvę**. Taip darėme *Aštuonių valdovių uždavinyje*. Pradinė sprendinių erdvė buvo gana didelė: buvo sutarta, kad kiekviena valdovė gali stovėti bet kuriame lentos langelyje (po vieną valdovę langelyje), ir galimų variantų skaičius viršijo $4 \cdot 10^9$. Tačiau jei perrenkant variantus, kiekviena valdovė statoma tik į tuščią eilutę – tai išnagrinėjamų variantų skaičius iš karto sumažėja iki $8! = 40\,320$.

Gali pavykti sumažinti ir skyrelio pradžioje pateikto uždavinio paieškos erdvę. Jei žinoma, kad visi skaičiai turi būti paspausti vienu metu, tai saugos kode nebus pasikartojančių skaitmenų. Be to, šitaip parenkant kodą nenustatoma skaitmenų tvarka, todėl galime dar sumažinti sprendinių erdvę: pakanka išbandyti visus derinius. Pavyzdžiui, bandant atspėti keturių skaitmenų saugos kodą, mus domina visi deriniai iš 10 po 4. Jų skaičius yra $C_{10}^4 = 210$ (palyginkite su 10 000).

Jei reikalingas tik vienas sprendinys, paiešką verta optimizuoti parenkant sprendinių nagrinėjimo tvarką taip, kad tikėtini sprendiniai būtų nagrinėjami pirmiausia, jei tik tai įmanoma padaryti.

Yra įvairiausių kitų metodų paieškai pagreitinti, dažnai priimtinų tik konkrečiam uždaviniui Pavyzdžiui, ieškant geriausio ėjimo stalo žaidimuose, naudojama *Minimax paieška su Alfa-Beta atkirtimu;* šis metodas leidžia anksčiau atkirsti daug neperspektyvių paieškos medžio šakų.

6 RIKIAVIMAS IR PAIEŠKA

Any inaccuracies in this index may be explained by the fact that it has been sorted with the help of a computer.
Bet kokie netikslumai šiame sąraše gali būti paaiškinti tuo, kad jis buvo išrikiuotas kompiuteriu.
Donaldas Knutas (Donald Knuth)

Su rikiavimo ir paieškos uždaviniais susiduriama labai dažnai. Rikiavimas ir paieška neretai yra sudėtinė kitų algoritmų dalis. Norint sėkmingai dalyvauti informatikos olimpiadose, būtina išmanyti rikiavimo ir paieškos algoritmus. Su jais susipažinsime šiame skyrelyje.

6.1 Rikiavimo uždavinys

Rikiavimo uždavinys apibrėžiamas taip: reikia rasti tokią sekos $a_1, a_2, ..., a_n$ perstatą $a_{j1}, a_{j2}, ..., a_{jn}$, kad $a_{j1} \le a_{j2} \le ... \le a_{jn}$.

Praktikoje retai rikiuojama vien tik skaičių seka. Dažniausiai dirbama su sudėtingesniais duomenimis (įrašais), kurie rikiuojami pagal vieną ar kelis įrašo laukus. Pastaruoju atveju norint palyginti du įrašus tenka apibrėžti, kada vienas įrašas „mažesnis“ už kitą, ir parašyti atskirą loginę funkciją dviems elementams palyginti.

Kad nekomplikuotume, algoritmus pateiksime rikiuodami skaičių masyvą didėjimo (nemažėjimo) tvarka:

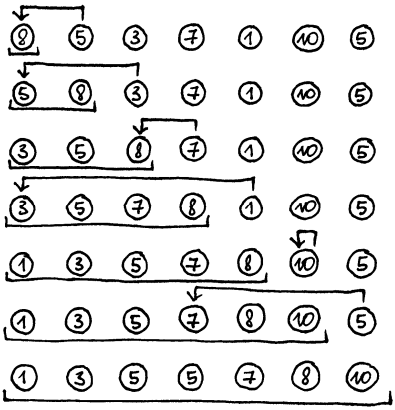
```
const MAXN = ...;     { maksimalus masyvo ilgis }
type masyvas = array [1..MAXN] of integer;
```

Rikiavimo algoritmų yra daug ir įvairių, tolesniuose skyreliuose aptarsime tik kelis naudingiausius. Pateiktus algoritmus bus nesunku pritaikyti ir kitokiems duomenų tipams.

6.2 Rikiavimas įterpimu

Rikiavimo įterpimu (angl. *Insertion sort*) algoritmo idėja primena rankoje laikomų kortų rikiavimą – į išrikiuotų kortų eilę kiekvienu žingsniu įterpiama viena nauja korta.

Pradedant rikiuoti masyvą, surikiuota masyvo dalis susideda iš vieno (pirmojo) elemento. Prieš atliekant k-ąjį žingsnį, jau yra išrikiuota masyvo dalis $[1..k]$, o k-uoju žingsniu $(k + 1)$-asis elementas įterpiamas į šią išrikiuotą dalį. Įterpimas atliekamas tokiu būdu: $(k + 1)$-asis elementas įsimenamas, visi didesni už jį išrikiuotos masyvo dalies elementai paslenkami pirmyn, o šis įterpiamas į naują savo vietą.



23 pav. Rikiavimo įterpimu pavyzdys

```
procedure rikiuok(const n : integer;
                  var A : masyvas);
var i, k, t : integer;
begin
    for k := 1 to n - 1 do begin
        t := A[k + 1];
        { skaičių t įterpsime į išrikiuotą masyvo dalį [1..k] }
        i := k;
        while (i > 0) and (A[i] > t) do begin
            A[i + 1] := A[i];
            i := i - 1;
        end;
        A[i + 1] := t;
     end;
end;
```

Algoritmo sudėtingumas blogiausiu atveju yra $O(n^2)$. Tuo nesunku įsitikinti panagrinėjus algoritmo veikimą rikiuojant seką, kuri jau išrikiuota **priešinga** tvarka – tuomet kiekvienu žingsniu elementas įterpiamas į masyvo pradžią. Taigi atliekamų veiksmų skaičius priklauso nuo pradinės masyvo tvarkos. Kuo tvarkingesnis (panašesnis į išrikiuotą) yra masyvas, tuo greičiau veikia rikiavimas įterpimu. Jei tenka rikiuoti beveik išrikiuotą masyvą, algoritmas veikia beveik tiesiškai.

Algoritmas nėra tinkamas rikiuoti didelių elementų masyvams, kadangi atliekama itin daug kopijavimo operacijų. Tačiau rikiavimą įterpimu efektyvu taikyti sąrašų (sudėtingesnių duomenų struktūrų) rikiavimui – juose elemento įterpimą galima atlikti nekopijuojant kitų elementų.

Taigi rikiavimą įterpimu verta naudoti, jei masyvas nedidelis, jame saugomi nedideli elementai arba iš anksto žinoma, kad teks kelis kartus rikiuoti tą patį masyvą, pavyzdžiui, pakeitus kelis jo elementus.

6.3 Greitasis rikiavimas

Greitojo rikiavimo algoritmas (angl. *Quicksort*) perskiria rikiuojamą masyvą į dvi dalis, ir kiekvieną dalį išrikiuoja atskirai. Pagalvokime, kokias sąlygas turi tenkinti masyvas, kad perskyrę jį pusiau ir šias dalis išrikiavę atskirai, gautume išrikiuotą masyvą. Atsakymas gana paprastas: pirmojoje dalyje turi būti mažesnieji elementai, o antroje – didesnieji, t.y. pirmoje dalyje neturi būti jokio elemento, kuris, išrikiavus masyvą, atsidurtų antroje dalyje ir atvirkščiai.

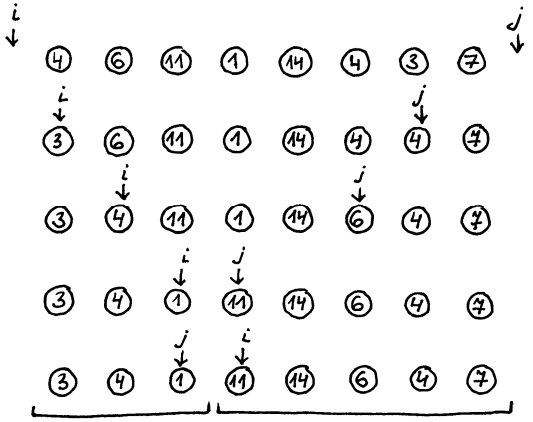
Deja, nežinomas joks greitas (tiesinis) „perkėlimo“ algoritmas. Tačiau nenusiminkime. Yra žinomi tiesinio sudėtingumo

algoritmai, kurie, perkeldami mažesniuosius elementus į pirmą dalies pusę, padalija masyvą *beveik* pusiau. T. y. tikimybė, kad padalijimas bus neblogas (abiejose pusėse elementų skaičius bus panašus), yra labai didelė.

Pateiksime funkciją *perskirk*, perskiriančią masyvo dalį [*k*..*d*] į dvi dalis [*k*..*v*] ir [*v*+1..*d*] taip, kad pirmojoje dalyje atsidurtų mažesnieji elementai, o antroje – didesnieji. Kadangi funkcija ne visuomet masyvo dalį perskiria pusiau, ji grąžina dalijamojo elemento indeksą *v* (t. y. vietą, kurioje masyvo dalis perskiriama). Šios informacijos reikia rikiavimo algoritmui.

```
function perskirk(var A : masyvas;
                  const k, d : integer) : integer;

    procedure sukeisk(var x, y : integer);
    var t : integer;
    begin
        t := x;
        x := y;
        y := t;
    end;

var x : integer; { dalijamoji reikšmė }
    i, j : integer;
begin
    x := A[k];
    i := k - 1;
    j := d + 1;
    perskirk := 0;
    while perskirk = 0 do begin { dalis dar neperskirta }
        repeat { praleidžiami elementai, mažesni už x }
            i := i + 1
        until A[i] >= x;
        repeat { praleidžiami elementai, didesni už x }
            j := j - 1
        until A[j] <= x;
        if i < j then sukeisk(A[i], A[j])
        else perskirk := j;
    end;
end;
```

Šis perskyrimo algoritmas pirmiausia pasirenka dalijamąją reikšmę x ir pamažu augina dvi masyvo dalis: $[k..i]$ su mažesniais už x elementais ir $[j..d]$ su elementais, didesniais už x. Kai indeksai i ir j „susitinka“, algoritmas baigia darbą, o funkcija grąžina perskyrimo vietą. Iš tiesų šioje funkcijoje slepiasi daug svarbių detalių ir ją programuoti reikia labai atidžiai.

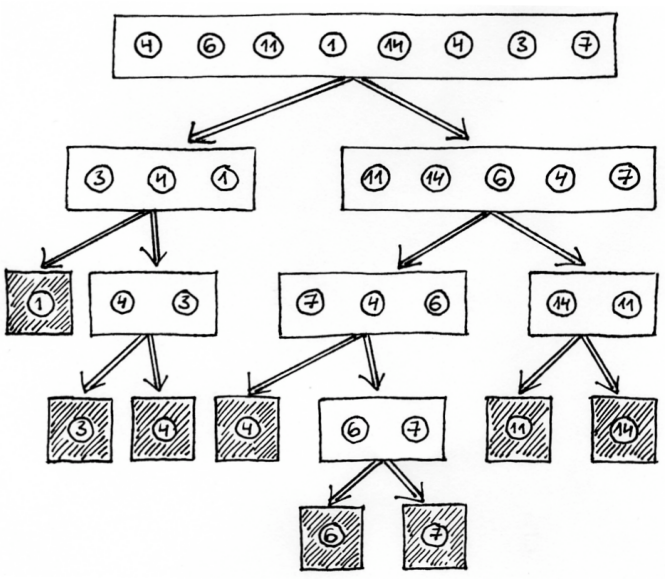


24 pav. Funkcijos `perskirk` *veikimo pavyzdys*

Dabar nesunku užrašyti greitojo rikiavimo algoritmą:

```
procedure rikiuok(var A : masyvas;
                  const k, d : integer);
var v : integer;
begin
    if k < d then begin
        v := perskirk(A, k, d);
        { rekursyviai išrikiuojamos kairioji ir dešinioji masyvo dalys }
        rikiuok(A, k, v);
        rikiuok(A, v + 1, d);
    end;
end;
```

Norint surikiuoti n elementų seką A, į procedūrą kreipiamasi
`rikiuok (A, 1, n);`



25 pav. Greitojo rikiavimo veikimo iliustracija

Nelengva apskaičiuoti greitojo rikiavimo algoritmo sudėtingumą, nes atliekamų veiksmų skaičius priklauso ne tik nuo duomenų skaičiaus, bet ir nuo pačių duomenų. Greitojo rikiavimo algoritmo sudėtingumas blogiausiu atveju yra $O(n^2)$, o vidutiniu – $O(n \log n)$.

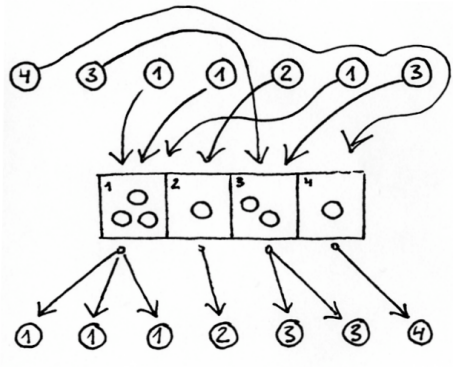
Nors yra rikiavimo algoritmų, net blogiausiu atveju išrikiuojančių n elementų per $O(n \log n)$ laiką, greitasis rikiavimas, nepaisant savo blogiausio atvejo sudėtingumo, praktiškai yra sparčiausias rikiavimo algoritmas. Be to, jį užrašyti procedūra nesudėtinga, o jo vykdymui nereikalinga papildoma atmintis.

Dėl išvardytų privalumų greitasis rikiavimas dažnai naudojamas praktikoje.

Ir įterpimo, ir greitojo rikiavimo algoritmai pagrįsti dviejų elementų palyginimais, t. y. šių algoritmų sudėtingumas proporcingas atliekamų palyginimų skaičiui. Yra įrodyta, kad nepavyks parašyti palyginimais paremto algoritmo, kurio efektyvumas būtų geresnis nei $O(n \log n)$, kur n – rikiuojamos sekos elementų skaičius. Tačiau duomenims, pasižymintiems tam tikromis savybėmis, galima sudaryti greitesnių rikiavimo algoritmų. Vienas tokių – rikiavimas skaičiavimu.

6.4 Rikiavimas skaičiavimu

Rikiavimas skaičiavimu (angl. *Counting sort*) skirtas rikiuoti sekoms, kurių visi elementai priklauso nedidelei aibei.



26 pav. Rikiavimas skaičiavimu

Pavyzdžiui, žinome, kad visi masyvo A elementai yra sveikieji skaičiai, priklausantys intervalui [1, 1000]. Tuomet atskirame 1000 elementų skaičių masyve C įsimenama, kiek kartų kiekviena reikšmė pasirodo pradiniame masyve A. Belieka pasinaudoti šia informacija ir elementus surašyti atgal į masyvą A didėjimo tvarka. Šio algoritmo sudėtingu-

mas yra O(n) (tiesinis), o jam reikalinga papildoma atmintis priklauso nuo aibės, kuriai priklauso rikiuojamo masyvo elementai, dydžio.

```
const MAXN = ...;    { maksimalus masyvo ilgis }
type skaičius = 1..1000;
     masyvas = array [1..MAXN] of skaičius;
     intMasyvas = array [skaičius] of integer;

procedure rikiuok(const n : integer;
                  var A : masyvas);
var c : intMasyvas;
    i, j : longint;
begin
    { suskaičiuojama, kiek kokių elementų yra masyve A }
    for i := low(C) to high(C) do
        C[i] := 0;
    for i := 1 to n do
        C[A[i]] := C[A[i]] + 1;
    { visi n masyvo A elementų surašomi iš eilės }
    j := low(C);
    for i := 1 to n do begin
        while C[j] = 0 do
            j := j + 1;
        C[j] := C[j] - 1;
        A[i] := j;
    end;
end;
```

6.5 Paieškos uždavinys

Paieškos uždavinys apibrėžiamas taip: duota seka a_1, a_2, ..., a_n ir elementas x. Reikia nustatyti, ar x yra šioje sekoje, o jei yra, tai koks jo numeris. Kitaip sakant, reikia rasti tokį sekos nario indeksą j, kad būtų $a_j = x$, arba nustatyti, kad x nėra lygus nė vienam iš sekos narių.

Praktikoje sekos nariai yra sudėtingi duomenų tipai (įrašai), o paieška atliekama pagal vieną arba kelis įrašo laukus, vadinamus

paieškos raktu. Paprastumo dėlei paiešką atliksime tik skaičių sekoje, kurią vaizduosime vienmačiu masyvu.

6.6 Tiesinė paieška

Paprasčiausias paieškos algoritmas – iš eilės patikrinti visus masyvo elementus – vadinamas **tiesine paieška** (angl. *Linear search*). Patikrinimą, ar n ilgio masyve A yra elementas x, atlieka tokia funkcija:

```
function ieškok (const n, x: integer;
                 var A: masyvas): integer;

var j: integer;
begin
    j := 1;
    while (A[j] <> x) and (j < n) do
        j := j + 1;
    if A[j] = x then
        ieškok := j
    else
        ieškok := 0; { elementas nerastas }
end;
```

Baigus vykdyti tiesinę paiešką, funkcijos reikšmė bus lygi ieškomo elemento indeksui masyve A arba nuliui, jei tokio elemento masyve nėra. Žinoma, priklausomai nuo masyvo rėžių gali tekti kitaip pažymėti nesėkmingą paieškos baigtį.

Tiesinės paieškos sudėtingumas, kaip teigia ir pats pavadinimas, yra $O(n)$. Netgi žinant, kad ieškomasis elementas tikrai yra masyve, vidutiniškai teks atlikti $n / 2$ patikrinimų (jei bet koks elementų išsidėstymas masyve vienodai tikėtinas). Taigi atliekamų veiksmų skaičius tiesiškai priklauso nuo masyvo ilgio n.

Svarbiausias šio algoritmo privalumas – paprastumas.

6.7 Dvejetainė paieška

Daug efektyviau galima atlikti paiešką išrikiuotame masyve – prisiminkime, kaip greitai randame norimą telefono numerį storoje telefonų knygoje.

Dvejetainės paieškos (angl. *Binary search*) principas labai paprastas: ieškomasis elementas palyginamas su surikiuotos sekos viduriniu nariu. Jei jie yra lygūs, vadinasi, radome ieškomą elementą sekoje. Jei ieškomasis elementas yra mažesnis už vidurinį, tai juo labiau jis mažesnis ir už visus „dešiniuosius“ sekos narius, todėl paiešką tęsime kairiojoje sekos dalyje. Analogiškai, jei ieškomasis elementas didesnis už vidurinį, paiešką tęsime dešiniojoje masyvo dalyje. Toliau ieškoma tuo pačiu principu, kol randamas ieškomas elementas arba paieškos sritis tampa tuščia.

Aprašytąjį algoritmą nesudėtinga užrašyti rekursyvia funkcija. Nesėkmingos paieškos atveju ši funkcija grąžins nulį, o sėkmingos – ieškomo elemento indeksą masyve.

```
function ieškok(x, k, d : integer;
                var A : masyvas) : integer;

var v : integer;
begin
    if k > d then
        ieškok := 0
    else begin
        v := (k + d) div 2;
        { pagal vidurinį masyvo dalies elementą toliau ieškoma
          kairiojoje arba dešiniojoje masyvo dalyje }
        if A[v] > x then
            ieškok := ieškok(x, k, v - 1, A)
        else if A[v] < x then
            ieškok := ieškok(x, v + 1, d, A)
        else { trečiuoju atveju A[v] = x (elementas rastas) }
            ieškok := v;
    end;
end;
```

Taigi jei norime sužinoti, ar skaičius x yra n elementų masyve A, turime patikrinti sąlygą `ieškok(A, x, 1, n) > 0`.

Dvejetainės paieškos algoritmas kiekvienu žingsniu sutrumpina paieškos sritį maždaug dvigubai. Kitaip tariant, jei masyvo ilgis padidėja dvigubai, tai algoritmui tenka atlikti tik *vieną papildomą žingsnį*. Dvejetainės paieškos sudėtingumas yra $O(\log n)$, t. y. logaritminis. Milijardo elementų dydžio masyve paieškai prireiktų ne daugiau kaip 30 žingsnių. Tačiau sąlygą, kad masyvas turi būti išrikiuotas, ne visuomet paprasta patenkinti.

Dvejetainės paieškos idėją galima panaudoti ne tik elemento paieškai išrikiuotame masyve. Geras pavyzdys – žaidimas *Atspėk skaičių*: pirmasis žaidėjas sugalvoja skaičių nuo 1 iki n, o antrasis bando jį atspėti; po kiekvieno spėjimo pirmasis žaidėjas pasako, ar jo sugalvotasis skaičius yra mažesnis, didesnis ar lygus spėtajam; žaidimo tikslas – atspėti skaičių kuo mažesniu bandymų skaičiumi. Vėliau žaidėjai apsikeičia vaidmenimis. Iš tiesų dvejetainė paieška – optimali spėjimo strategija. Nepaisant to, gali laimėti žaidėjas, kuriam tądien labiau sekasi.

Bendriausiu atveju dvejetainę paiešką galima pritaikyti sprendžiant lygtį $f(x) = y$ tam tikrame intervale, kur $f(x)$ – **monotoninė** (nedidėjanti arba nemažėjanti) **funkcija**.

6.8 Kada rikiuoti?

Jei programoje laikome masyvą, kuriame teks ieškoti elementų, reikia atsakyti į klausimą: ar nerikiuoti masyvo ir atlikti tiesinę paiešką, ar išrikiuoti masyvą ir ieškoti jame naudojant daug efektyvesnę dvejetainę paiešką.

Olimpiadose programos paprastumas – didelė vertybė. Todėl visuomet geriau naudoti kuo paprastesnius algoritmus, jei tik programos veikimo laikas yra pakankamas.

Tarkime, masyvą sudaro n elementų, o jame žadame ieškoti m kartų. Naudodami tiesinę paiešką nerikiuotame masyve, užtruksime $O(mn)$ laiko. Masyvo rikiavimas ir m kartų atlikta dvejetainė paieška užtruktų $O(n\log n + m\log n)$. Taigi, šiuo atveju rikiuoti masyvą verta tik tada, kai $m > \log n$.

6.9 Rikiavimo uždaviniai olimpiadose, uždavinys *Sekos rikiavimas*

Olimpiadose tiesioginių rikiavimo ar paieškos uždavinių pasitaiko retai. Daug dažniau rikiavimas ir paieška tėra kito, sudėtingesnio, algoritmo dalis[22].

Tuo tarpu uždaviniams, kuriuose tiesiogiai minimas rikiavimas, dažniausiai reikia sugalvoti kokią nors kitą originalią idėją, o ne taikyti žinomus rikiavimo ar paieškos algoritmus.

Kaip pavyzdį panagrinėkime pasaulinės informatikų olimpiados uždavinį *Sekos rikiavimas*[23].

> *Duota skaičių seka, kurios nariai gali įgyti tik tris skirtingas reikšmes: vienetą, dvejetą ir trejetą. Seką reikia surikiuoti*

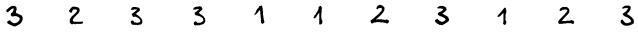
[22] Programavimo kalbų C ir C++ standartinėse bibliotekose yra realizuoti svarbiausi paieškos ir rikiavimo algoritmai, tad juos galima taikyti neprogramuojant šių algoritmų.

[23] Šis uždavinys buvo pateiktas 1996 metais Vengrijoje vykusioje Pasaulinėje informatikos olimpiadoje. Čia pateikėme sutrumpintą sąlygą.

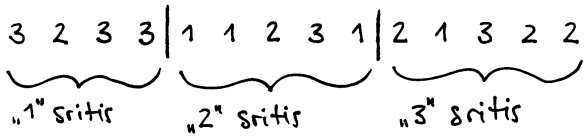
nemažėjimo tvarka. Rikiuojama sukeičiant vietomis po du sekos narius.

Užduotis. *Reikia rasti minimalų sukeitimo operacijų, reikalingų sekai surikiuoti, skaičių.*

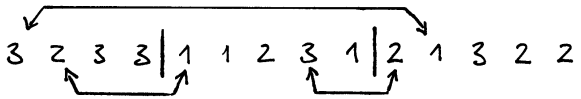
Toliau pateikti piešiniai iliustruoja rikiavimo algoritmą rikiuojantį seką minimaliu sukeitimų skaičiumi.



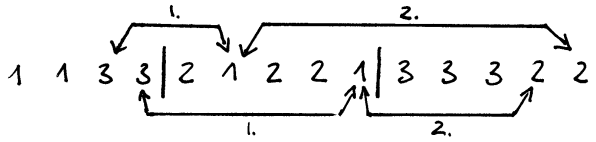
27 pav. Uždavinio „Sekos rikiavimas“ sprendimo iliustracija; paveiksle pateikta seka, kurią reikia išrikiuoti



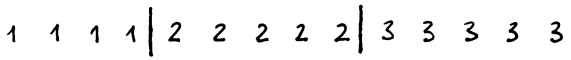
28 pav. 1 žingsnis: suskaičiuojama, kiek sekoje yra vienetų, dvejetų ir trejetų (šiuo atveju 4 vienetai, 5 dvejetai ir 5 trejetai), ir seka padalijama į vienetų, dvejetų ir trejetų sritis



28 pav. 2 žingsnis: randamos visos poros, kurių narius sukeitus vietomis, ***abu*** *atsidurs savo srityse, ir atliekami sukeitimai*



28 pav. 3 žingsnis: ne savo srityse likę skaičiai sukeitinėjami po tris; kiekvienam trejetui sutvarkyti prireiks dviejų sukeitimų



29 pav. Gavome surikiuotą seką: buvo atlikti 7 sukeitimai, sukeitimų skaičius yra minimalus

7 PIRMA PAŽINTIS SU GRAFAIS: PAIEŠKA PLATYN IR GILYN

Mathematicians are like Frenchmen: whenever you say something to them, they translate it into their own language, and at once it is something entirely different.
Matematikai – kaip prancūzai: kad ir ką jiems bepasakytum, jie iškart išverčia tai į savo kalbą, ir tai iškart tampa visiškai skirtingu dalyku.
Johanas Volfgangas fon Gėtė (Johann Wolfgang von Goethe)

Per Karaliaučių tekančioje Priegliaus upėje yra dvi gražios salos. Septyni tiltai jungia krantus su šiomis salomis, taip pat abi salas tarpusavyje. Maždaug prieš tris šimtus metų Karaliaučiaus gyventojus sudomino klausimas: ar galimas toks maršrutas, kuris prasidėtų viename iš krantų, kad per kiekvieną tiltą būtų pereinama tik vieną kartą, o pasivaikščiojimas pagaliau baigtųsi ten, kur ir prasidėjo.

30 pav. Karaliaučiaus žemėlapis Oilerio laikais

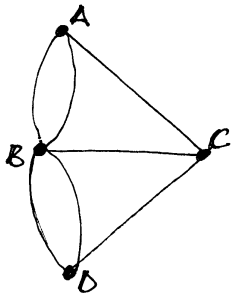
1736 m. šio uždavinio ėmėsi garsus šveicarų matematikas Leonardas Oileris (*Leonhard Euler*). Krantai, salos, tiltai – visa tai jam buvo tik uždavinio „apvalkalas“: salos buvo tam tikri objektai, o tiltai – šiuos objektus siejantys ryšiai. Krantus bei salas Oileris sutapatino su taškais, o tiltus – su linijomis. Jis paprastai įrodė, jog maršruto, tenkinančio minėtus kriterijus, nėra ir negali būti. Šitaip buvo pradėta grafų teorija. Tačiau apie tai vėliau.

Šiame skyrelyje išsiaiškinsime, kas tie grafai, kam jie reikalingi, taip pat susipažinsime su dviem pirmaisiais grafų teorijos algoritmais.

7.1 Grafo sąvoka

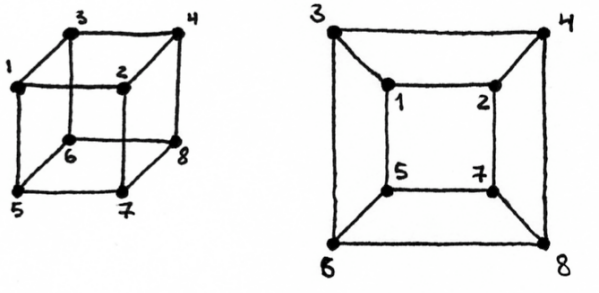
Grafas yra sąsajų struktūra, nurodanti, kad yra objektų grupė, ir kad šie objektai yra tarpusavy susiję (arba nesusiję).

Objektai vadinami grafo **viršūnėmis**, o jų ryšius apibūdina grafo **briaunos**. Jei grafe briauna jungia dvi viršūnes, tai šios viršūnės vadinamos **gretimomis**. Viršūnei gretimos viršūnės dar vadinamos jos **kaimynėmis**. Jei briauna jungia viršūnę su ja pačia, tai ta briauna vadinama **kilpa.**



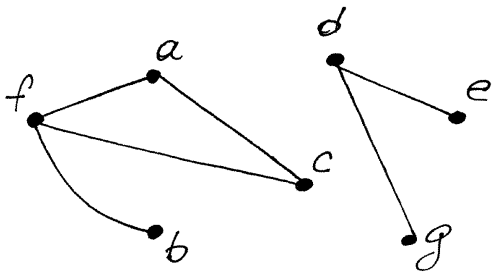
31 pav. Karaliaučiaus tiltų brėžinys, kai krantai sutapatinti su taškais, o tiltai – su linijomis

Labai svarbi yra grafų geometrinė interpretacija: grafo viršūnes patogu vaizduoti plokštumos taškais, o briaunas – linijomis, jungiančiomis viršūnių (taigi taškų) porą. Viršūnių (taškų) ir briaunų (linijų) geometrinis išdėstymas nesvarbus, svarbus tik pačių sąsajų pavaizdavimas. Tą patį grafą galima nupiešti daugybe skirtingų būdų.



32 pav. Tas pats grafas, pavaizduotas dviem skirtingais būdais, grafo viršūnės žymimos skaičiais

Matematiškai grafas apibrėžiamas kaip dviejų aibių – viršūnių ir briaunų – rinkinys: $G = (V, B)$. Briaunų aibės elementai – tai viršūnių poros. Pavyzdžiui, 33 pav. pavaizduotą grafą atitinka tokios viršūnių ir briaunų aibės: $V = \{ a, b, c, d, e, f, g \}$, $B = \{ (a, c), (a, f), (b, f), (c, f), (d, e), (d, g) \}$.



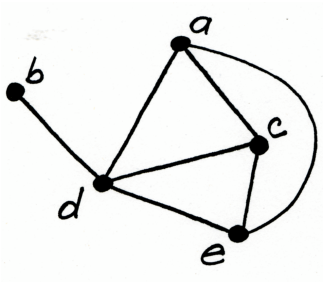
33 pav. Grafas, jo viršūnės žymimos raidėmis

Pastabesni galėtų prikibti – aibėje negali būti pasikartojančių elementų. Tačiau skyrelio pradžioje sutikome grafą, kurio dvi

viršūnes jungia **kelios briaunos** (salą su tuo pačiu krantu jungia keli tiltai).

Grafai su pasikartojančiomis briaunomis vadinami **multigrafais**. Tokių grafų matematiniame modelyje aibė pakeičiama multiaibe (aibe, kurioje elementai gali kartotis). Daugelyje uždavinių vietoj multigrafo pakanka nagrinėti paprastą grafą, gautą iš multigrafo, iš kelių dvi viršūnes jungiančių briaunų paliekant tik vieną, tinkamiausią uždaviniui spręsti.

Keliu grafe vadinama gretimų viršūnių seka, kai ta pati briauna kelyje sutinkama tik vieną kartą, o ta pati viršūnė gali būti kelyje sutinkama kelis kartus. Jei kelias prasideda ir baigiasi toje pačioje viršūnėje, jis vadinamas **ciklu**. Kelio ilgis lygus pereitų briaunų skaičiui.



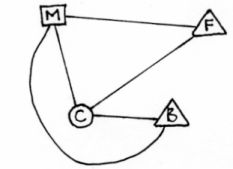
34 pav. Viršūnių seka a—d—c—e—d—b yra kelias, kurio ilgis 5, o seka a—c—d—e—a yra ciklas (ilgis 4)

Tačiau kam gi reikalinga grafų teorija? Pasirodo, grafų teorijos „kalba“ galima išreikšti daugelį svarbių (dažnai praktinių) uždavinių. Vieną jų ką tik matėme – tai maršruto, kai kiekviena briauna pereinama lygiai vieną kartą, paieška. Grafu galima pavaizduoti ir miesto planą, briaunomis žymint jo gatves. Tuomet, kiekvienai briaunai priskyrę po teigiamą skaičių – gatvės ilgį, galime klausti: koks trumpiausias kelias iš viršūnės *a* į viršūnę *b*? Šio uždavinio sprendimui taip pat yra efektyvus algoritmas, kurį pateiksime vėliau.

Ne visuomet ryšys tarp uždavinio ir jo sumodeliavimo grafų teorijos terminais toks akivaizdus. Štai dar vieno uždavinio pavyzdys: tarkime, kad *n* vaikų pasirinko mokytis kai kuriuos iš *m* dalykų.

Reikia sudaryti optimalų (kuo glaustesnį) užsiėmimų tvarkaraštį. Sudarykime grafą, kurio m viršūnių atitiks visus dalykus, o briauna, jungianti viršūnes a ir b, reikš, kad bent vienas vaikas pasirinko abu dalykus a ir b.

MATEMATIKA	FIZIKA	CHEMIJA	BIOLOGIJA
DAIVA JULIUS TOMAS	JULIUS TOMAS	ASTA DAIVA TOMAS	ASTA



35 pav. Viršūnių spalvinimo uždavinys. Sudarę ir nuspalvinę pasirinkimų grafą, darome išvadą, kad fizikos ir biologijos užsiėmimai gali vykti vienu metu

Dabar galime spręsti *grafo viršūnių spalvinimo uždavinį*: kaip, panaudojant kuo mažiau spalvų, nuspalvinti grafo viršūnes, kad jokios dvi gretimos grafo viršūnės nebūtų nuspalvintos ta pačia spalva. Viena spalva nuspalvintomis viršūnėmis pažymėtų dalykų užsiėmimai gali vykti vienu metu: tai netrukdys nė vienam moksleiviui. Taigi svarbioji uždavinio dalis bus išspręsta. Deja, viršūnių spalvinimo uždaviniui nežinomas joks efektyvus algoritmas.

7.2 Grafų vaizdavimas

Grafus vaizduoti aibėmis pagal jų matematinį apibrėžimą dažniausiai nėra patogu. Tarus, kad grafas turi n viršūnių, sunumeruotų nuo 1 iki n, viršūnių aibės nurodyti nebūtina – pakanka žinoti viršūnių skaičių n. Grafo briaunas paprasčiausia pavaizduoti dvimačiu $n \times n$ loginiu masyvu: elementus $[u, v]$ ir $[v, u]$ pažymint reikšme `true`, jei

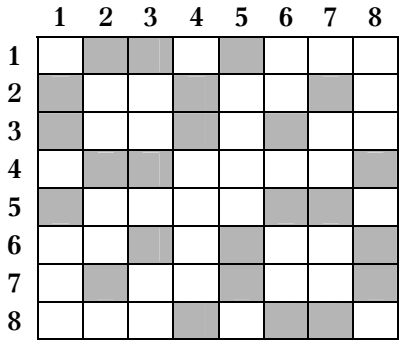
viršūnes su numeriais *u* ir *v* grafe jungia briauna. Šis masyvas visuomet[24] yra simetrinis įstrižainės atžvilgiu.

```
const MAXN = ...; { maksimalus grafo viršūnių skaičius }

type grafas = record
         n : integer;
         briauna : array [1..MAXN,
                          1..MAXN] of boolean;
     end;
```

Kol visos masyvo `briauna` reikšmės lygios `false`, grafe nėra nė vienos briaunos. Priklausomai nuo uždavinio pradinių duomenų, kai kurias viršūnes reikės sujungti briauna. Tai atlieka tokia procedūra:

```
procedure papildyk_briauna(var g : grafas;
                           u, v : integer);
begin
    g.briauna[u, v] := true;
    g.briauna[v, u] := true;
end;
```

Toks grafo vaizdavimas vadinamas **kaimynystės matrica**. Tokio vaizdavimo kompiuteryje privalumai – jo paprastumas ir galimybė sparčiai patikrinti, ar dvi viršūnes jungia briauna. Deja, yra ir svarbus trūkumas – norėdami rasti visas viršūnės *v* kaimynes, turime patikrinti visą *v*-ąją masyvo `briauna` eilutę, tikrindami sąlygą, ar `briauna[v, u] = true`. Jei



36 pav. Kaimynystės matrica pavaizduotas 32 pav. grafas; pilki langeliai žymi grafo briaunas

[24] 9 skyriuje nagrinėsime orientuotus grafus, kurie vaizduojami nesimetriškais dvimačiais masyvais.

grafas yra **retas** (t. y. jame palyginti nedaug briaunų), tai atmintis, skiriama beveik tuščiam masyvui, neefektyviai išnaudojama.

Kai grafe briaunų daug (grafas **tankus**), tai šis paprastas vaizdavimo būdas labai patogus.

Iš anksto žinant, kad grafas bus retas, geriau naudoti kitą vaizdavimo būdą – **kaimynystės sąrašus** – t. y. kiekvienai viršūnei saugoti jai gretimų viršūnių (jos kaimynių) sąrašą.

Naudojant sudėtingesnes dinamines duomenų struktūras šiems sąrašams saugoti, galima sutaupyti atminties. Tačiau olimpiadose, jei tik įmanoma, geriau vengti dinaminių duomenų struktūrų – jas kur kas sudėtingiau teisingai realizuoti per trumpą laiką.

Savo pavyzdžiuose paprastumo dėlei kaimynių sąrašą saugosime masyvu. Kadangi iš anksto nežinome, kiek daugiausiai kaimynių gali turėti kiekviena viršūnė, tai šių masyvų ilgis bus toks, koks gali būti didžiausias viršūnių skaičius.

```
const MAXN = ...; { maksimalus grafo viršūnių skaičius }

type viršūnė = record
         k : integer;                  { kaimynių skaičius }
         ksąrašas : array [1..MAXN] of integer;
                                       { kaimynių sąrašas }
     end;

     grafas = record
         n : integer;                    { viršūnių skaičius }
         vir : array [1..MAXN] of viršūnė;
                                       { viršūnių sąrašas }
     end;
```

Kai grafe nėra briaunų, visų viršūnių kaimynių skaičiaus atributas lygus nuliui. Įterpti briauną (u, v) į šitaip vaizduojamą grafą reiškia

papildyti viršūnių *u* ir *v* kaimynių sąrašus. Tai atlieka tokia procedūra:

```
procedure papildyk_briauna(var g : grafas;
                           u, v : integer);
begin
    with g do begin
        inc(vir[u].k);
        vir[u].ksąrašas[vir[u].k] := v;
        if v <> u then begin { jei tai ne kilpa }
            inc(vir[v].k);
            vir[v].ksąrašas[vir[v].k] := u;
        end;
    end;
end;
```

Nors surasti vienos viršūnės kaimynes galime labai greitai, patikrinti, ar viršūnes *u* ir *v* grafe jungia briauna tapo sudėtingiau: tam reikia perbėgti vienos iš šių viršūnių kaimynių sąrašą, ieškant antrosios.

Kurį iš aptartų vaizdavimo būdų pasirinkti? Tai priklauso nuo sprendžiamo uždavinio. Daugelyje algoritmų tenka surasti duotosios viršūnės kaimynes, o rečiau – patikrinti, ar viršūnes jungia briauna. Kai reikalingas abiejų šių operacijų efektyvumas, tą patį grafą gali tekti vaizduoti dviem būdais.

viršūnės	*kaimynės*
1	2, 3, 5
2	1, 4, 7
3	1, 4, 5
4	2, 3, 8
5	1, 6, 7
6	3, 5, 8
7	2, 5, 8
8	4, 6, 7

37 pav. Taip atrodys 32 paveikslo grafas, jį pavaizdavus kaimynystės sąrašais

Galimas ir dar kitoks grafo pavaizdavimo būdas. Jei grafe viršūnių labai daug, o briaunų nedaug, galime saugoti briaunų (viršūnių porų) sąrašą. Tuomet briauną, jungiančią viršūnes *u* ir *v*, verta vaizduoti dviem poromis: (*u*, *v*) ir (*v*, *u*). Išrikiavę tokį sąrašą, konkrečios briaunos paiešką galime atlikti per

O(log *b*) laiko (*b* – briaunų skaičius), pasitelkę dvejetainę paiešką. Praktikoje šis būdas retai naudojamas.

7.3 Paieška gilyn

Karalaitė slapta padavė Tesėjui kamuoliuką siūlų ir pamokė, ką reikia daryti, kad nepaklystų vingiuotuose paslaptingojo statinio koridoriuose. Tesėjas pririšo siūlo galą prie labirinto angos ir, eidamas priekin, vyniojo rankoje laikomą kamuoliuką.

Iš graikų mitų

Pirmieji grafų algoritmai, su kuriais susipažinsime, – tai paieška grafe gilyn ir platyn. Pradėjus nuo kažkurios viršūnės, aplankomos visos kitos briaunomis pasiekiamos viršūnės. Dvi skirtingos viršūnių aplankymo strategijos – paieška gilyn ir platyn – dažnai yra kitų algoritmų sudėtinė dalis.

Pradėsime nuo **paieškos gilyn** (angl. *Depth-First Search, DFS*), jos principas panašus į grįžimo metodo. Algoritmo parametras yra pradžios viršūnė v_0, iš jos aplankomos kitos viršūnės: aplankius viršūnę v_0, aplankoma dar neaplankyta v_0 kaimynė v_1, tada ieškoma dar neaplankyta v_1 kaimynė v_2 ir taip toliau, kol pasiekiama viršūnė v_m, kuri nebeturi neaplankytų kaimynių. Tuomet grįžtama vieną žingsnį ir žiūrima, ar viršūnė v_{m-1} dar turi nors vieną neaplankytą kaimynę v_m`. Jei turi, – ieškoma gilyn, jei ne – grįžtama dar per vieną žingsnį ir t. t. Paiešką gilyn, kaip ir grįžimo metodu pagrįstus algoritmus, paprasta realizuoti naudojant rekursiją.

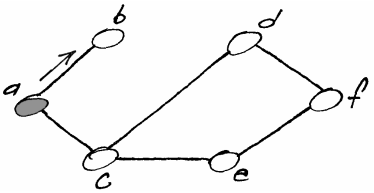
Skirtingai negu grįžimo metodas, paieška gilyn yra efektyvus algoritmas, kadangi kiekviena grafo viršūnė aplankoma tik vieną kartą. Tuo tarpu jei taikytume grįžimo metodą, ta pati viršūnė

galėtų būti aplankyta daug kartų, nes būtų išbandomi visi įmanomi keliai grafe, prasidedantys viršūnėje v_0.

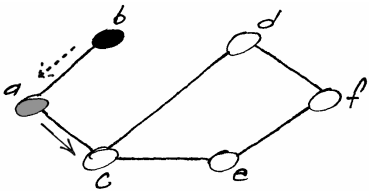
Paieškos gilyn algoritmas veikimo metu kiekvieną viršūnę nuspalvina tam tikra spalva – balta, pilka arba juoda. Viršūnių spalvoms žymėti aprašysime specialų duomenų tipą:

```
type spalvos = (balta, pilka, juoda);
```

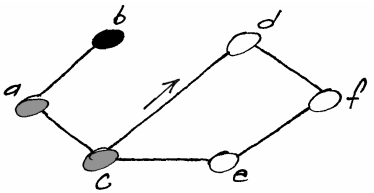
Prieš pradedant vykdyti algoritmą visos viršūnės nuspalvinamos baltai (pažymimos neaplankytomis). Algoritmo veikimo metu, aplankant viršūnę, ji nuspalvinama pilkai, o įvykdžius algoritmą su visomis neaplankytomis jos kaimynėmis – juodai.



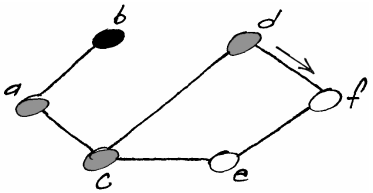
(1)



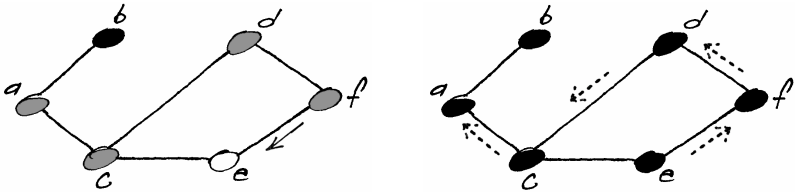
(2)



(3)



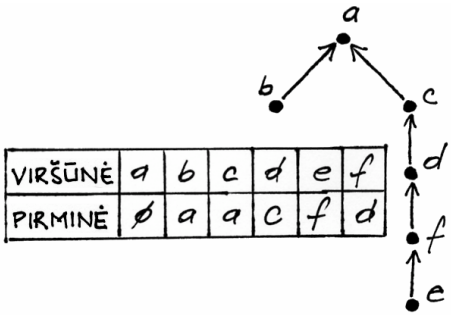
(4)



38 pav. Paieškos gilyn veikimo iliustracija, kai pradine viršūne pasirinkta viršūnė a

Algoritmas taip pat išsaugo paieškos į gylį pirminumo medį, t. y. kiekvienai viršūnei įsimena, iš kurios ši buvo aplankyta.

Žemiau pateiktas algoritmo tekstas Paskalio kalba. Algoritmo veikimo metu dažnai reikės rasti kurios nors viršūnės kaimynes, todėl grafą vaizduosime kaimynystės sąrašais.



39 pav. Paieškos gilyn pirminumo medis

```
type spalvos = (balta, pilka, juoda);
     sp_masyvas = array [1..MAXN] of spalvos;
     masyvas = array [1..MAXN] of integer;

var spalva : sp_masyvas;   { pradinės reikšmės – balta}
    pirminė : masyvas;     { pradinės reikšmės – 0}
```

```
procedure ieškok_gilyn(var[25] g: grafas;
                       v : integer { aplankoma viršūnė });
var u, i : integer;
begin
    spalva[v] := pilka;
    with g do
        { toliau paieška iš eilės vykdoma visose neaplankytose
          (baltose) kaimynėse }
        for i := 1 to vir[v].k do begin
            u := vir[v].ksąrašas[i];
            if spalva[u] = balta then begin
                pirminė[u] := v;
                ieškok_gilyn(g, u);
            end;
        end;
    spalva[v] := juoda;
end;
```

Iškvietus `ieškok_gilyn(`v_0`)`, visos viršūnės, kurias galima pasiekti briaunomis iš viršūnės v_0, bus pažymimos juodai. Atspausdinti kelią, kuriuo buvo pasiekta viršūnė *u*, nesunku pasinaudojus masyve `pirminė` išsaugota informacija:

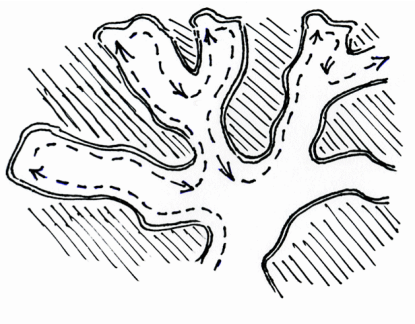
```
procedure spausdink_kelią(u : integer);
begin
    if pirminė[u] <> 0 then
        spausdink_kelią(pirminė[u]);
    writeln(u);
end;
```

Iš tiesų algoritme pakaktų viršūnes spalvinti tik dviem spalvomis: atskirti aplankytas nuo neaplankytų. Tačiau naudojant tris spalvas algoritmas tampa aiškesnis. Be to, gali būti naudinga atskirti viršūnes, kuriose pradėtas vykdyti paieškos gilyn algoritmas, bet

[25] Procedūra `ieškok_gilyn` nepakeičia grafo *g*, tačiau kintamasis *g* perduodamas kaip parametras-kintamasis, nes kurti sudėtingos duomenų struktūros kopiją būtų neefektyvu.

nebaigtas (pilkas viršūnes), pavyzdžiui, norint pritaikyti paieškos gilyn algoritmą ciklo paieškai.

Procedūros `ieškok_gilyn` sudėtingumas yra O(b), kur b yra grafo briaunų skaičius. Tokiam efektyvumui pasiekti grafą būtina vaizduoti kaimynystės sąrašais. Jei grafą vaizduotume kaimynystės matrica, galėtume pasiekti tik O(n^2) sudėtingumą.

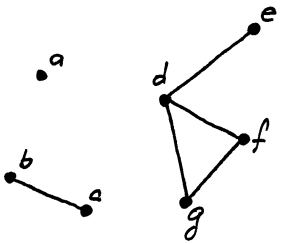


40 pav. Labirintas; brūkšnine linija pavaizduotas kelias, kuris rodo, kaip apeinamas labirintas

Kaip tik paieška gilyn ir naudojosi Tesėjas, ieškodamas labirinte Minotauro. Kiekvienoje koridorių sankirtoje jis pasirinkdavo tolimesnę kryptį ir jei prieidavo aklavietę, grįždavo atgal iki praeitos sankirtos bandyti kitos krypties. O jei toje sankirtoje visi koridoriai jau išbandyti – grįždavo į dar ankstesnę sankirtą ir taip toliau. Siūlas padėjo Tesėjui rasti Minotaurą.

7.4 Patikrinimas, ar grafas jungus

Grafas yra **jungus**, jei iš bet kurios viršūnės galima pasiekti bet kurią kitą viršūnę einant briaunomis. Priešingu atveju grafas vadinamas **nejungiu**.



41 pav. Nejungus grafas, sudarytas iš trijų jungumo komponentų

Nejungus grafas yra sudarytas iš jungių dalių, vadinamų **jungumo komponentais.**

Grafo jungumą tenka tikrinti sprendžiant įvairiausius uždavinius. Paprasčiausia tai padaryti taikant paiešką į gylį grafe. Pritaikysime praeitame skyrelyje pateiktą algoritmą, grafą vaizduosime kaimynystės sąrašais. Šiuo atveju viršūnes užteks spalvinti tik dviem spalvomis (t. y. atskirti aplankytas nuo neaplankytų), tad tam naudosime loginį masyvą. Pirminės viršūnės taip pat nesvarbios, taigi paiešką gilyn realizuoti bus paprasčiau. Tačiau paieškos gilyn procedūrą papildysime skaičiavimu, kiek viršūnių aplankyta. Grafas yra jungus tada ir tik tada, jei įvykdžius paiešką gilyn iš bet kurios jo viršūnės, bus aplankytos **visos** grafo viršūnės.

```
function jungus(var g : grafas) : boolean;
var aplankyta : array [1..MAXN] of boolean;

    procedure ieškok_gilyn(v : integer;
                           var sk : integer);
    { v - aplankoma viršūnė, sk – aplankytų viršūnių skaičius }
    var u, i : integer;
    begin
        aplankyta[v] := true;
        inc(sk);
        with g do
            for i := 1 to vir[v].k do begin
                u := vir[v].ksąrašas[i];
                if not aplankyta[u] then
                    ieškok_gilyn(u, sk);
            end;
    end;

var v, sk : integer;
begin
    for v := 1 to g.n do
        aplankyta[v] := false;
    sk := 0;
    ieškok_gilyn(1, sk);
    { jei buvo aplankytos visos viršūnės – tai grafas jungus }
    jungus := sk = g.n;
end;
```

7.5 Uždavinys *Epidemijos modeliavimas:* grafo jungumo komponentų paieška

Taikydami grafų teoriją išspręsime pirmą konkretų uždavinį *Epidemijos modeliavimas*[26]*:*

> *Plinta pavojinga paukščių liga. Jeigu paukštis užsikrečia šia liga, tai nuo jo užsikrės visi kiti paukščiai, turintys su juo nuolatinius kontaktus, po to nuo jų užsikrės dar kiti (turintys nuolatinius kontaktus su naujai užsikrėtusiais) ir t. t. Paukščiai, neturintys tarpusavyje nuolatinių kontaktų, tiesiogiai vienas nuo kito užsikrėsti negali.*
>
> ***Užduotis****: Žinoma, kad m paukščių jau yra užsikrėtę liga, ir žinomos visos paukščių poros, turinčios nuolatinius kontaktus. Deja, nežinoma, kurie iš visų n paukščių jau yra užsikrėtę. Reikia nustatyti, kiek daugiausiai šios rūšies paukščių gali užsikrėsti dėl epidemijos.*

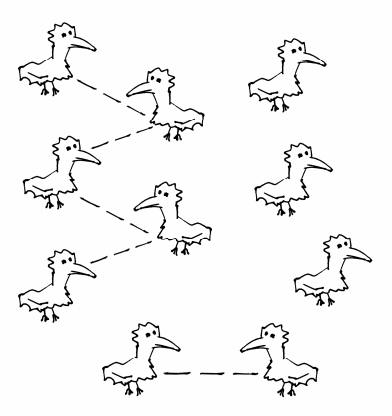
Paukščiai atitiks grafo viršūnes, o nuolatiniai kontaktai – briaunas. Grafas gali būti nejungus, t. y. jame gali egzistuoti keletas jungių komponentų, kuriuos toliau sprendimo aprašyme vadinsime paukščių šeimomis. Atskiru atveju šeimą gali sudaryti tik vienas paukštis.

Jei užsikrės nors vienas paukštis iš šeimos, tai nuo šio paukščio užsikrės visa šeima. Tad užsikrėtusių paukščių bus daugiausiai, jei iš pradžių bus užsikrėtę po vieną paukštį iš kuo gausesnių šeimų.

---

[26] Šis uždavinys buvo pateiktas Lietuvos moksleivių informatikos olimpiados III etape 2006 metais.

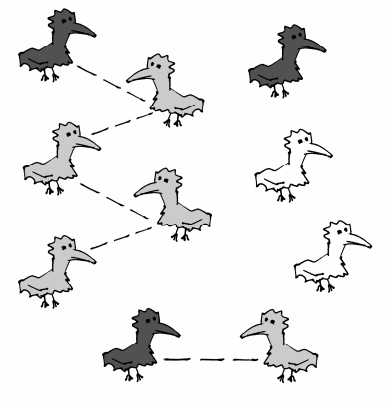
Taigi norint išspręsti šį uždavinį, reikia rasti viršūnių skaičių kiekviename **jungumo komponente**, tuomet jas išrikiuoti nedidėjimo tvarka ir suskaičiuoti, kiek yra viršūnių didžiausiuose m komponentų.



42 pav. Kontaktus palaikantys paukščiai sujungti punktyrine linija

Piešinyje pateiktame pavyzdyje yra penkios paukščių šeimos: tris šeimas sudaro vieniši paukščiai, vieną šeimą sudaro paukščių pora, o dar vieną – penki paukščiai. Išrikiavę gautume: 5, 2, 1, 1, 1.

Sakykime, užsikrėtė 3 paukščiai. Tad didžiausias galimų užsikrėtusių paukščių skaičius lygus: $5 + 2 + 1 = 8$.



43 pav. Užsikrėtus trims, gali nukentėti daugiausiai aštuoni paukščiai

Tačiau kaip ieškoti jungumo komponentų? Pasirinkime bet kurią grafo viršūnę – ji priklauso kažkokiam grafo jungumo komponentui. Jei pradėdami joje įvykdysime paiešką gilyn, tai bus aplankomos visos komponento viršūnės. Todėl norėdami suskaičiuoti, kiek jungumo komponentų sudaro grafą, galime iteruoti per visas grafo viršūnes ir radę neaplankytą,

vykdyti paiešką gilyn (aplankančią visas aptikto komponento viršūnes). Kiek kartų iteruodami aptiksime neaplankytą viršūnę, tiek ir jungumo komponentų yra grafe.

Šiame uždavinyje svarbu sužinoti ir pačių komponentų dydžius, todėl panaudosime paieškos gilyn procedūrą, kurią naudojome grafo jungumo tikrinimui – įsimenančią, kiek viršūnių buvo aplankyta paieškos metu.

```
type log_mas = array [1..MAXN] of boolean;
     masyvas = array [1..MAXN] of integer;

function užsikrės(var g : grafas;
                      m : integer): integer;
{ m – jau užsikrėtusių paukščių skaičius;
  g – grafas, vaizduojamas kaimynystės sąrašais }

var aplankyta : log_mas;
    i, komp_sk, iki : integer;
    komp_dydis : masyvas;

begin
    for i := 1 to g.n do
        aplankyta[i] := false;
    komp_sk := 0;

    for i := 1 to g.n do
        if not aplankyta[i] then begin
             komp_sk := komp_sk + 1;
             komp_dydis[komp_sk] := 0;
             ieškok_gilyn[27](i, komp_dydis[komp_sk]);
        end;
    rikiuok[28](komp_sk, komp_dydis);
```

[27] Procedūros `ieškok_gilyn` tekstą rasite 7.4 skyrelyje.

[28] Procedūros `rikiuok` tekstą rasite 6.2 skyrelyje (jei pasirinksite rikiavimą įterpimu) arba 6.3 skyrelyje (jei pasirinksite greitąjį rikiavimą ir modifikuosite kreipinį).

```
    užsikrės := 0;
    { užsikrėtusių paukščių gali būti daugiau
      nei jungumo komponentų }
    if m > komp_sk then
        iki := 1
    else
        iki := komp_sk - m + 1;
    for i := komp_sk downto iki do
        užsikrės := užsikrės + komp_dydis[i];
end;
```

7.6 Paieška platyn

Paieškos platyn (angl. *Breadth-First Search, BFS*) algoritmas aplanko viršūnes pagal griežtą taisyklę: pradėjus nuo pasirinktos viršūnės (tarkime, *p*), aplankomos visos viršūnės, kurios pasiekiamos iš *p* viena briauna (vienu ėjimu), tuomet – pasiekiamos iš *p* dvejomis briaunomis (dviem ėjimais) ir t. t.

Kaip užtikrinti tokią viršūnių lankymo tvarką? Algoritmas naudoja aplankytų viršūnių eilę: pirmiausia į eilę įrašoma pradinė viršūnė; kol eilė netuščia, iš jos pradžios imama viršūnė ir visos neaplankytos jos kaimynės įrašomos į eilės galą. Šitaip eilėje pirmiausia atsiduria viršūnės, pasiekiamos viena briauna, tada – dviem briaunomis ir t. t.

Programuodami paiešką gilyn panaudojome rekursiją, tad nekonstravome savo dėklo[29] duomenų struktūros. Paieškai platyn jau reikalinga *eilės* duomenų struktūra:

```
type eilė = record
          duom : array [1..MAXN] of integer;
          pradžia, pabaiga : integer;
      end;
```

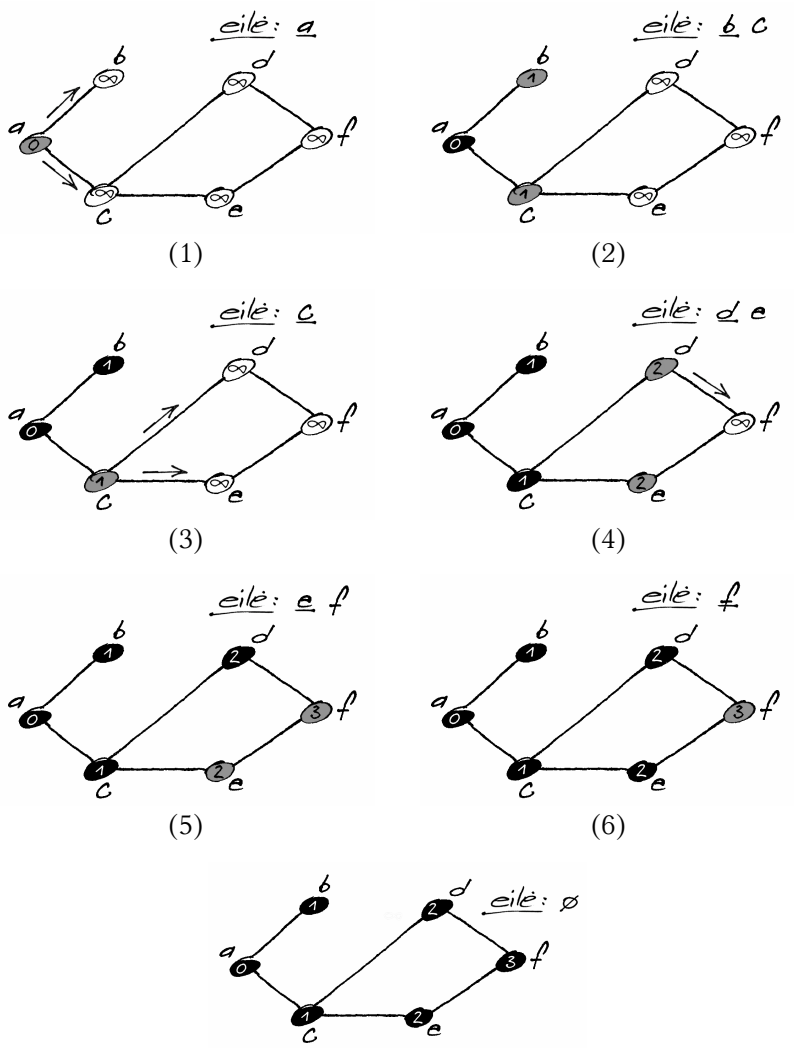
[29] Žr. 4.1 skyrelį.

Nauji elementai dedami į eilės galą, o imami iš eilės pradžios. Žemiau pateiktos procedūros su eile atlieka veiksmus, kurių reikės paieškos platyn algoritmui:

```
procedure išvalyk(var eil : eilė);
begin
    eil.pradžia := 0;
    eil.pabaiga := 0;
end;

function tuščia(var eil : eilė) : boolean;
begin
    tuščia := eil.pradžia = eil.pabaiga;
end;

procedure įdėk(var eil : eilė; x : integer);
begin
    eil.pabaiga := eil.pabaiga + 1;
    eil.duom[eil.pabaiga] := x;
end;

function išimk(var eil : eilė) : integer;
begin
    eil.pradžia := eil.pradžia + 1;
    išimk := eil.duom[eil.pradžia];
end;
```

Kaip ir paieškos gilyn atveju, algoritmo vykdymo metu viršūnės spalvinamos balta, pilka ir juoda spalvomis, nors užtektų ir dviejų spalvų. Balta spalva nuspalvintos dar neaplankytos viršūnės, pilka – viršūnės, kurios įtrauktos į eilę, o juoda – jau išnagrinėtos (pašalintos iš eilės) viršūnės.

Paieškos platyn viršūnių aplankymo strategija garantuoja, kad kiekviena viršūnė iš pradinės bus aplankyta **trumpiausiu keliu** (jį sudaro mažiausias briaunų skaičius). Taigi paieška platyn – tinkamas algoritmas trumpiausio kelio paieškai grafuose, kurių visos briaunos yra lygiavertės (grafų teorijos terminais, **besvorės**).

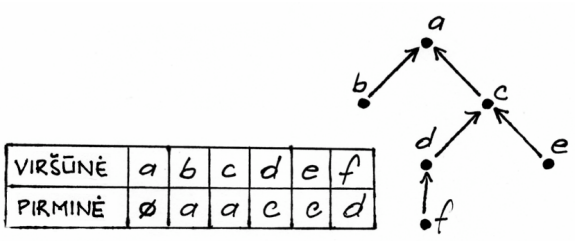


44 pav. Paieškos platyn veikimo iliustracija, kai pradinė viršūne pasirinkta viršūnė a

Trumpiausi atstumai iki kiekvienos viršūnės saugomi atskirame masyve. Kol nerastas kelias iki viršūnės, šis atstumas laikomas begaliniu. Grafas vaizduojamas kaimynystės sąrašais.

```
const BEGALINIS = MAXINT;
type spalvos = (balta, pilka, juoda);

var atstumas, { saugomi atstumai nuo pradinės iki
                visų kitų viršūnių }
    pirminė : array [1..MAXN] of integer;
    spalva : array [1..MAXN] of spalvos;

procedure ieškok_platyn(var g : grafas;
                        p : integer { pradinė viršūnė } );
var eil  : eilė;
    i, u, v : integer;
begin
    for v := 1 to g.n do begin
        atstumas[v] := BEGALINIS;
        pirminė[v] := 0;
        spalva[v] := balta;
    end;

    išvalyk(eil);
    { į eilę įtraukiama pradinė viršūnė }
    spalva[p] := pilka;  atstumas[p] := 0;
    pirminė[p] := 0;     įdėk(eil, p);

    while not tuščia(eil) do begin
        v := išimk(eil);
        with g do
        { dar neaplankytos (baltos) v kaimynės įtraukiamos į eilę }
            for i := 1 to vir[v].k do begin
                u := vir[v].ksąrašas[i];
                if spalva[u] = balta then begin
                    spalva[u] := pilka;
                    pirminė[u] := v;
                    atstumas[u] := atstumas[v] + 1;
                    įdėk(eil, u);
                end;
            end;
        spalva[v] := juoda;
    end;
end;
```

Jei reikia išspausdinti trumpiausią kelią nuo pradinės viršūnės iki viršūnės *u*, naudojamės sudarytu pirminumo medžiu ir kreipiamės į procedūrą `spausdink_kelią(u)` – ji pateikta 7.3 skyrelyje.



45 pav. Paieškos platyn pirminumo medis

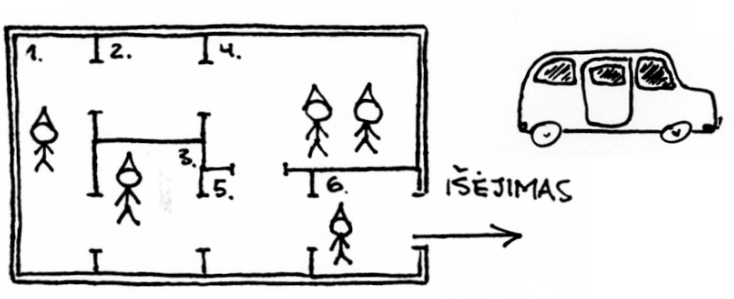
Algoritmo sudėtingumas yra toks pat kaip ir paieškos gilyn: $O(n + b)$, jei grafas vaizduojamas kaimynystės sąrašais, ir $O(n^2)$, jei grafas vaizduojamas kaimynystės matrica. Čia n – grafo viršūnių, b – briaunų skaičius.

7.7 Uždavinys *Nykštukai*[30]

Nykštukai gyvena vienaukščiame name, kuriame yra daug kambarių. Jei tarp dviejų kambarių yra durys, tai galima pereiti iš vieno kambario į kitą. Įėjimas iš namo yra tik pro vienintelį kambarį. Namas stebuklingas ir durys gali būti tarp bet kurių kambarių. Išėjimas pro duris į kitą kambarį arba į lauką užtrunka vieną laiko vienetą.

[30] Analogiškas uždavinys buvo pateiktas Lietuvos informatikos olimpiados III etape 2005 metais.

Netikėtai nykštukai sužinojo, kad po t laiko vienetų iš aikštelės šalia namo išvažiuoja autobusas į NKL (Nykštukų krepšinio lygos) finalines varžybas. Žinoma, kiek nykštukų yra name ir kokiuose kambariuose. Kiekvienas nykštukas žino greičiausią kelią iki išėjimo ir iš namo bėgs būtent tuo keliu.

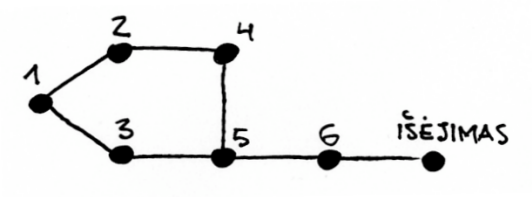


46 pav. Namo išplanavimo pavyzdys; parodyta, kuriuose kambariuose pradiniu momentu yra nykštukai

Užduotis. *Reikia nustatyti, kurie nykštukai suspės į autobusą, jeigu kiekvienas bėgs greičiausiu keliu.*

Uždavinį modeliuojame grafu: kambariai bei išėjimas laikomi grafo viršūnėmis, o durys – briaunomis.

Reikia sužinoti, per kiek mažiausiai laiko vienetų kiekvienas nykštukas gali išbėgti laukan. Laiko vienetų skaičius lygus perbėgamų durų skaičiui, t. y. briaunų skaičiui mūsų sudarytame grafe. Taigi ieškome trumpiausių kelių iš viršūnių, kuriose „stovi“ nykštukai, iki išėjimo viršūnės. Nepamirškime – mus domina ne patys keliai, o tik jų ilgiai.



47 pav. Pavyzdyje pateiktą namą atitinkantis grafas

Trumpiausio kelio paieškai galime panaudoti ką tik išmoktą paieškos platyn algoritmą, ir vykdyti jį iš kiekvienos viršūnės, kurioje „stovi" bent vienas nykštukas. Tačiau galima uždavinį išspręsti efektyviau: įsivaizduokime, kad lauke stovi vienas nykštukas (pavyzdžiui, autobuso vairuotojas?); jei rasime visus nykštukus, pas kuriuos šis nykštukas gali atbėgti per t ar mažiau laiko vienetų, tai ir išspręsime uždavinį. Pakanka **vieną kartą** įvykdyti paieškos platyn algoritmą iš išėjimo viršūnės. Žinant trumpiausių kelių ilgius nuo išėjimo iki kiekvieno kambario, nesunku baigti spręsti uždavinį.

Toliau pateiktame programos fragmente paieška platyn realizuota paprasčiau, kadangi mus domina ne patys keliai, o tik atstumai. Neaplankytas viršūnes atpažinsime ne pagal spalvą, o pagal tai, kad atstumas iki jų yra pažymėtas begaliniu. Grafas vaizduojamas kaimynystės sąrašais.

```
const BEGALINIS = MAXINT;
      MAXN = ...; {maksimalus kambarių (grafo viršūnių) skaičius}
      MAXK = ...; {maksimalus nykštukų skaičius}

type  masyvas = array [1..MAXK] of integer;
      loginis = array [1..MAXK] of boolean;

var atstumas : array [1..MAXN] of integer;
```

```
procedure ieškok_platyn(var g : grafas;
                        p : integer { pradinė viršūnė } );

var eil : eilė;
    i, u, v : integer;

begin
    for v := 1 to g.n do
        atstumas[v] := BEGALINIS;
    išvalyk(eil);
    { į eilę įtraukiama pradinė viršūnė }
    atstumas[p] := 0;
    idėk(eil, p);
    while not tuščia(eil) do begin
        v := išimk(eil);
        with g do
        { visos dar neaplankytos (pažymėtos begaliniu atstumu)
          v kaimynės įtraukiamos į eilę }
            for i := 1 to vir[v].k do begin
                u := vir[v].ksąrašas[i];
                if atstumas[u] = BEGALINIS then begin
                    atstumas[u] := atstumas[v] + 1;
                    įdėk(eil, u);
                end;
            end;
    end;
end;

procedure kas_spės(var g : grafas;
                   var kamb : masyvas;
                   { kamb[i] – i-ojo nykštuko kambario numeris }
                   išėjimas, t : integer;
                   var spės : loginis);
begin
    ieškok_platyn(g, išėjimas);
    for i := 1 to nyk_sk do
       spės[i] := atstumas[kamb[i]] <= t;
end;
```

8 OILERIO IR HAMILTONO CIKLAI

The whole branch of mathematics was born out of the game.
Iš žaidimų kilo net atskira matematikos šaka.
Nežinomas autorius apie grafų teorijos atsiradimą

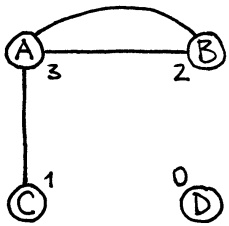
Pažintį su grafais pradėjome nuo istorijos apie septynių Karaliaučiaus tiltų uždavinį. Šiame skyrelyje papasakosime, kaip tą uždavinį išsprendė Oileris, taip pat išspręsime jau ne kartą sutiktą Hamiltono ciklų uždavinį.

Tam prireiks keleto naujų grafų teorijos sąvokų, tad nuo jų ir pradėkime.

8.1 Kelios grafų teorijos sąvokos

Briaunų, jungiančių viršūnę v su kokia nors kita viršūne, skaičius vadinamas viršūnės v **laipsniu**. Paprastuose grafuose viršūnės laipsnis yra jai gretimų viršūnių skaičius, tačiau multigrafuose (grafuose, kuriuose dvi viršūnes gali jungti daugiau nei viena briauna), šie du skaičiai gali skirtis.

Briauna, kurią pašalinus iš grafo padidėtų jungumo komponentų skaičius, vadinama **tiltu**. Kitaip tariant, jei norėtume iš kurios nors viršūnės pasiekti kitą, ir tam **būtinai** turėtume eiti briauna (u, v), ši briauna ir būtų tiltas.



48 pav. Grafo viršūnių laipsniai užrašyti šalia viršūnių; grafas turi vienintelį tiltą – briauną AC

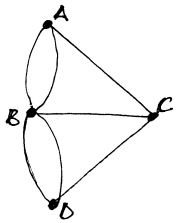
8.2 Oilerio keliai ir ciklai

49 pav. Šveicarų matematikas Leonardas Oileris (Leonhard Euler), 1707–1783

Jau minėjome, kad Oileris neigiamai atsakė į Karaliaučiaus gyventojus dominantį klausimą: nėra maršruto, kuris prasidėtų viename iš krantų, pereitų kiekvienu iš septynių tiltų lygiai vieną kartą ir baigtųsi tame pačiame krante. Išnagrinėjęs tokių maršrutų galimybę, Oileris padėjo grafų teorijos pagrindus.

Jam pavyko išspręsti šį uždavinį: Oileris nustatė, kokios sąlygos yra būtinos ir pakankamos, kad grafe egzistuotų toks maršrutas, be to, pateikė algoritmą jam rasti. Maršrutai, prasidedantys ir užsibaigiantys toje pačioje viršūnėje bei apeinantys visas grafo briaunas po vieną kartą, nuo tada vadinami **Oilerio ciklais** arba (jei baigiasi kitoje viršūnėje negu prasideda) **Oilerio keliais**.

Nėra sudėtinga Karaliaučiaus tiltų atveju įrodyti, kad neegzistuoja Oilerio ciklas. Prisiminkime, kaip atrodė grafas, kurio viršūnės – Karaliaučiaus salos ir krantai, o briaunos – juos jungiantys tiltai (50 pav.).



50 pav. Karaliaučiaus tiltų uždavinį atitinkantis

Ieškome maršruto, kuris kiekviena briauna pereitų vieną kartą. Panagrinėjus kurią nors vieną viršūnę, nesunku pastebėti, jog ieškomas maršrutas neįmanomas, jei viršūnės laipsnis nelyginis: juk išeiti

iš šios viršūnės turėsime lygiai tiek kartų, kiek ir atėjome, ir kiekvienąkart vis kita briauna, taigi visas briaunų skaičius turi būti dviejų kartotinis, lyginis. Tačiau šiame grafe visų viršūnių laipsniai nelyginiai (3, 3, 3 ir 5), taigi ir toks maršrutas tikrai neįmanomas.

Įrodėme, kad toks maršrutas negalimas, jei bent vienos viršūnės laipsnis nelyginis. Oileris įrodė griežtesnį teiginį: **Oilerio ciklas egzistuoja tada ir tik tada, kai visų viršūnių laipsniai yra lyginiai**.

O kaip su Oilerio keliais? Jei maršrutas turi baigtis ne toje pačioje viršūnėje kur prasidėjo, tai minėta taisyklė turi galioti visoms viršūnėms, išskyrus pradinę ir galinę. **Oilerio kelias egzistuoja tada ir tik tada, kai lygiai dviejų viršūnių laipsniai yra nelyginiai**.

Be abejo, grafai, turintys Oilerio ciklą arba kelią, turi pasižymėti dar viena natūralia savybe – jie turi būti jungūs[31].

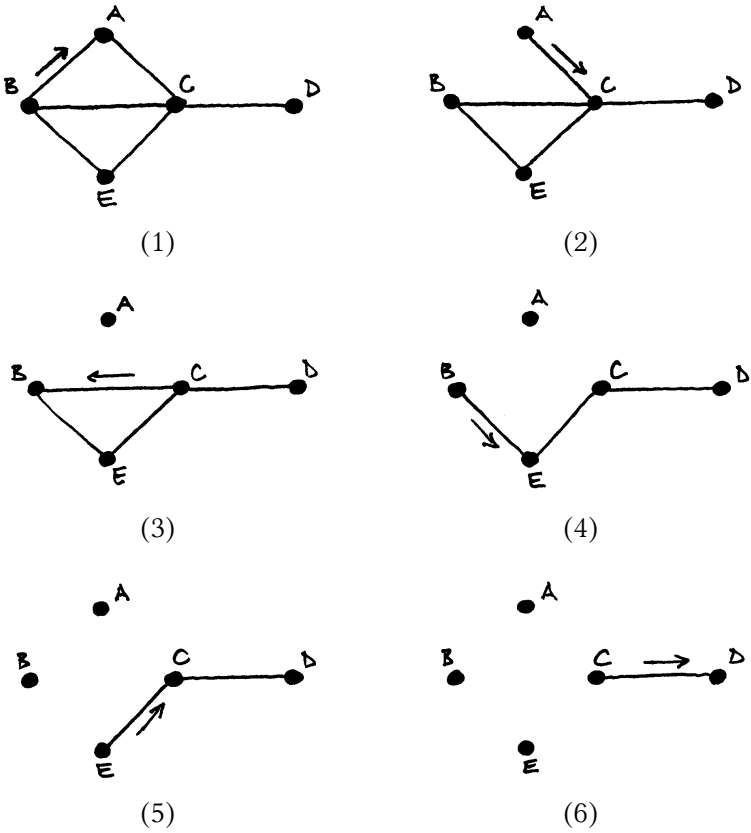
8.3 Flerio algoritmas

Žinome, kas yra Oilerio kelias ir ciklas, ir netgi mokame duotame grafe nustatyti, ar toks egzistuoja. Tačiau kaip ieškoti Oilerio kelio? Pasirodo, tai labai paprasta. Flerio algoritmas, skirtas Oilerio ciklų ir kelių paieškai, gali būti nusakytas trimis žodžiais: *venk eiti tiltu*.

[31] Išimtis, jei grafe yra izoliuotų (t.y. iš kurių neišeina nė viena briauna) viršūnių; tokiu atveju Oilerio ciklas gali egzistuoti, nors grafas ir nejungus.

Flerio algoritmą, ieškantį Oilerio ciklo, galime išskaidyti į tokius žingsnius:

- pradedame bet kurioje viršūnėje;
- jei įmanoma, einame briauna, kuri nėra tiltas (jei tokios briaunos nėra, tai einame tiltu);
- briauną, kuria jau ėjome, pašaliname iš grafo;
- kartojame nuo antro žingsnio, o jei nėra kur eiti, baigiame.



51 pav. Flerio algoritmo Oilerio keliui rasti iliustracija

Oilerio kelio paieška skiriasi tik pradinės viršūnės pasirinkimu: pradedame viršūnėje su nelyginiu laipsniu.

Flerio algoritmą Oilerio ciklo arba kelio paieškai multigrafe (t. y. grafe, kuriame dvi viršūnes gali jungti daugiau negu viena briauna) užrašysime programavimo kalba. Multigrafą vaizduosime kaimynystės matrica – kiekvienai porai viršūnių įsiminsime, kelios briaunos jas jungia:

```
type grafas = record
          n : integer;
          briaunų_sk : array [1..MAXN,
                              1..MAXN] of integer;
          laipsnis : array [1..MAXN] of integer;
      end;
```

Tarsime, kad masyvas `laipsnis` užpildomas sudarant grafą.

Prieš pradedant ieškoti svarbu įsitikinti, ar tenkinamos būtinos ir pakankamos sąlygos. Paprastumo dėlei tarsime, kad grafas jungus, arba jį sudaro tik vienas nevienetinio dydžio jungumo komponentas. Šias sąlygas nesunku patikrinti pasinaudojus paieška gilyn, kaip tai darėme 7.4 skyrelyje.

Viršūnių laipsnių ribojimą patikrinti dar paprasčiau: tereikia suskaičiuoti, kiek viršūnių grafe turi nelyginius laipsnius. Jei tokios bus dvi, tai ieškosime Oilerio kelio ir turėsime pradėti vienoje iš nelyginio laipsnio viršūnių, priešingu atveju galėsime pradėti bet kurioje viršūnėje.

Patikrinus, ar tenkinamos abi sąlygos, galima pradėti vykdyti Flerio algoritmą: įsiminti pradinę viršūnę, pasirinkti tolesnę ir briauną, kuria jau ėjome, išbraukti iš grafo. Tolesnę lankomą viršūnę renkamės pagal minėtą sąlygą – stengiamės neiti tiltu, jei tik įmanoma.

```
const MAXB = ...; { maksimalus briaunų skaičius }
type  masyvas = array [1..MAXB+1] of integer;

procedure Flerio(var g : grafas;
                 var kelio_ilgis : integer;
                 var kelias : masyvas);
{ jei Oilerio ciklas/kelias grafe neegzistuoja, tai „kelio_ilgis“ reikšmė lygi
  nuliui, kitu atveju Oilerio ciklas/kelias įrašomas į masyvą „kelias“ }
var k, p, v, u, nelyg : integer;
begin
    nelyg := 0;
    { suskaičiuojama, kiek yra nelyginio laipsnio
    viršūnių, ir parenkama pradinė (v) }
    v := 1;
    for k := 1 to g.n do
        if odd(g.laipsnis[k]) then begin
            nelyg := nelyg + 1;
            { jei randama bent viena nelyginio laipsnio viršūnė,
              tai v priskiriamas jos numeris }
            v := k;
        end;
    kelio_ilgis := 0;
    if ((nelyg = 0) or (nelyg = 2))
    { jei tenkinamos būtinos Oilerio ciklo/kelio egzistavimo sąlygos }
    then begin { vykdomas Flerio algoritmas }
        while v > 0 do begin
            inc(kelio_ilgis);
            kelias[kelio_ilgis] := v;
            p := v; { paskutinė pereita viršūnė }
            v := 0;
            { pagal Flerio algoritmą pasirenkama sekanti viršūnė }
            for u := 1 to g.n do
                if (g.briaunų_sk[p, u] > 0) and
                   ((v = 0) or not tiltas(g, p, u))
                then
                    v := u;
            if v > 0 then begin { ištrinama briauna }
                dec(g.briaunų_sk[p, v]);
                dec(g.briaunų_sk[v, p]);
            end;
        end;
    end;
end;
```

Liko nerealizuota funkcija `tiltas`. Ji turėtų grąžinti `true` reikšmę, jei grafe g briauna (u, v) yra tiltas. Tai patikrinti nesunku: jei (u, v) yra tiltas, tai pašalinus šią briauną viršūnės u ir v atsidurs skirtinguose jungumo komponentuose. Taigi pašalinkime šią briauną, paieška gilyn patikrinkime, ar v pasiekiama iš u, ir sugrąžinę pašalintą briauną pateikime rezultatą.

```
function tiltas(var g : grafas;
                u, v : integer) : boolean;
var k : integer;
begin
    if g.briaunų_sk[u, v] > 1 then
        tiltas := false
    else begin
        for k := 1 to g.n do
            spalva[k] := balta;

        g.briaunų_sk[u, v] := 0; { pašalinama briauna }
        g.briaunų_sk[v, u] := 0;
        ieškok_gilyn[32](g, u);
        g.briaunų_sk[u, v] := 1; { atstatoma briauna }
        g.briaunų_sk[v, u] := 1;

        tiltas := spalva[v] = balta;
    end;
end;
```

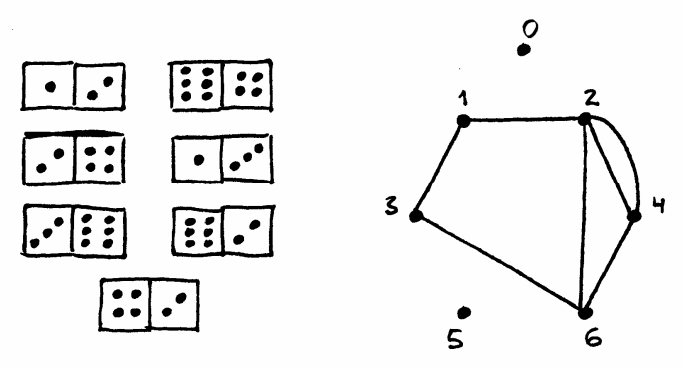
[32] Ši procedūra pateikta 7.3 skyrelyje, tačiau kitaip pavaizduotam grafui, taigi prieš taikant ją būtina modifikuoti.

8.4 Uždavinys *Domino kauliukai*[33]

Yra krūvelė domino kauliukų. Kiekvienas domino kauliukas perskirtas pusiau, kiekvienoje pusėje užrašytas skaičius iš intervalo 0..6. Du kauliukus galima sujungti, jei sutampa skaičiai, užrašyti ant sujungiamų kauliukų pusių.

Užduotis. *Reikia nustatyti, ar krūvelėje esančius kauliukus galima sudėlioti į nenutrūkstamą liniją.*

Uždavinį modeliuosime grafais. Grafas turės septynias viršūnes, sunumeruotas nuo 0 iki 6 (mat nuo 0 iki 6 taškų gali būti ant domino kauliuko puselės). Kauliukus atitiks grafo briaunos.



52 pav. Kauliukų rinkinys ir juos atitinkantis grafas; grafe Oilerio kelias yra toks: 6—4—2—1—3—6—2, tad kauliukus galima sudėlioti į vieną eilę: [*6,4*] [*4,2*] [*2,1*] [*1,3*] [*3,6*] [*6,2*] [*2,4*]

[33] Šis uždavinys buvo pateiktas Lietuvos informatikos olimpiados III etape 1995 metais.

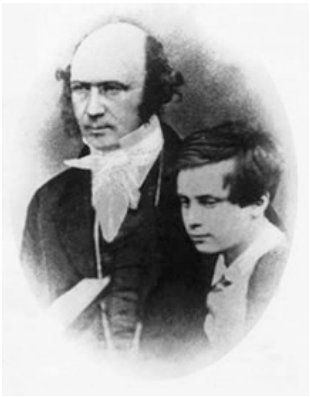
Kauliukų dėliojimas į liniją atitinka kelią, kai visomis grafo briaunomis apeinama po vieną kartą, t. y. Oilerio kelią. Norint išspręsti šį uždavinį tereikia patikrinti, ar grafe egzistuoja Oilerio kelias.

8.5 Hamiltono keliai ir ciklai

O brooding Spirit of Wisdom and of Love,
Whose mighty wings even now o'ershadow me,
Absorb me in thine own immensity,
And raise me far my finite self above!
Mąslioji išminties ir meilės siela,
kurios eiklių sparnų šešėlyje slepiuos,
leisk prisiliesti prie gelmės tavos
ir peržengt savo ribotumo sieną! [34]

Seras Viljamas Rovanas Hamiltonas (Sir William Rowan Hamilton)

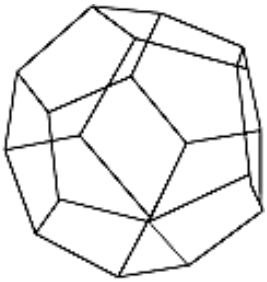
Airija nėra šalis, kurią garsina matematikai. Tačiau vieną jų – matematiką ir poetą serą Viljamą Rovaną Hamiltoną – žino daugelis. Deja, labai gabaus, daug kalbų žinojusio mokslininko asmeninis gyvenimas buvo nenusisekęs: jis sirgo alkoholizmu ir depresija. 1859 metais – pasakoja, jog pristigęs pinigų – jis sukonstravo ir



53 pav. Seras Viljamas Rovanas Hamiltonas, (Sir William Rowan Hamilton) 1805–1865 su vienu iš sūnų apie 1845 m.

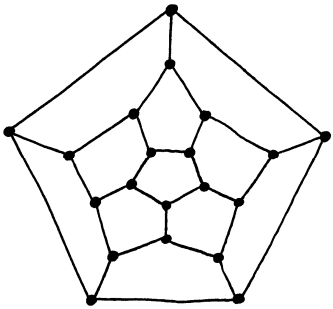
[34] Eiles vertė Gediminas Pulokas.

Dublino žaislų gamintojams pardavė galvosūkį „Aplink pasaulį“ – iš medžio pagamintą taisyklingą dodekaedrą su dvidešimčia viršūnių. Ant kiekvienos viršūnės buvo užrašytas miesto pavadinimas. Tai buvo galvosūkis: reikėjo rasti kelią dodekaedro briaunomis, kuriuo būtų kiekvienas miestas aplankytas po vieną kartą.



54 pav. Erdvinis dodekaedro vaizdas

Hamiltono keliu imta vadinti kelią, kuriuo einant kiekviena viršūnė aplankoma po vieną kartą.



55 pav. Dodekaedras, pavaizduotas plokštumoje

Atrodytų, Hamiltono kelio uždavinys nedaug skiriasi nuo Oilerio kelio uždavinio: vienu atveju reikia rasti kelią, kuris apeitų visas viršūnes po vieną kartą, kitu atveju – kelią, kuris po vieną kartą apeitų visas briaunas. Tačiau jų sprendimai iš esmės skiriasi. Oilerio ciklus ir kelius sėkmingai randa Flerio algoritmas, o **Hamiltono kelio paieška – NP sunkus uždavinys**. Tai reiškia, kad nėra žinoma jokio efektyvaus algoritmo šiam uždaviniui spręsti.

Hamiltono kelio paieško uždavinys labai artimas keliaujančio pirklio uždaviniui, su kuriuo jau susidūrėme 1.7 skyrelyje. Pastarajame uždavinyje grafas yra **pilnas** (t. y. bet kurios dvi viršūnės yra sujungtos briauna, nes egzistuoja kelias tarp bet kurių dviejų miestų) ir **svorinis** (grafo briaunoms yra priskirti svoriai, t. y. atstumai tarp miestų).

8.6 Hamiltono kelio paieška

Ieškodami visų Hamiltono kelių grafe, kurio viršūnės sunumeruotos nuo 1 iki n, galėtume generuoti visus skaičių nuo 1 iki n kėlinius (t. y. visas galimas viršūnių apėjimo tvarkas) k_1, k_2, ..., k_n, o sugeneravę patikrinti, ar egzistuoja visos briaunos (k_i, k_{i+1}), $(i = 1, 2, ..., n - 1)$.

Tačiau retuose (t. y. tokiuose, kurie turi nedaug briaunų) grafuose Hamiltono kelių galima ieškoti kur kas efektyviau. Viršūnių seką galima iškart sudaryti taip, kad gretimas sekos viršūnes jungtų briauna. Naudodami grįžimo metodą parašysime procedūrą, spausdinančią visus konkrečioje viršūnėje v prasidedančius Hamiltono kelius. Grafą vaizduosime kaimynystės sąrašais.

```
const MAXN = ...;
var seka : array [1..MAXN] of integer;
    aplankyta : array [1..MAXN] of boolean;

procedure ieškok(var g : grafas;
                 k,           { kiek viršūnių apeita }
                 v : integer { kurioje viršūnėje sustota } );
var i, u : integer;
begin
    seka[k] := v;
    {aplankytomis žymimos konstruojamame kelyje esančios viršūnės}
    aplankyta[v] := true;
    if (k = g.n) then
        { jei apeitos visos viršūnės – tai rastas Hamiltono kelias}
        spausdink[35](g.n)
    else
        { bandoma toliau eiti į visas neaplankytas v kaimynes }
        for i := 1 to g.vir[v].k do begin
            u := g.vir[v].ksąrašas[i];
            if (not aplankyta[u]) then
                ieškok(g, k + 1, u);
        end;
    { pabaigus, v pažymima kaip neaplankyta }
    aplankyta[v] := false;
end;
```

Norint rasti Hamiltono kelius, prasidedančius visose viršūnėse, reikia įvykdyti:

```
for v := 1 to g.n do
    ieškok(g, 1, v);
```

Jei ieškotume ne kelių, o ciklų, tuomet sugeneravus visą seką reiktų papildomai patikrinti, ar egzistuoja briauna, jungianti pirmą ir paskutinę kelyje esančias viršūnes.

[35] Procedūra `spausdink(m)` išveda masyvo elementus nuo 1 iki *m*; ji analogiška 5.1 skyrelyje pateiktai procedūrai.

9 ORIENTUOTIEJI GRAFAI, TOPOLOGINIS RIKIAVIMAS

It is amazing how often messy applied problems have a simple description and solution in terms of classical graph properties.
Nuostabu, kaip dažnai sudėtingus praktinius uždavinius pavyksta paprastai suformuluoti ir išspręsti naudojant grafų teoriją.
Stivenas S. Skiena (Steven S. Skienna)

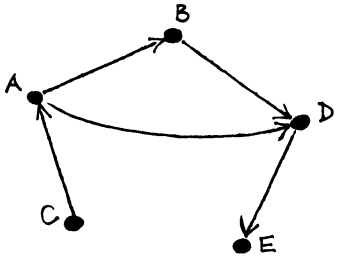
Iki šiol nagrinėjome tik paprasčiausius grafus: juose briaunos rodo, kad tarp dviejų objektų ryšys yra arba jo nėra. Tačiau ne visada ryšys tarp objektų esti vienodas. Šiame ir tolesniame skyreliuose praplėsime grafo sąvoką ir panagrinėsime uždavinius, modeliuojamus sudėtingesniais grafais.

9.1 Orientuotieji grafai

Kartais gali tekti grafais pavaizduoti nesimetrišką ryšį tarp objektų. Pavyzdžiui, jei grafu norėtume vaizduoti miesto planą, kur viršūnės atitiktų sankryžas, o briaunos – jas jungiančias gatves, tai galbūt tektų parodyti, kad kai kurios gatvės yra vienos krypties eismo: iš sankryžos A galima nuvažiuoti į sankryžą B, bet ne atvirkščiai. Tarkime, atėjo pavasaris ir grafu modeliuojame, kuri mergaitė patinka kuriam berniukui, ir kuriai mergaitei – kuris berniukas. Čia vėl (deja!) susiduriame su nesimetrišku sąryšiu: Pauliui gali patikti Milda, o ši galbūt net nežvilgtelės į jo pusę.

Taigi kartais prasminga grafo briaunoms suteikti kryptį. Grafas, kuriame briaunos turi kryptį, vadinamas **orientuotu** arba **kryptiniu**, o tokio grafo briaunos (su kryptimi) vadinamos **lankais**. Piešiant grafą, lankai vaizduojami rodyklėmis. Jei lankas nukreiptas iš viršūnės A į viršūnę B, sakoma, kad lankas **išeina iš viršūnės** A ir **ateina į viršūnę** B.

Ankstesniame skyrelyje buvome apibrėžę kai kurias grafų teorijos sąvokas, kurias reikėtų patikslinti orientuotiems grafams: viršūnės laipsnis, kelias, jungus grafas.



56 pav. Orientuotojo grafo pavyzdys

Orientuoto grafo viršūnės **išėjimo laipsnis** lygus lankų, išeinančių iš šios viršūnės, skaičiui, o **įėjimo laipsnis** lygus įeinančių į viršūnę lankų skaičiui.

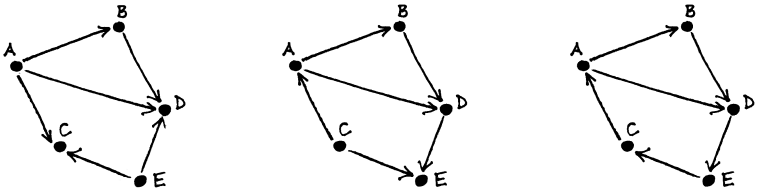
Kelias ir ciklas orientuotame grafe traktuojamas taip pat, kaip ir neorientuotame grafe, tačiau einama tik lanko kryptimi.

Orientuotame grafe yra keli jungumo tipai. Orientuotas grafas vadinamas **silpnai jungiu**, jei orientuotą grafą pakeitus jį atitinkančiu neorientuotu grafu (t. y. grafo lankus pakeitus briaunomis), šis yra jungus.

Vienkryptiškai jungiu grafu vadinamas orientuotas grafas, jei pasirinkus bet kurias dvi viršūnes A ir B, egzistuoja kelias iš A į B arba iš B į A.

Orientuotas grafas yra **stipriai jungus** (stipriai susietas), jei iš bet kurios viršūnės galima pasiekti bet kurią kitą viršūnę.

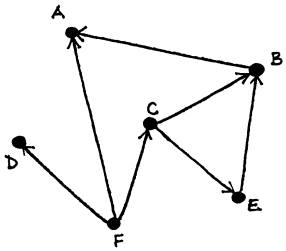
Visi stipriai jungūs grafai yra ir vienkryptiškai jungūs, o visi vienkryptiškai jungūs yra ir silpnai jungūs.

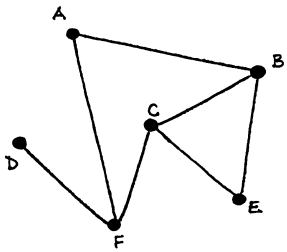


57 pav. Paveiksle pavaizduoti trys grafai;pirmasis jų – silpnai jungus, antrasis – vienkryptiškai jungus ir trečiasis – stipriai jungus

Orientuotas grafas yra bendresnis grafo atvejis, ir atvirkščiai – paprastas grafas G' yra atskiras orientuoto grafo G atvejis, kuriame grafo G' *briauną* (*u*, *v*) *atitinka du lankai* (*u*, *v*) ir (*v*, *u*). Taigi nereikia beveik jokių pakeitimų norint pavaizduoti orientuotą grafą kompiuteriu. Jei grafą vaizduojame kaimynystės matrica, tai ši matrica tiesiog nebus simetrinė (lanko įterpimas į grafą reikš reikšmės įrašymą į *vieną* matricos langelį). Jei grafą vaizduojame kaimynystės sąrašais, tai lanko įterpimas į grafą reikš *vienos* viršūnės kaimynių sąrašo papildymą (dar mažiau darbo negu vaizduojant paprastą grafą).

Beveik be jokių pakeitimų orientuotuose grafuose veiks jau aptarti algoritmai: paieška gilyn ir platyn, Oilerio ciklų bei Hamiltono ciklų paieška. Tiesa, algoritmai turės naują prasmę. Pavyzdžiui, paieškos gilyn ir platyn algoritmai aplankys ne visas vizualiai prijungtas viršūnes, bet tik tas, kurios pasiekiamos einant lankais (tik viena briaunos kryptimi). Šiek tiek skiriasi Oilerio ciklo egzistavimo sąlyga, tačiau ji labai natūrali: įeinančių lankų skaičius (įėjimo laipsnis) turi būti lygus išeinančių lankų skaičiui (išėjimo laipsniui) kiekvienoje viršūnėje (į kiekvieną viršūnę turime ateiti tiek kartų, kiek ir išeiti).





	A	B	C	D	E	F
A						
B						
C						
D						
E						
F						

	A	B	C	D	E	F
A						
B						
C						
D						
E						
F						

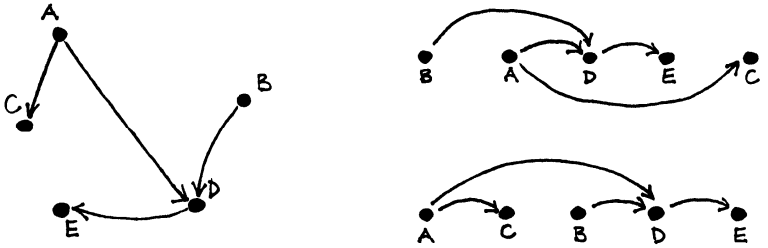
58 pav. Paveiksle pavaizduotas orientuotas grafas, jį atitinkantis neorientuotas grafas bei šiuos grafus atitinkanti kaimynystės matrica; matome, kad vienu atveju matrica yra simetriška, kitu atveju – ne

9.2 Topologinis rikiavimas

Įsivaizduokite, kad pradėjote ruoštis atostogoms ir norite keliauti į tolimą šalį. Teks nuveikti nemažai darbų: užsisakyti lėktuvo bilietus, numatyti ar užsisakyti nakvynės vietas, susipakuoti daiktus, galbūt gauti vizas, išsirinkti valstybę, į kurią vyksite, susiplanuoti maršrutą ir t. t. Akivaizdu, kad šių darbų bet kokia tvarka atlikti negalima. Prieš perkant lėktuvo bilietus būtina išsirinkti valstybę, į kurią vyksite, prieš numatant nakvynės vietas – susiplanuoti maršrutą, kuriuo keliausite. Reikia visus pasiruošimo atostogoms darbus surikiuoti į eilę taip, kad juos atlikdami ta tvarka sėkmingai išvyktume

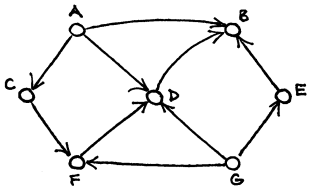
atostogauti. Darbus galime vaizduoti grafo viršūnėmis, o faktą, kad darbas A turi būti atliktas prieš darbą B, žymėti lanku iš A į B.

Šis uždavinys bus grafo **topologinio rikiavimo** uždavinys: orientuoto grafo viršūnes reikia išrikiuoti į vieną eilę taip, kad bet kuriam grafo lankui (u, v), toje eilėje viršūnė u eitų prieš viršūnę v.

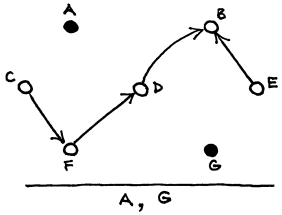


59 pav. Orientuotas beciklis grafas ir du skirtingi topologiniai jo išrikiavimai

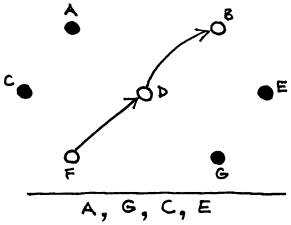
Ar visada galima topologiškai surikiuoti grafo viršūnes? Tarkime, kad darbas A turi būti atliktas prieš darbą B, darbas B – prieš darbą C, o darbas C – prieš darbą A. Topologiškai surikiuoti tokios darbų sekos neįmanoma, tačiau ir pačios darbų sekos turbūt negalima pavadinti korektiška. Tad grafo viršūnes topologiškai galima išrikiuoti, jei ir tik jei grafe nėra ciklų.



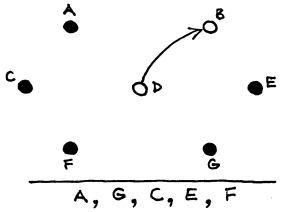
(1) Orientuotas beciklis grafas; viršūnių A ir G įėjimo laipsniai lygūs 0



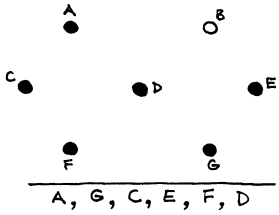
(2) Pašalinami lankai, išeinantys iš viršūnių A ir G, o šios viršūnės įtraukiamos į seką



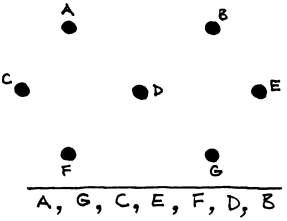
(3) . Į seką įtraukiamos naujos viršūnės C ir E, kurių laipsniai tapo lygūs 0



(4) Pašalinamas lankas, išeinantis iš viršūnės F, nes jos laipsnis lygus 0; viršūnė F įtraukiama į sekos galą



(5)



(6)

60 pav. Topologinio rikiavimo pavyzdys; įvykdžius visus algoritmo žingsnius, gaunama viršūnių seka, kuri yra topologinis grafo išrikiavimas

Topologinio rikiavimo algoritmas gana intuityvus: pirma išrenkamos ir į seką įtraukiamos viršūnės, kurių įėjimo laipsniai lygūs 0 (iš tiesų reikia pradėti nuo darbų, prieš kuriuos nieko daugiau nereikia atlikti). Tuomet pašalinami iš šių viršūnių išeinantys lankai ir atnaujinama informacija apie visų viršūnių laipsnius. Toliau vėl kartojami tie patys veiksmai, kol į seką įtraukiamos visos viršūnės.

Norint efektyviai vykdyti algoritmo žingsnius, grafą reikia vaizduoti kaimynystės sąrašais. Viršūnių, kurios neturi įeinančių lankų, galima ieškoti kiekvienąkart ciklu perbėgant visas viršūnes. Tačiau efektyviau laikyti eilę viršūnių, kurių įėjimo laipsniai lygūs 0, ir ją vis papildyti iš grafo trinant lankus. Tam panaudosime *eilės* duomenų struktūrą, aprašytą 7.6 skyrelyje.

```
const MAXN = ...;          { maksimalus viršūnių skaičius }
      MAXB = MAXN*MAXN;    { maksimalus lankų skaičius }

type viršūnė = record
         k : integer;           { kaimynių skaičius  ir sąrašas }
         ksąrašas : array [1..MAXN] of integer;
     end;

     grafas = record
         n : integer;                    { viršūnių skaičius }
         laipsnis : array [1..MAXN] of integer;
                                         { įėjimo laipsnis }
         vir : array [1..MAXN] of viršūnė;
                                         { viršūnių sąrašas }
     end;

procedure topologinis_rikiavimas(var g : grafas;
                                 var seka : masyvas);
{ topologinis išrikiavimas įrašomas į masyvą seka }
var v, u, i, nr : integer;
    eil : eilė;
begin
    išvalyk(eil);
```

```
    { į eilę įtraukiamos viršūnės, kurių įėjimo laipsniai lygūs 0 }
    for v := 1 to g.n do
        if g.laipsnis[v] = 0 then
            įdėk(eil, v);

    nr := 0; { išrikiuotų viršūnių sekos indeksas }
    while not tuščia(eil) do begin
        v := išimk(eil);
        nr := nr + 1;
        seka[nr] := v; { v įrašoma į seką }
        { „ištrinami“ iš v išeinantys lankai ir papildoma eilė }
        for i := 1 to g.vir[v].k do begin
            u := g.vir[v].ksąrašas[i]; { kaimynė }
            g.laipsnis[u] := g.laipsnis[u] - 1;
            if g.laipsnis[u] = 0 then
                įdėk(eil, u);
        end;
    end;
end;
```

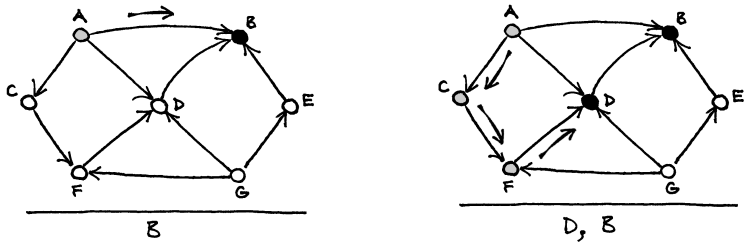
Jei baigus vykdyti algoritmą, į seką nebuvo įtrauktos visos viršūnės (t. y. `nr < g.n`), tai reiškia, kad grafe aptikta ciklų, ir topologinis išrikiavimas neįmanomas. Atkreipkite dėmesį, jog pateiktoje procedūroje grafo lankai iš tiesų nėra ištrinami, tik atnaujinama informacija apie viršūnių įeinančius laipsnius. Šio algoritmo sudėtingumas – $O(n + b)$, kur b – lankų skaičius.

Yra ir kitas tokio paties sudėtingumo topologinio rikiavimo algoritmas, naudojantis paieška gilyn; šį algoritmą realizuoti yra paprasčiau. Jo teksto nepateiksime, tik trumpai paaiškinsime idėją.

Pasirinkus bet kurią viršūnę, nuo jos atliekama paieška gilyn. Paieškos gilyn metu pirmiausia juodai nuspalvinamos „giliausios“ viršūnės: jei orientuotasis grafas neturi ciklų, tai viršūnė v bus nuspalvinta juodai tik tada, kai jau nuspalvintos juodai visos iš jos pasiekiamos viršūnės. Todėl jei spalvinant viršūnę juodai, ji dar ir įterpiama *į sekos pradžią*, tai gautoji seka ir yra topologinis grafo išrikiavimas.

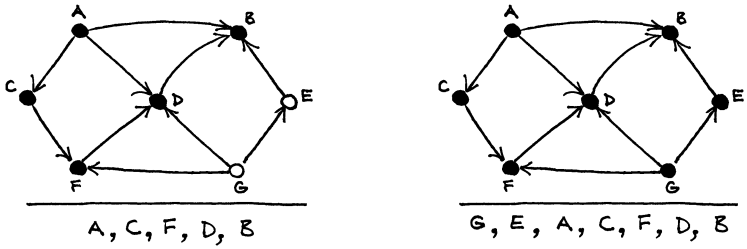
Jei paieška gilyn baigta, bet dar likę baltų viršūnių, tuomet vėl pasirenkama bet kuri balta viršūnė ir nuo jos atliekama paieška gilyn, kartojant jau aprašytus veiksmus. Šis algoritmas taip pat gali aptikti ciklą grafe: nagrinėjant viršūnės kaimynes neturi būti aptinkama *pilka* viršūnė, nes joje pradėta ir dar nebaigta paieška gilyn.

Žemiau pateikti paveikslai iliustruoja topologinį rikiavimą taikant paiešką gilyn.



(1) *(2)*

Pradėjus nuo pasirinktos viršūnės A, vykdoma paieška gilyn; juodai nuspalvintos viršūnės įtraukiamos į sekos pradžią; įtraukus į seką visas viršūnes šios atsidurs sekos pabaigoje



(3) *(4)*

Jei baigus vykdyti paiešką gilyn lieka baltų viršūnių, tai pasirenkama bet kuri iš jų ir vėl vykdoma paieška gilyn

61 pav. Topologinio rikiavimo, taikant paiešką gilyn, pavyzdys

9.3 Uždavinys *Abėcėlė*[36]

Dauguma mūsų moka išrikiuoti žodžius pagal abėcėlę. Šiame uždavinyje nagrinėsime atvirkščią procesą. Duotas nežinomos kalbos žodžių, surikiuotų pagal tos kalbos abėcėlę, sąrašas. Į pateiktus žodžius įeina visos tos kalbos abėcėlės raidės.

Užduotis. *Reikia rasti šios nežinomos kalbos abėcėlę.*

Visos raidės rikiavimo ir abėcėlės požiūriu laikomos skirtingomis, taip pat trumpesnis žodis eina prieš ilgesnį žodį, gautą iš to trumpesnio pirašant raidžių jo pabaigoje. Pavyzdžiui, lietuvių kalboje žodis „aš" eina prieš žodį „ašara". Sąraše pakanka informacijos abėcėlei nustatyti.

Prisiminkime, ką kalbėjome apie olimpiadinius uždavinius, kurių sąlygose minimas rikiavimas: dažniausiai jų sprendimui žinomų rikiavimo algoritmų tiesiogiai pritaikyti negalėsime. Taip bus ir šį kartą. Nors sąlygoje kalbama apie rikiavimą, tai iš tiesų yra topologinio rikiavimo uždavinys.

Sakykime, žinome, kuris iš dviejų žodžių, išrikiavus abėcėlės tvarka, eina pirmas. Pavyzdžiui:

ARKLYS
ARKTINIS

Ką galime sužinoti apie raidžių tvarką abėcėlėje? Pirmosios skirtingos žodžių raidės yra L ir T, tad į jas ir buvo atsižvelgta

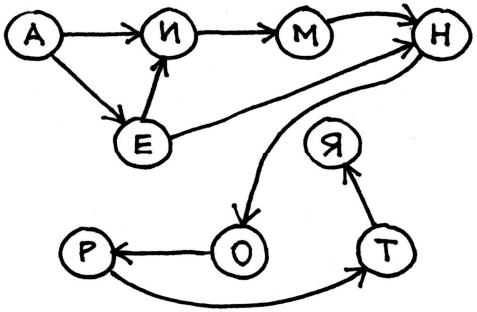
[36] Analogiškas uždavinys buvo pateiktas Lietuvos informatikos olimpiados III etape 2005 metais.

rikiuojant žodžius. Vadinasi, nežinomoje abėcėlėje raidė L eina prieš raidę T.

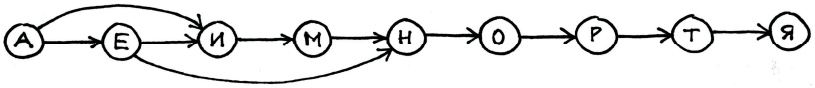
Raides žymėsime grafo viršūnėmis, o sąryšius tarp raidžių – lankais. Nustatę, kad raidė A abėcėlėje eina prieš raidę B, nuvesime lanką iš viršūnės A į viršūnę B. Gautasis grafas bus aibė reikalavimų, kuriuos turi tenkinti tos kalbos raidžių tvarka abėcėlėje. Abėcėlę, tenkinančią šiuos reikalavimus, rasime būtent topologiškai išrikiavę grafo viršūnes. Uždavinio sąlyga teigia, jog sąraše pakanka informacijos raidžių tvarkai nustatyti, taigi sudaryto grafo viršūnes bus įmanoma topologiškai išrikiuoti vieninteliu būdu.

Sudarinėjant grafą, pakanka išnagrinėti *tik gretimų* sąrašo žodžių poras: jei žinoma, kad raidė A eina prieš raidę B, o raidė B prieš raidę C, tai šias raides topologiškai išrikiavus raidė A būtinai eis prieš raidę C. Pavyzdžiui, sudarykime grafą iš tokio rusų kalbos žodžių, išrikiuotų abėcėlės tvarka, sąrašo:

ЕМ
ИМЯ
МАМА
МЕНЯ
МНЕ
МОНЕТА
НЕТ
НИНА
ОНА
ОНИ
РОТ
ТОТ
Я



62 pav. Grafas, atitinkantis kairėje pateiktą žodžių sąrašą



63 pav. Topologiškai išrikiavę raidžių grafą, randame nežinomos kalbos abėcėlę

Žemiau pateiktame sprendime grafo struktūroje žymėsime, kurias raides atitinkančios viršūnės yra grafe, nes ne visi simboliai įeina į abėcėlę. Procedūrai `rask_abėcėlę` perduodamas išrikiuotų pagal abėcėlę žodžių masyvas.

```
const MAXŽODŽIŲ = ...;
type žodžiai = array [1..MAXŽODŽIŲ + 1] of string;
     grafas = record
         n : integer; { viršūnių skaičius }
         viršūnė : array [char] of boolean;
                   { ar grafe yra raidę atitinkanti viršūnė }
         lankas : array [char, char] of boolean;
         įein_lankų : array [char] of integer;
     end;

procedure rask_abėcėlę(sk : integer; { žodžių skaičius }
                       var ž : žodžiai;
                       var abėcėlė : string);
                     { atsakymas įrašomas į eilutę abėcėlė }

    procedure sudaryk_grafą(var g : grafas);
    var i, j, m : integer;
        c, d : char;
    begin
        { išvalomi masyvai }
        g.n := 0;
        for c := low(char) to high(char) do begin
            g.viršūnė[c] := false;
            g.įein_lankų[c] := 0;
            for d := low(char) to high(char) do
                g.lankas[c, d] := false;
        end;
```

```
        { sudaromas grafas }
        ž[sk + 1] := ''; { pridedame tuščią žodį }
        for i := 1 to sk do begin
            { jei randama naujų raidžių – jos įtraukiamos į grafą }
            for j := 1 to length(ž[i]) do
                if not g.viršūnė[ž[i][j]] then begin
                    g.viršūnė[ž[i][j]] := true;
                    inc(g.n);
                end;
            m := min[37](
                length(ž[i]), length(ž[i + 1]));
            j := 1;
            while (j <= m) and (ž[i][j] = ž[i+1][j])
                do inc(j); { ieškoma nesutampanti raidė }
            if (j <= m) and
               not g.lankas[ž[i][j], ž[i+1][j]]
            then begin
            { rasta nesutampanti raidė – grafas papildomas lanku }
                g.lankas[ž[i][j], ž[i+1][j]] := true;
                inc(g.įein_lankų[ž[i + 1][j]]);
            end;
        end;
    end;

var g : grafas;
    c, d : char;
begin
    sudaryk_grafą(g);
    { topologiškai išrikiuojamas grafas (randama abėcėlė) }
    abėcėlė := '';
    while g.n > 0 do begin
        c := low(char);
        { randama viršūnė be įeinančių lankų }
        while (not g.viršūnė[c]) or
            (g.įein_lankų[c] > 0) do inc(c);
        { raidė pridedama prie abėcėlės }
        abėcėlė := abėcėlė + c;
```

[37] Funkcijos `min` teksto nepateikiame – ji paprasčiausiai grąžina mažesnįjį iš dviejų parametrų.

```
            { atnaujinami kaimynių laipsniai }
            for d := low(char) to high(char) do
                if g.lankas[c, d] then
                    dec(g.įein_lankų[d]);
            { viršūnė ištrinama iš grafo }
            g.viršūnė[c] := false;
            dec(g.n);
        end;
    end;
```

Atkreipiame dėmesį, kad šio uždavinio sprendime topologinis rikiavimas realizuotas kitaip – grafas vaizduojamas kaimynystės matrica, nenaudojama eilės duomenų struktūra. Algoritmo sudėtingumas – $O(n^2)$.

10 SVORINIAI GRAFAI, TRUMPIAUSIO KELIO PAIEŠKA – DIJKSTROS ALGORITMAS

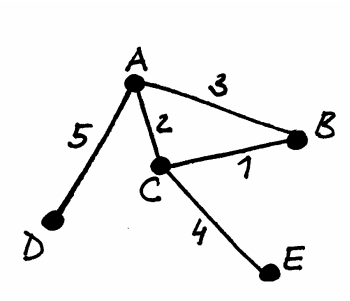
Most of fundamental ideas of science are essentially simple.
Dauguma fundamentalių mokslo idėjų yra iš esmės paprastos.
Albertas Enšteinas (Albert Einstein)

Dažnai tikslinga grafo briaunai (arba lankui) priskirti kokį nors dydį. Pavyzdžiui, jei grafu modeliuojame vietovės žemėlapį (viršūnėmis – miestus, o briaunomis – kelius), tai briaunoms galima priskirti tų kelių ilgius. Šiame skyrelyje aptarsime tokių grafų vaizdavimą ir vieną iš garsiausių algoritmų – Dijkstros algoritmą trumpiausiam keliui rasti.

10.1 Svoriniai grafai

Grafas, kurio visoms briaunoms (lankams) yra priskirti dydžiai (svoriai), vadinamas **svoriniu**. Dažniausiai nagrinėjami svoriniai grafai, kurių briaunų svoriai yra skaičiai.

Galime tarti, kad paprastas **besvoris** grafas tėra atskiras svorinio grafo atvejis, kai visų briaunų svoriai lygūs 1.



64 pav. Svorinio grafo pavyzdys

Duomenų struktūra, kuria galime pavaizduoti svorinį grafą, nesudėtinga. Galime naudoti kaimynystės matricą, kurioje saugosime briaunų svorius. Ta pati matrica turėtų saugoti ne tik briaunų svorius, bet ir parodyti, kurios grafo briaunos egzistuoja, kurios ne. Neegzistuojančias briaunas galime žymėti tokiu skaičiumi, kokio svorio būti negali, pavyzdžiui „begaliniu“

(labai dideliu) svoriu arba neigiamu skaičiumi (pavyzdžiui, jei grafo svoriai reiškia atstumus tarp miestų). Bet kuriuo atveju reikia būti tikram, kad jokioje algoritmo vykdymo stadijoje egzistuojančios briaunos svoris negalės įgauti tokios reikšmės.

```
const MAXN = ...; { maksimalus grafo viršūnių skaičius }
      BEGALINIS = MAXINT;

type grafas = record
         n : integer;       { viršūnių skaičius }
         svoris : array [1..MAXN,
                         1..MAXN] of integer;
                            { briaunų svorių matrica }
     end;
```

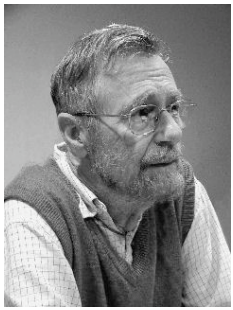
Šitaip vaizduojant grafą, viršūnes *u* ir *v* jungia briauna, jei `G.svoris[u, v] < BEGALINIS`.

Jei grafas vaizduojamas kaimynystės sąrašais, tai briaunos svorį tenka arba saugoti atskirame dvimačiame masyve, arba kiekvienai viršūnei sudaryti iš jos išeinančių briaunų svorių sąrašą.

10.2 Trumpiausio kelio paieškos algoritmas – Dijkstros algoritmas

Nagrinėjant olimpiadinių uždavinių sprendimus dažnai gali tekti susidurti su Dijkstros algoritmu trumpiausio kelio grafe paieškai. Šio algoritmo autorius – E. V. Dijkstra (Edsger Wybe Dijkstra) – olandų mokslininkas, daug nusipelnęs kompiuterių mokslui, ypač programavimo kalbų srityje. Trumpiausio kelio algoritmas nėra svarbiausias jo darbas, tačiau daugelis Dijkstros pavardę sieja būtent su šiuo algoritmu.

Pats E. V. Dijkstra apie tai rašo: „*Daug metų plačiuose sluoksniuose trumpiausio kelio algoritmas garsino mano vardą ir teikė šlovės, tačiau nuostabu tai, kad jis buvo sukurtas net be popieriaus ir pieštuko, geriant kavą su žmona saulėtoje Amsterdamo kavinės terasoje, sukurtas tik pademonstruoti kompiuterio galimybėms...*“



65 pav.
E. V. Dijsktra
(Edsger Wybe Dijkstra)
1930–2002

Jau esame aptarę vieną algoritmą, tinkamą trumpiausio kelio paieškai – paiešką platyn. Pradėta viršūnėje p, paieška platyn pirmiau ima viršūnes, kurių atstumas nuo viršūnės p (matuojamas briaunų, kuriomis einama, skaičiumi) yra mažiausias.

Nagrinėkime svorinį grafą G, kurio briaunos (u, v) svoris reiškia atstumą tarp viršūnių u ir v. **Kelio svoriniame grafe ilgiu** vadinsime visų kelią sudarančių briaunų svorių sumą. Nagrinėsime svorinį grafą G, kurio briaunos (u, v) **neneigiamas** svoris reiškia atstumą tarp viršūnių u ir v. Kaip ieškoti trumpiausio kelio tokiame grafe? Nesunku įsitikinti, kad paieška platyn čia visai netinkamas algoritmas, kadangi trumpiausias kelias nebūtinai reikš mažiausią briaunų, kuriomis einama, skaičių (pavyzdžiui, pasiekti viršūnę einant dviem briaunomis, kurių svoriai atitinkamai, 1 ir 2, yra „pigiau“ negu viena briauna, kurios svoris 5, nes $1 + 2 = 3 < 5$).

Dijkstros algoritmas, kaip ir paieška platyn, iš duotosios viršūnės p randa trumpiausius kelius iki **visų** svorinio grafo viršūnių. Algoritmas skirsto viršūnes į dvi aibes: tų, iki kurių trumpiausi keliai (ir atstumai) jau žinomi (jas vadinsime *prijungtomis*), ir visų kitų.

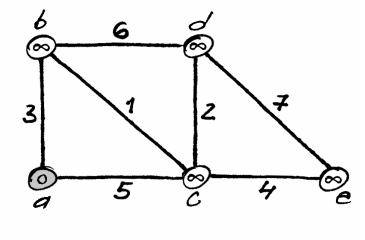
Pradžioje nežinomas trumpiausias kelias nė iki vienos viršūnės, išskyrus pradinę p, tad pažymima, kad atstumai iki šių viršūnių yra

begaliniai. Atstumas (nuo pradinės) iki pradinės viršūnės jau žinomas – jis lygus nuliui.

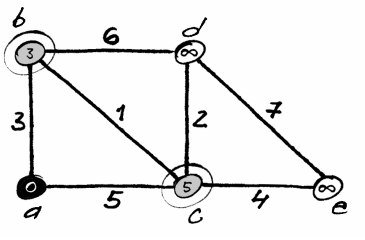
Kiekvienu žingsniu algoritmas suranda dar *neprijungtą* viršūnę, iki kurios atstumas yra mažiausias (pirmu algoritmo žingsniu tai pradinė viršūnė p, kadangi iki visų kitų viršūnių atstumai yra begaliniai). Pasirinktoji viršūnė prijungiama, o tuomet atnaujinama informacija apie visas *neprijungtas* jos kaimynes: galbūt kelias iki šios viršūnės dar nebuvo rastas, o jei buvo – tai galbūt kelias, einantis per ką tik prijungtąją viršūnę iki šios kaimynės, yra trumpesnis už iki šiol rastąjį.

Taigi pirmuoju algoritmo žingsniu prijungiama pradinė viršūnė p. Antruoju – artimiausia p kaimynė. Kiekvienu žingsniu prijungiamų viršūnių atstumai sudaro nemažėjančią seką, kadangi visąlaik bandoma prijungti kuo artimesnes viršūnes. Šie samprotavimai intuityviai pagrindžia algoritmo teisingumą. Prijungdami viršūnę, galime būti tikri, jog rastasis atstumas yra trumpiausias, kadangi visi kiti, vėliau atrasti, trumpiausi atstumai bus tik ilgesni už šį.

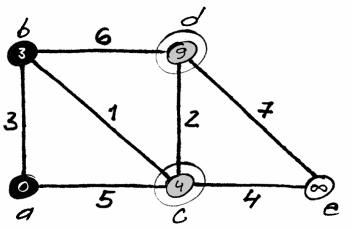
Kadangi ieškoma trumpiausių kelių, o ne tik jų ilgių, kiekvienai viršūnei išsaugoma jos pirminė viršūnė (tai viršūnė, iš kurios į ją ateinama einant trumpiausiu keliu). Kol kelias iki viršūnės nerastas, jos pirminė viršūnė yra neapibrėžta. Atnaujinant atstumą iki viršūnės, kartu pažymima, iš kurios viršūnės į ją ateinama. Algoritmo vykdymo metu kiekvienos viršūnės pirminė viršūnė (kaip ir trumpiausias rastas atstumas) gali ne kartą pasikeisti. 66 paveiksle pavaizduotas Dijkstros algoritmo vykdymas konkrečiame grafe, ieškomi trumpiausi keliai iš viršūnės a iki kitų grafo viršūnių.



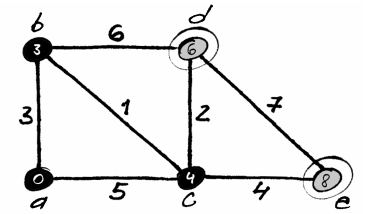
(1) Pradinė situacija: trumpiausio kelio iki viršūnės a (pasirinktosios pradinės viršūnės) ilgis lygus 0, o iki kitų viršūnių — nežinomas;



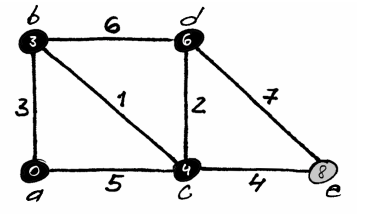
(2) Viršūnė a turi dvi kaimynes b ir c; iki šių viršūnių rasti trumpesni keliai



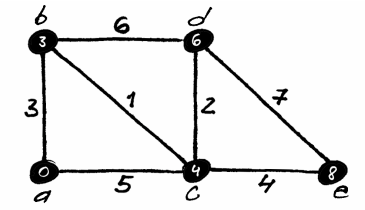
(3) Iš neprijungtų viršūnių išrenkama ta, iki kurios atstumas trumpiausias (viršūnę b); trumpesnio kelio iki b rasti negalima, ji prijungiama; peržiūrimos neprijungtos b kaimynės c ir d ir pastebima, kad iki šių abiejų viršūnių rasti trumpesni keliai per viršūnę b: iki viršūnės d kelias anksčiau nebuvo rastas, o iki viršūnės c buvo rastas tiesioginis kelias iš a; tačiau naujasis kelias per viršūnę b yra trumpesnis



(4)



(5)



(6) Baigus vykdyti Dijkstros algoritmą visos viršūnės yra prijungtos (t. y. visos yra pasiekiamos iš pradinės viršūnės) ir žinomi trumpiausi atstumai iki jų: trumpiausio kelio iki viršūnės b ilgis lygus 3, iki c — 4, iki d — 6, iki e — 8.

66 pav. Dijkstros algoritmo iliustracija

Toliau pateikiamas algoritmo tekstas, tinkamas trumpiausių kelių paieškai tiek orientuotame, tiek ir neorientuotame grafe. Grafas vaizduojamas kaimynystės matrica.

```
type masyvas = array [1..MAXN] of longint;
     logmas = array [1..MAXN] of boolean;

procedure dijkstra(var G : grafas;
                   var atstumas, pirminė : masyvas;
                   p : integer);

var prijungta : logmas;
    v, u : integer;
    min : longint;

begin
    { įrašomos pradinės masyvų reikšmės }
    for u := 1 to G.n do begin
        atstumas[u] := BEGALINIS;
        pirminė[u] := -1;
        prijungta[u] := false;
    end;
    atstumas[p] := 0;
```

```
    v := p;
    while v <> 0 do begin
        { jei v <> 0, tai rasta viršūnė, kurią galima prijungti }
        prijungta[v] := true;
        for u := 1 to G.n do { peržiūrimos kaimynės }
            if (G.svoris[v, u] < BEGALINIS) and
               (atstumas[u] >
                   atstumas[v] + G.svoris[v, u])
            then begin { į viršūnę u verčiau eiti per v }
                atstumas[u] :=
                    atstumas[v] + G.svoris[v, u];
                pirminė[u] := v;
            end;

        { randama tolesnė kandidatė -
          dar neprijungta viršūnė su mažiausiu atstumu }
        v := 0;
        min := BEGALINIS;
        for u := 1 to G.n do
            if not prijungta[u] and
               (atstumas[u] < min)
            then begin
                v := u;
                min := atstumas[u];
            end;
        { jei jokia viršūnė nerasta, tai v = 0 ir ciklas nutraukiamas }
    end;
end;
```

Užrašytojo algoritmo sudėtingumas yra $O(n^2)$, kur n – grafo viršūnių skaičius. Pasitelkus sudėtingesnes duomenų struktūras, Dijkstros algoritmą galima pagreitinti iki $O((n + b) \log n)$ (čia b – grafo briaunų skaičius). Pastarasis sudėtingumas yra kur kas geresnis **retuose** (turinčiuose nedaug briaunų) grafuose.

10.3 Uždavinys Aplink žemę per 80 dienų[38]

Žiulio Verno knygoje pasakojama, kaip Filijas Fogas apkeliavo aplink Žemę per 80 dienų. Tačiau galbūt sudarius labai gerą maršrutą, jam būtų pasisekę apkeliauti dar greičiau.

Žinomi įvairių transporto priemonių, vykstančių į rytus (Filijas Fogas keliavo tik į rytus), tvarkaraščiai, tie patys visomis dienomis. Apie kiekvieną reisą žinoma šitokia informacija: išvykimo miestas, išvykimo laikas, miestai, kuriuose sustojama, kelionės trukmė tarp dviejų gretimų stočių. Visi tvarkaraščiai nurodyti Grinvičo laiku.

Laikomasi susitarimo, kad tarpinėje stotyje transporto priemonės neužsibūna: atvyksta ir išvyksta tą pačią minutę, taip pat kad persėsti iš vienos transporto priemonės į kitą galima tą pačią minutę.

***Užduotis**. Žinomas miestas, iš kurio pradedama keliauti. Kelionės pradžia yra lygiai vidurnaktis Grinvičo laiku. Parašykite programą, kuria nustatytumėte, ar galima apkeliauti aplink Žemės rutulį pagal pateiktus susisiekimo priemonių tvarkaraščius ir, jei galima, informuotumėte, kada anksčiausiai įmanoma grįžti į miestą, iš kurio buvo išvykta.*

Kaip jau galėjote atspėti, uždavinys bus sprendžiamas taikant Dijkstros algoritmą. Tačiau olimpiada nėra kontrolinis darbas, kuriuo tikrinama, ar gerai dalyviai moka vieną ar kitą algoritmą. Tad

[38] Panašus uždavinys buvo pateiktas Lietuvos moksleivių informatikos olimpiadoje III etape 2000 metais.

ir uždaviniai olimpiadose pateikiami tokie, kad net žinant algoritmą, tenka jį modifikuoti ir pritaikyti neįprastai situacijai.

Sudarysime orientuotą grafą, kurio viršūnės atitiks miestus. Reikia rasti trumpiausią kelią iš pradinio miesto atgal į jį patį, tik trumpiausią laiko prasme. Tačiau Dijkstros algoritmas kiekvieną viršūnę nagrinėja tik po vieną kartą, todėl pradinį miestą (į kurį turime sugrįžti) pavaizduosime dviem viršūnėmis (M ir M'): viena turės tik išeinančius lankus, kita – tik įeinančius.

Galime būti tikri, kad bet kuri kelionė iš viršūnės M į viršūnę M' bus kelionė aplink pasaulį, kadangi visi maršrutai yra tik rytų krypties.

Į bet kurį maršrutą galima žiūrėti kaip į kelių tiesioginių (be persėdimų) ir nepriklausomų reisų rinkinį. Kiekvieną tokį (tiesioginį) reisą grafe atitiktų lankas, turintis du parametrus (svorius): reiso pradžios laiką ir jo trukmę. Kiekvienu Dijkstros algoritmo žingsniu būtų prijungiama viršūnė, iki kurios galime atvykti anksčiausiai. Prijungus viršūnę peržiūrimi visi iš jos išeinantys lankai. Pagal atvykimo į šią viršūnę laiką ir maršruto trukmę apskaičiuojama, kada galima nuvykti į kaimynines viršūnes.

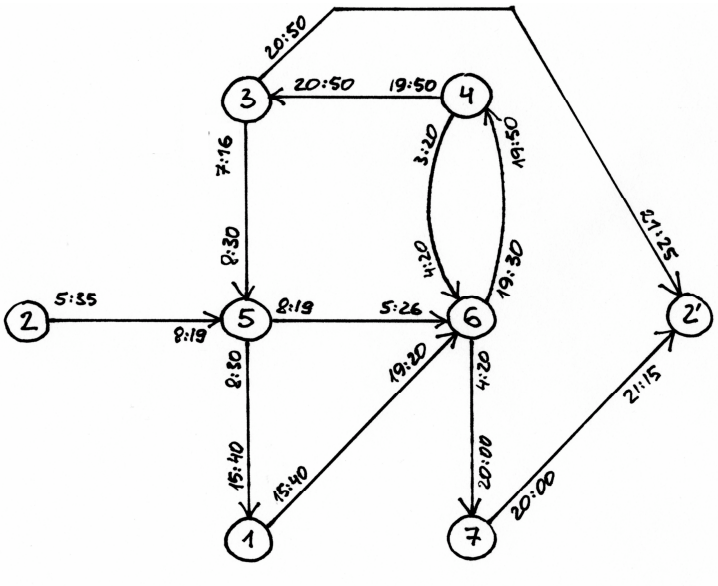
Panagrinėkime pavyzdį. Sakykime, duoti 7 miestai, Filijas Fogas kelionę pradeda ir baigia antrajame, ir galimi tokie maršrutai:

Pirmasis maršrutas: $2 \to 5 \to 6$, išvykimo laikas – 5:35, važiavimų trukmės: 2:44 ir 21:07.

Antrasis maršrutas: $3 \to 5 \to 1 \to 6$, išvykimo laikas – 7:16, važiavimų trukmės: 1:14, 7:10, 3:40.

Trečiasis maršrutas: $4 \to 6 \to 7 \to 2$, išvykimo laikas – 3:20, važiavimų trukmės: 1:00, 15:40, 1:15.

Ketvirtasis maršrutas: $6 \to 4 \to 3 \to 2$, išvykimo laikas – 19:30, važiavimų trukmės: 0:20, 1:00, 0:35.



67 pav. Pavyzdyje pateiktus maršrutus atitinkantis grafas; kad iliustracija būtų aiškesnė, vietoj važiavimo trukmių nurodyti atvykimo laikai (nė vienas reisas netrunka ilgiau nei parą)

Šiuos maršrutus atitinkantis grafas pateiktas 67 paveiksle. Tarkime, Filijas Fogas pradeda kelionę iš antrojo miesto. Jis anksčiausiai sugrįš namo, jei stotyje lauks iki ryto ir 5:35 išvyks į penktąjį miestą (tai, beje, vienintelis reisas iš antrojo miesto). Penktajame mieste jam verta persėsti ir važiuoti į pirmąjį miestą, o iš ten – į šeštąjį, kuriame jis atsiduris 19 val. 20 min. Ir spės į reisą, išvykstantį į ketvirtąjį miestą 19 val. 30 min. O iš ten be persėdimo važiuos iki pradinio miesto. Kelionės trukmė: 21 val. 25 min.

Jeigu Filijas Fogas penktajame mieste nepersėstų ir važiuotų toliau į šeštąjį miestą, tuomet jis ten atsidurtų kitos dienos ryte: 5 val. 26 min. ir pavėluotų į į rytinį reisą, vykstantį į septintą miestą. Jam tektų laukti iki vakaro ir tik 19 val. 30 min. jis galėtų išvykti į ketvirtąjį miestą. Kelionė aplink pasaulį truktų 1 parą, 21 val. ir 25 min., t. y. lygiai parą ilgiau nei optimaliu atveju.

Kadangi gali būti keli skirtingi reisai tarp tų pačių miestų, grafą būtina vaizduoti kaimynystės sąrašais. Sutarsime, kad skaitant pradinius duomenis, visi tarpinių sustojimų turintys maršrutai iš karto išskaidomi į persėdimų neturinčius reisus ir tuo pačiu sudaromas grafas. Taip pat sutarsime, kad, kuriant grafą, išvykimo laikai perskaičiuoti minutėmis. Rezultatas (laikas, kada anksčiausiai įmanoma grįžti) taip pat pateikiamas minutėmis nuo kelionės pradžios.

```
const BEGALINIS = MAXLONGINT;
      PARA = 24 * 60;
      MAXM = ...; { maksimalus miestų skaičius }
      MAXR = ...; { maksimalus reisų skaičius }

type masyvas = array [1..MAXM + 1] of longint;
     logmas = array [1..MAXM + 1] of boolean;
     reisas = record
         kur, kada, trukmė : longint;
     end;

     reisai_iš_miesto = record
         k : longint; { reisų skaičius }
         reisai : array [1..MAXR] of reisas;
     end;

     grafas = record
         n : longint; { miestų skaičius }
         mst : array [1..MAXM+1] of reisai_iš_miesto;
     end;
```

```
procedure dijkstra(var G : grafas;
                   pr : longint; { pradinis miestas }
                   var laikas : masyvas {atvykimo laikai});


var i, u, v, t, min, atvykta, išvyksta : longint;
    prijungta : logmas;
begin
    { įrašomos pradinės masyvų reikšmės }
    for u := 1 to G.n do begin
        laikas[u] := BEGALINIS;
        prijungta[u] := false;
    end;
    laikas[pr] := 0;


    v := pr;
    while v <> 0 do begin
        { prijungiama viršūnė v }
        prijungta[v] := true;
        { atnaujinama informacija apie kaimynes }
        for i := 1 to G.mst[v].k do begin
            u := G.mst[v].reisai[i].kur;
            t := G.mst[v].reisai[i].trukmė;
            { kiek reikės laukti mieste v ? }
            atvykta := laikas[v] mod PARA;
            išvyksta := G.mst[v].reisai[i].kada;
            if atvykta <= išvyksta then
                { reisu pavyks išvykti tą pačią parą }
                t := t + (išvyksta - atvykta)
            else { teks laukti kitos dienos }
                t := t + (PARA - atvykta) + išvyksta;
            { ar į u verta vykti per v? }
            if laikas[u] > laikas[v] + t then
                laikas[u] := laikas[v] + t;
        end;
```

```
        { randama tolesnė kandidatė –
        dar neprijungta viršūnė su mažiausiu atstumu }
        v := 0;
        min := BEGALINIS;
        for u := 1 to G.n do
            if not prijungta[u] and (laikas[u] < min)
            then begin
                v := u;
                min := laikas[u];
            end;
    end;
end;



procedure keliauk(var G : grafas; { informacija apie visus
                                    reisus iš kiekvieno miesto}
                  pr : longint;   { pradinis miestas}
                  var atvykimas : longint { sprendinys});
var i, j, pb : longint;
    laikas : masyvas;

begin
    { pradinis miestas keičiams dviem miestais: miestu, kuriame
    kelionė prasidėjo ir fiktyviu, kuriame kelionė baigėsi }
    G.n := G.n + 1;
    pb := G.n;
    for i := 1 to G.n - 1 do
        for j := 1 to g.mst[i].k do
            if G.mst[i].reisai[j].kur = pr then
                G.mst[i].reisai[j].kur := pb;

    { suskaičiuojama, per kokį mažiausią laiką galima
      nuvykti į kiekvieną miestą }
    dijkstra(G, pr, laikas);

    atvykimas := laikas[pb];
    { jei maršruto nėra, atvykimas = BEGALINIS }
end;
```

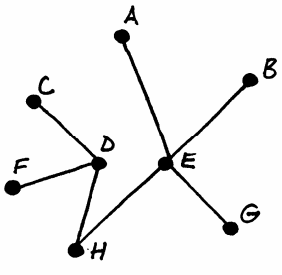
11 MEDŽIAI, MINIMALAUS JUNGIAMOJO MEDŽIO RADIMAS

I hope to convince you that mathematical trees are no less lovely than their biological counterparts.
Tikiuosi, galiausiai jūs sutiksite, jog matematiniai medžiai džiugina širdį ne mažiau nei tikrieji.
Džo Malkevičius (Joe Malkevitch)

Ankstesniuose skyriuose išplėtėme grafo sąvoką: susipažinome su orientuotais bei svoriniais grafais. Šį kartą susiaurinsime grafo sąvoką. Panagrinėsime medžius – grafus, pasižyminčius tam tikromis savybėmis. Pamatysime, jog medžiai dažnai pasitaiko, kai grafais modeliuojami praktiniai uždaviniai.

11.1 Medžiai

Medžiu vadinamas neorientuotas jungus ciklų neturintis grafas.



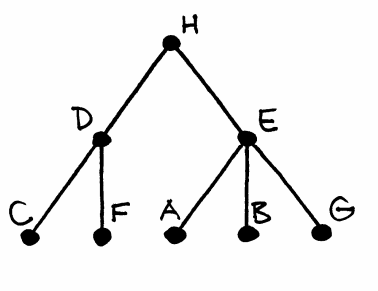
68 pav. Medis

Kiekvienas medis pasižymi tokiomis savybėmis:

- bet kurias dvi viršūnes medyje jungia vienintelis kelias;
- visos medžio briaunos yra tiltai (t. y. panaikinus bet kurią briauną medis taptų nejungus);
- n viršūnių turintis medis visuomet turi $(n-1)$ briauną, ir atvirkščiai – jungusis grafas turintis n viršūnių ir $(n-1)$ briauną yra medis.

Kai kada medžiuose viena viršūnė išskiriama iš kitų ir pavadinama **medžio šaknimi**, o pats medis – **šakniniu medžiu**.

Šakninio medžio viršūnės taip pat turi skirtingus pavadinimus. Bet kuri viršūnė *u*, esanti tiesioginiame kelyje tarp šakninės viršūnės ir viršūnės *v,* vadinama viršūnės *v* **protėviu**, o viršūnė *v* vadinama viršūnės *u* **vaikaičiu**. Šakninė viršūnė yra visų medžio viršūnių protėvis, o visos medžio viršūnės yra jos vaikaičiai, kiekviena viršūnė yra savo pačios protėvis ir vaikaitis. Jei viršūnė *u* yra *v* protėvis ir egzistuoja briauna, jungianti *u* ir *v*, tuomet *u* vadinama **pirmine (tėvine)** *v* viršūne, o *v* vadinama **antrine (vaikine)** *u* viršūne. Medžio viršūnė, neturinti antrinių viršūnių, vadinama **lapu**.



69 pav. Šakninis medis, gautas iš praeitame paveiksle pateikto medžio, šaknine pasirinkus viršūnę H; medis turi penkis lapus (C, F, A, B, G); šakninė viršūnė H turi dvi antrines viršūnes (D ir E) ir 7 vaikaičius (D, E, C, F, A, B, G).

11.2 Medžių vaizdavimas

Medis yra susiaurinta grafo sąvoka: bet kuris medis yra grafas, bet ne atvirkščiai. Tad medžius galima vaizduoti lygiai tomis pačiomis duomenų struktūromis kaip ir grafus: kaimynystės matricomis ir

kaimynystės sąrašais. Kita vertus, galime tikėtis rasti elegantiškesnį būdą medžiams pavaizduoti. Juk iš anksto žinome, jog toks grafas turės $(n-1)$ briauną, bus jungus bei neturės ciklų.

Iš tiesų, jei medis šakninis, tai kiekviena viršūnė, išskyrus medžio šaknį, turi lygiai vieną pirminę viršūnę. Todėl medį visiškai apibrėžia jau anksčiau minėtas pirminumo masyvas:

```
const MAX = ...; { maksimalus medžio viršūnių skaičius }
type masyvas = array [1..MAX] of integer;

var pirminė : masyvas; { kiekvienai viršūnei įsimenama jos
                         pirminė viršūnė }
```

Taip pavaizdavę medį, galime sužinoti visas medžiui priklausančias briaunas (jos yra pavidalo `(k, pirminė[k])`) ir efektyviai rasti kelius nuo viršūnės *v* iki šakninės viršūnės, pereinant visus viršūnės *v* protėvius. Tačiau „leistis“ medžiu, t. y. ieškoti kiekvienos viršūnės antrinių viršūnių efektyviai negalime, nes tektų peržiūrėti visą masyvą.

Siekdami efektyviai rasti tiek pirminę, tiek ir antrines viršūnes, šakninį medį galime vaizduoti įrašų masyvu, kuriame saugoma kiekvienos viršūnės pirminė viršūnė ir antrinių viršūnių sąrašas:

```
type viršūnė = record
          pirminė : integer;
          antr_sk : integer; { antrinių viršūnių skaičius }
          antr_sąr : array [1..MAX] of integer
     end;
     medis = array [1..MAX] of viršūnė;
```

Toks vaizdavimas neefektyvus atminties požiūriu: nors visų viršūnių sąrašų `antr_sąr` ilgių suma bus lygi $(n-1)$, šiems masyvams skiriama $O(n^2)$ atminties, nes iš anksto nežinoma, kiek kuri viršūnė turės antrinių. Šią problemą galima spręsti naudojant dinaminę

atmintį, kuomet atmintis išskiriama tik tada, kai jos prireikia, ir kiekvienam sąrašui išskirti tik tiek atminties, kiek būtina. Tačiau dinaminės duomenų struktūros yra gana sudėtingos, jų realizavimas ir derinimas atima nemažai laiko, todėl olimpiadose geriau jų vengti.

Kokį vaizdavimą pasirinkti? Tai visuomet priklauso nuo sprendžiamo uždavinio. Dažnai pakanka medį saugoti pirminumo masyvu. Kai norima efektyviai ieškoti antrinių viršūnių, medį tenka vaizduoti antruoju būdu, jei tik viršūnių skaičius nėra per didelis. Be to, kai kuriuose uždaviniuose nagrinėjami specifiniai medžiai, pavyzdžiui, kurių kiekviena viršūnė turi ne daugiau kaip dvi antrines viršūnes (dvejetainiai medžiai). Jiems nesunku pritaikyti įrašo tipo struktūrą.

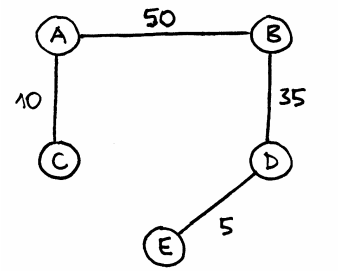
11.3 Minimalus jungiamasis medis

Panagrinėsime optimizavimo uždavinį, su kuriuo dažnai susiduriama praktikoje. Tarkime, kad tiesiamos elektros linijos tiekti elektrai į N miestelių. Šiuo tikslu visus N miestelių reikia sujungti į vieną elektros tinklą. Yra apskaičiuota linijos nutiesimo tarp bet kurių dviejų miestelių kaina, ir norima sudaryti tokį elektros linijų planą, kad visų linijų tiesimo kainų suma būtų kuo mažesnė. Be abejo, nutiesus linijas, kiekvienas miestelis turi turėti elektrą.

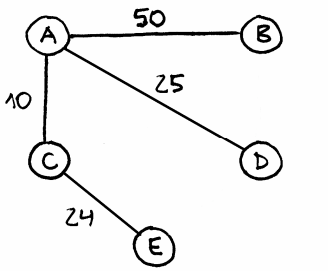
Panagrinėkime pavyzdį. Tarkime, kad miestelių yra penki, o elektros linijų tiesimo tarp miestelių porų kainos yra tokios:

	A	B	C	D	E
A	-	50	10	25	10
B	50	-	20	35	40
C	10	20	-	15	24
D	25	35	15	-	5
E	10	40	24	5	-

Paveiksluose pateikiami keli elektros linijų tiesimo planai.



(1) Visi penki miesteliai sujungti į tinklą; tokio sujungimo kaina – 100



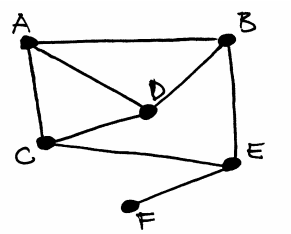
(2) Tie patys miesteliai sujungti į tinklą kitu būdu; šio sujungimo kaina – 109

70 pav. Du galimi miestelių sujungimo į tinklą būdai

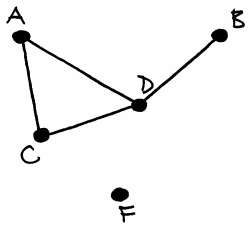
Matyti, kad yra ne vienas būdas sujungti miestelius į tinklą, ir vieni šių būdų gali būti ekonomiškesni už kitus.

Turbūt jau supratote, jog šį uždavinį nesunku formaliai apibrėžti grafų teorijos terminais. Tačiau prieš tai įvesime dar kelias sąvokas.

Grafo G **pografiu** vadinamas grafas G', kurį papildžius viršūnėmis ir (arba) briaunomis, gaunamas grafas G. Pografis G' negali turėti briaunos arba viršūnės, kurios neturi grafas G.



(1)

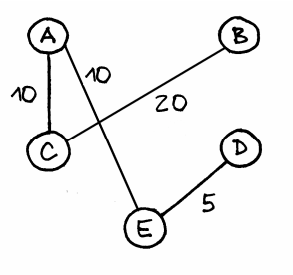


(2)

71 pav. Grafas ir vienas jo pografių

Grafo G pografis, kuriam priklauso visos G viršūnės ir kuris yra medis, vadinamas grafo G **jungiamuoju medžiu**. Nesunku suvokti, kad vienas grafas gali turėti daugiau nei vieną jungiamąjį medį. Tačiau jei grafas nejungus, jis neturi jungiamojo medžio.

Dabar žinome viską, ko reikia nagrinėjamam uždaviniui formalizuoti. Jei kiekvieną miestelį atitinka grafo G viršūnė, o elektros linijos tiesimo iš miestelio A į miestelį B kainą žymi briaunos (A, B) svoris, tai ieškomasis linijų tiesimo planas yra grafo G jungiamasis medis, kurio briaunų svorių suma mažiausia. Toks medis vadinamas **minimaliu jungiamuoju medžiu** (MJM), o pats uždavinys – minimalaus jungiamojo medžio uždaviniu.



72 pav. Grafo, sudaryto iš skyrelio pradžioje nagrinėto pavyzdžio, minimalus jungiamasis medis; sujungimo kaina – 45

Kitame skyrelyje panagrinėsime efektyvius algoritmus minimalaus jungiamojo medžio paieškai.

11.4 Primo ir kiti algoritmai MJM rasti

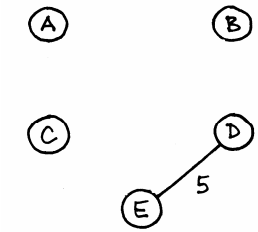
Knygose ir mokslinėje literatūroje ilgą laiką buvo rašoma, kad pirmieji MJM ieškančius algoritmus sukūrė Džozefas Bernardas

Kruskalas (*Joseph Bernard Kruskal*) ir Robertas Klėjus Primas (*Robert Clay Prim*) apie 1956–1957 metus. Šie algoritmai vėliau buvo pavadinti jų vardais. Deja, liko nepastebėta, kad labai gražų ir elegantišką algoritmą MJM paieškai net dvidešimčia metų anksčiau jau siūlė čekų mokslininkas Otakaras Boruvka (*Otakar Borůvka*). Galbūt šio mokslininko darbas buvo nepastebėtas todėl, kad straipsnį jis išspausdino čekų kalba. Dar daugiau – pasirodo, Primo algoritmas taip pat buvo atrastas anksčiau kito čekų matematiko Vojtecho Jarniko (*Vojtěch Jarník*), o algoritmui jau buvo prigijęs Primo algoritmo vardas.

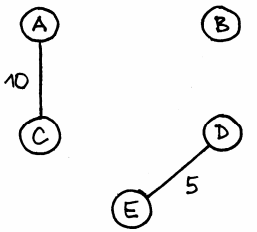
73 pav. Džozefas Bernardas Kruskalas (Joseph Bernard Kruskal)

Šiame skyrelyje aprašysime visus tris algoritmus MJM paieškai, tačiau pateiksime tik Primo algoritmo realizaciją. Tam yra rimta priežastis – Primo algoritmo MJM paieškai realizacija skiriasi nuo Dijkstros trumpiausio kelio algoritmo vos keliomis eilutėmis.

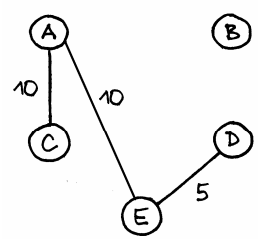
Visi trys algoritmai remiasi **godžiąja strategija**, t.y. kiekviename žingsnyje pasirenkamas palankiausias tuo momentu sprendimas. Ko gero, aiškiausias yra **Kruskalo algoritmas**, kuriuo konstruojamas MJM prijungiant grafo briaunas. Iš pradžių medis yra tuščias, o kiekvienu tolesniu žingsniu prijungiama pigiausia (mažiausio svorio) briauna, kurios prijungimas nesudarytų ciklo. Medis baigiamas konstruoti, kai daugiau negalima prijungti nė vienos briaunos. Kadangi medis turi lygiai $(n-1)$ briauną, tai MJM sudaryti prireikia lygiai $(n-1)$ žingsnių (n – grafo viršūnių skaičius).



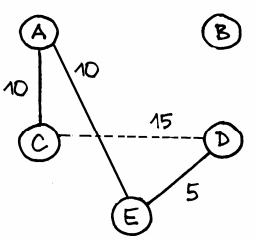
(1) Pirmasis žingsnis: randama pigiausia briauna (jos kaina – 5) ir įtraukiama į MJM



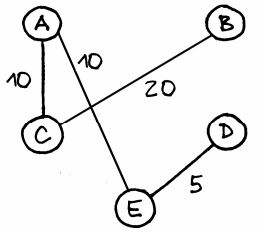
(2) Pasirenkama kita pigiausia briauna (yra dvi tokios briaunos AC ir AE, imama bet kuri) ir įtraukiama į MJM



(3) Kita pigiausia briauną yra AE; ji įtraukiama į MJM



(4) Tolesnė pigiausia briauna yra CD (jos kaina 15), tačiau jos įtraukti į MJM negalima, nes susidarytų ciklas, tad ši briauna praleidžiama



(5) Prijungiama ketvirtoji pigiausia briauna (BC, jos kaina 20) ir gaunamas MJM; jo kaina – 45

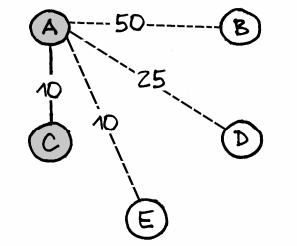
74 pav. Kruskalo algoritmo veikimo iliustracija

Nors Kruskalo algoritmą suprasti labai lengva, jį realizuoti sudėtingiau, nes nuolat tenka tikrinti, ar prijungiant briauną nesusidarys ciklas.

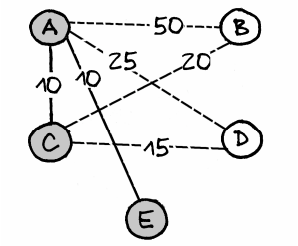
Primo algoritmu taip pat MJM konstruojamas prijungiant grafo briaunas, tačiau pradedama nuo medžio, kurį sudaro viena laisvai pasirinkta viršūnė. Prijungiamoji briauna taip pat turi būti pigiausia, tačiau tenkinti kitokią sąlygą negu Kruskalo algoritme: lygiai viena briaunos viršūnė turi priklausyti konstruojamam medžiui. Ši sąlyga garantuoja, kad prijungiant briauną nesusidarys ciklas.

75 pav. Robertas Klėjus Primas (Robert Clay Prim)

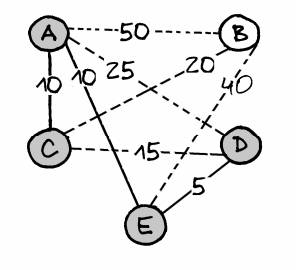
Toliau iliustruojama, kaip veikia Primo algoritmas. Prijungtos viršūnės spalvinamos pilkai, ir iliustracijose pateikiamos tik tos briaunos, kurios yra arba jau prijungtos prie MJM, arba kurių lygiai viena viršūnė priklauso MJM.



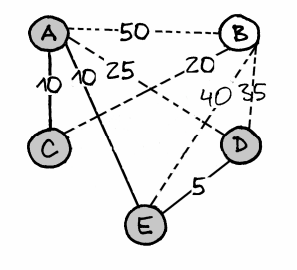
(1) Pasirenkame pradinę viršūnę (pavyzdžiui, A); matome, kad pigiausiai prie jos galime prijungti viršūnes C arba E; pasirenkame bet kurią – C



(2) Prie sudarinėjamo MJM, kuris kol kas turi dvi viršūnes A, C ir briauną tarp jų, pigiausiai galime prijungti viršūnę E (briaunos AE svoris 10)



(3) Toliau pigiausiai galima prijungti viršūnę D (briaunos svoris 5)



(4) Liko viena neprijungta viršūnė; ją pigiausiai galima prijungti briauna CB, jos svoris – 20; gauname 72 pav. pavaizduotą MJM

76 pav. Primo algoritmo veikimo iliustracija

Kaip jau minėjome, Primo algoritmo realizacija labai primena Dijkstros algoritmą. Pradedant nuo tuščio medžio, kiekvienu žingsniu išsirenkama ir prijungiama nauja viršūnė. Todėl, kaip ir Dijkstros algoritme, visos viršūnės paskirstomos į dvi aibes: prijungtų prie konstruojamo medžio ir dar neprijungtų. Kiekvienu žingsniu norėsime prie medžio prijungti tą viršūnę, kurią galima prijungti pigiausia briauna. Todėl Primo algoritmas išlaiko mažiausią žinomą kiekvienos viršūnės prijungimo kainą. Pradžioje šios kainos nustatomos begalinės visoms viršūnėms, išskyrus pasirinktąją. Kiekvienu žingsniu prijungus viršūnę su mažiausia prijungimo kaina, galbūt bus rastas geresnis būdas, kaip prie medžio prijungti jos kaimynes. Todėl peržiūrimos ir, jei reikia, atnaujinamos prijungtosios viršūnės kaimynių prijungimo kainos. Atliekamų žingsnių skaičius lygus grafo viršūnių skaičiui.

Toliau pateiktame algoritme grafas vaizduojamas kaimynystės matrica, o minimalus jungiamasis medis – pirminumo masyvu.

```
const BEGALINIS = MAXINT;
      MAXN = ...; { maksimalus viršūnių skaičius }
type grafas = record
         n : longint; { viršūnių skaičius }
         svoris : array [1..MAXN,
                         1..MAXN] of integer;
     end;
     masyvas = array [1..MAXN] of integer;
     logmas  = array [1..MAXN] of boolean;

procedure Primo(var G : grafas;
                var pirminė : masyvas);
{ ieškomasis medis grąžinamas masyve „pirminė“ }

var prijungta : logmas;
    kaina : masyvas;
    v, u, min : integer;

begin
    { įrašomos pradinės masyvų reikšmės }
    for u := 1 to G.n do begin
        kaina[u] := BEGALINIS;
        pirminė[u] := -1;
        prijungta[u] := false;
    end;
    v := 1;
    kaina[v] := 0; { pradėsime nuo pirmos viršūnės }
    while v <> 0 do begin
        { jei v <> 0, tai rasta viršūnė, kurią galima prijungti }
        prijungta[v] := true;
        for u := 1 to G.n do { nagrinėjamos kaimynės }
            if (not prijungta[u]) and
               (G.svoris[v, u] < BEGALINIS) and
               (kaina[u] > G.svoris[v, u])
            then begin { viršūnę u verčiau jungti prie v }
                kaina[u] := G.svoris[v, u];
                pirminė[u] := v;
            end;
```

```
            { randama tolesnė kandidatė –
            dar neprijungta viršūnė su mažiausia prijungimo kaina }
            v := 0;
            min := BEGALINIS;
            for u := 1 to G.n do
               if (not prijungta[u]) and (kaina[u] < min)
               then begin
                    v := u;
                    min := kaina[u];
               end;
             { jei jokia viršūnė nerasta, tai v = 0 ir ciklas nutraukiamas }
      end;
   end;
```

Įvykdžius algoritmą, minimaliam jungiamajam medžiui priklauso briaunos `(v, pirminė[v])`, kur v – bet kuri grafo viršūnė, išskyrus pradinę. Primo algoritmo sudėtingumas – $O(n^2)$.

Aprašysime ir nepelnytai pamirštą, tačiau ne mažiau elegantišką nei Primo ar Kruskalo algoritmai, **Boruvkos algoritmą**.

Algoritmas operuoja medžių sąrašu. Pradžioje šį sąrašą sudaro N medžių, kurių kiekvieną sudaro viena (kiekvienam kita) grafo viršūnė. Tuomet paeiliui nagrinėjami visi medžiai. Kiekvienam jų randama pigiausia į medį ateinanti, tačiau medžiui nepriklausanti briauna, ir įtraukiama į jį. Jei keliems medžiams buvo parinkta ta pati pigiausia briauna, tai tie medžiai sujungiami. Veiksmai kartojami tol, kol lieka tik vienas medis. Tai ir bus minimalus jungiamasis medis.

11.5 Uždavinys *Tinklas*[39]

Firma ALFA gavo užsakymą: sujungti k kompiuterių ir m komutatorių[40] *į vieną laidinį tinklą. Reikalavimai tinklo architektūrai tokie:*

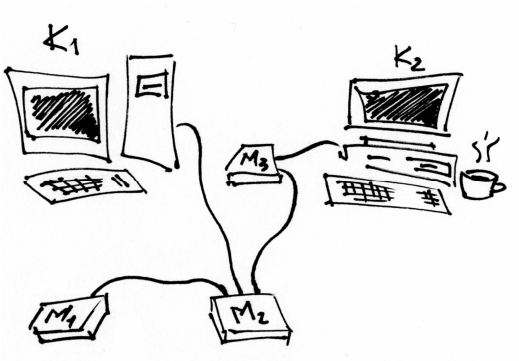
- *Kiekvienas kompiuteris tiesiogiai vienu laidu sujungiamas su bet kuriuo vienu (ir tik vienu) komutatoriumi;*
- *Prie kiekvieno komutatoriaus tiesiogiai laidais galima prijungti bet kokį skaičių kitų įrenginių (kompiuterių arba komutatorių); du įrenginiai tiesiogiai sujungiami vienu laidu;*
- *Visi m komutatorių ir k kompiuterių turi sudaryti jungų tinklą, t. y. bet kuris įrenginys turi būti tiesiogiai arba netiesiogiai (per kitus įrenginius) sujungtas su visais kitais;*

Užduotis. *Duotos kompiuterių ir komutatorių sujungimo kainos. Reikia rasti tokią tinklo jungimų schemą, kurios kaina būtų mažiausia.*

---

[39] Panašus uždavinys buvo pateiktas Lietuvos informatikos olimpiadoje III etape 2005 metais.

[40] Komutatorius – įtaisas, skirtas sujungti į bendrą tinklą du ar daugiau kitų įrenginių ar tinklų.



77pav. Galima dviejų kompiuterių ir trijų komutatorių jungimo į tinklą schema

Kiekvienas kompiuteris turi būti prijungtas tik prie vieno įrenginio, būtent, komutatoriaus. Kadangi kompiuterį galime prijungti prie bet kurio iš jų, tai išsirinksime tą komutatorių, prie kurio prijungti kompiuterį yra pigiausia.

Tačiau visi įrenginiai turi sudaryti jungų tinklą, todėl komutatoriai turės būti sujungti tarpusavyje. Žinomos kiekvieno galimo jungimo kainos, todėl šiam jungimui rasti galime pritaikyti bet kurį minimalaus jungiamojo medžio paieškos algoritmą.

Pateiktame programos tekste visi įrenginiai sunumeruoti nuosekliai: komutatoriai nuo 1 iki m, o kompiuteriai – nuo $(m + 1)$ iki $k + m$. Procedūrai perduodamas užpildytas įrenginių jungimo kainų masyvas, taip pat įrenginių skaičius (k ir m). Grafas vaizduojamas briaunų svorių matrica (žr. 10.1 skyrelį).

```
const BEGALINIS = MAXINT;
      MAXM = ...; { maksimalus komutatorių skaičius }
      MAXK = ...; { maksimalus kompiuterių skaičius }

type masyvas = array [1..MAXM] of integer;
     jungimas = record
         įrenginysA, įrenginysB : integer;
     end;
     jungimų_mas =
         array [1..MAXM + MAXK] of jungimas;
     kainų_mas = array [1..MAXM + MAXK,
                        1..MAXM + MAXK] of integer;

procedure rask_jungimus(var kaina : kainų_mas;
                        m, k : integer;
                        var jung_sk,
                            jung_kaina : integer;
                        var jungimai : jungimų_mas);
{ k – kompiuterių, m – komutatorių skaičius, „kaina“ – įrenginių jungimo
kainų masyvas; atsakymas pateikiamas masyve „jungimai“ }
    procedure junk(a, b : integer);
    { įrenginys a sujungiamas su įrenginiu b }
    begin
        jung_sk := jung_sk + 1;
        jungimai[jung_sk].įrenginysA := a;
        jungimai[jung_sk].įrenginysB := b;
        jung_kaina := jung_kaina + kaina[a, b];
    end;

var i, j, t : integer;
    g : grafas;
    pirminė : masyvas;

begin
    jung_sk := 0; jung_kaina := 0;
    { prijungiame kiekvieną kompiuterį prie „artimiausio“
    komutatoriaus (kompiuteriai sunumeruoti nuo (m + 1)
    iki (m + k), komutatoriai - nuo 1 iki m) }
    for i := m + 1 to m + k do begin
        t := 1;
        for j := 1 to m do
            if kaina[i, t] > kaina[i, j] then t := j;
        junk(i, t);
    end;
```

```
        { komutatorių jungimui sudarome grafą ir randame
          minimalų jungiamąjį medį }
        g.n := m;
        for i := 1 to m do
            for j := 1 to m do
                if i <> j then
                    g.svoris[i, j] := kaina[i, j]
                else { jei i = j, tai briaunos (kilpos) nėra }
                    g.svoris[i, j] := BEGALINIS;
        { pagal Primo algoritmą randamas MJM }
        Primo(g, pirminė);

        { medžio briaunos yra (i, pirminė[i]), visoms i, išskyrus 1 }
        for i := 2 to g.n do
            junk(i, pirminė[i]);
    end;
```

12 DINAMINIS PROGRAMAVIMAS

Computer science is no more about computers
than astronomy is about telescopes.
Kompiuterių mokslą vadinti mokslu apie kompiuterius būtų tas pats,
kas vadinti astronomiją mokslu apie teleskopus.
Edsgeras V. Dijkstra (Edsger W. Dijkstra)

Apie dinaminį programavimą būtų galima pasakyti panašiai kaip ir apie kompiuterių mokslą: dinaminis programavimas neturi jokio tiesioginio ryšio su programavimu. Matematikai šį žodį vartoja nusakyti taisyklėms ir principams, kurių laikantis sprendžiamas uždavinys.

Dinaminis programavimas yra efektyvus sprendinių radimo būdas, kurį galima pritaikyti kai kuriems, ypač optimizavimo, uždaviniams spręsti.

Šį metodą pasiūlęs ir 1952 metais aprašęs amerikiečių mokslininkas Ričardas Belmanas savo autobiografijoje pasakoja, iš kur kilo pavadinimas *Dinaminis programavimas:*

78 pav. Ričardas Belmanas (Richard Bellman), 1920–1984

1950-ieji metai buvo ne itin palankūs matematiniams tyrinėjimams. Tuo metu Vašingtone dirbo labai įdomus ponas, pavarde Vilsonas. Jis buvo gynybos sekretorius ir patologiškai bijojo to, kas vadinama moksliniais tyrinėjimais. <...> Jo veidas parausdavo ir jis įtūždavo, jei kas nors jam girdint pavartodavo šią sąvoką. Galite tik įsivaizduoti, kaip jį veikė žodis „matematika“. Tuomet aš dirbau RAND korporacijoje, kurią samdė Oro pajėgos, o pastarosios buvo pavaldžios ir Vilsonui. Taigi kažkokiu būdu reikėjo nuslėpti, kad užsiimu matematiniais tyrinėjimais. Kokį pavadinimą galėjau pasirinkti? Mane domino planavimas ir sprendimų priėmimas, tačiau „planavimas“ nebuvo tinkamas žodis,

tad pasirinkau „programavimą“. Žodis „dinaminis“ atspindėjo daugiapakopiškumą, buvo būdvardis ir turėjo labai tikslią reikšmę fizikine prasme. <...> Kita vertus, šis žodis jokiame kontekste neįgaudavo menkinančios reikšmės. Tad pasirinkau pavadinimą, kuriam net Kongreso narys negalėjo prieštarauti ir tai buvo priedanga mano tolesniems darbams.

Uždavinių, sprendžiamų dinaminiu programavimu, dažnai pasitaiko olimpiadose. Šiame skyriuje apžvelgsime dinaminio programavimo principus. Iš pirmo žvilgsnio gali pasirodyti nelengva suprasti dinaminį programavimą, kadangi tai nėra algoritmas, o veikiau uždavinio sprendimo schema. Todėl daug dėmesio skirsime iliustravimui, kaip taikyti šį metodą sprendžiant konkrečius uždavinius.

12.1 Optimizavimo uždaviniai

Optimizavimo uždaviniu vadiname uždavinį, kai yra daug galimų sprendinių, kurių kiekvieną galima kaip nors įvertinti, o ieškoma sprendinio, turinčio tam tikrą (optimalią) vertę. Štai klasikinio optimizavimo uždavinio, vadinamo *Kuprinės uždaviniu*, pavyzdys:

Vagis, naktį įsilaužęs į muziejų, rado n vertingų eksponatų. Žinoma kiekvieno eksponato vertė v_k bei svoris s_k (sveikasis skaičius). Vagis gali panešti kuprinę, sveriančią ne daugiau kaip S kilogramų. Kuriuos eksponatus jis turėtų susikrauti į kuprinę, kad bendra jų vertė būtų kuo didesnė, o kuprinė – panešama?

Tai optimizavimo uždavinys, nes yra daug būdų sudaryti eksponatų rinkinį, kurį galėtų panešti vagis, ir kiekvienas jų turi tam tikrą vertę, o ieškoma rinkinio, kurio vertė *maksimali*.

Svarbu skirti sąvokas **sprendinys** ir **sprendinio vertė**. Mūsų pavyzdyje sprendinio vertė yra pasirinktų eksponatų verčių suma, o sprendinys yra pats eksponatų rinkinys. Optimizavimo uždaviniuose dažniausiai nesunku rasti bet kurį sprendinį (pavyzdžiui, bet kurį eksponatų rinkinį, kurį galėtų panešti vagis). Kur kas sunkiau rasti optimalų sprendinį (tokį panešamą eksponatų rinkinį, kurio vertė maksimali).

Optimizavimo uždavinių sprendimui galima taikyti **godųjį algoritmą**, kiekviename algoritmo žingsnyje pasirenkant geriausią variantą *toje situacijoje* (pavyzdžiui, rinktis eksponatus, kurių vertės ir svorio santykis kuo didesnis). Godieji algoritmai dažniausiai yra efektyvūs. Tačiau kiekviename žingsnyje renkantis lokalųjį optimumą, nebūtinai gaunamas globalusis optimumas. Reikia įsitikinti, kad godusis algoritmas tikrai ras geriausią sprendinį.

11.3 skyriuje nagrinėta *Minimalaus jungiamojo medžio paieška* taip pat yra optimizavimo uždavinys, o jį sprendžiantys Primo bei Kruskalo algoritmai – godieji algoritmai.

Galima pamanyti, kad ir *Kuprinės uždaviniui* spręsti tiktų godusis algoritmas: išrikiuoti eksponatus jų vertės ir svorio santykio (t. y. svorio vieneto vertės) mažėjimo tvarka, ir paeiliui, kol neviršijamas svoris s, rinkti kuo vertingesnius eksponatus iš šio sąrašo. Deja, toks sprendimas ne visuomet randa optimalų sprendinį. Pateiksime kontrpavyzdį. Tegu $s = 50$, o eksponatų svoriai bei vertės pateikti lentelėje:

Nr.	*Svoris*	*Vertė*	*Svorio vieneto vertė*
1	10	60	6
2	20	100	5
3	30	120	4

Taikant godųjį algoritmą, būtų pasirenkami pirmasis ir antrasis eksponatai, kurių bendra vertė yra 160. Trečio eksponato nebepavyktų paimti, nes visi trys jie netilptų į kuprinę. Tuo tarpu pasirinkus antrąjį ir trečiąjį eksponatus, gaunama vertė 220.

Optimizavimo uždavinius galima spręsti **perrinkimu**: perrinkti visus įmanomus sprendinius ir iš jų išrinkti optimalų. Nors šiuo būdu galima rasti optimalų sprendinį, deja, perrinkimas praktiškai netaikomas, nes yra labai neefektyvus, t. y. nepolinominio sudėtingumo.

Dinaminis programavimas turi abiejų metodų gerąsias savybes: viena vertus, kiekviename žingsnyje pasirenkamas geriausias variantas (gaunamas efektyvus algoritmas), kita vertus – peržiūrimi visi galimi pasirinkimai, galintys vesti prie optimalaus sprendinio (randamas optimalus sprendinys).

Dinaminiu programavimu galima spręsti tuos optimizavimo uždavinius, kuriuose optimalų sprendinį pavyksta rekursyviai išreikšti per analogiškų, bet mažesnių optimizavimo uždavinių sprendinius.

12.2 Dinaminio programavimo principai

Uždavinio sprendimas dinaminiu programavimu susideda iš keturių žingsnių:

1. Nustatoma optimalaus sprendinio struktūra.
2. Rekursyviai apibrėžiama sprendinio vertė.
3. Apskaičiuojama optimalaus sprendinio vertė (skaičiuojant ją įsimenamos smulkesnių uždavinių sprendinių vertės, kurios panaudojamos ieškomai optimalaus sprendinio vertei rasti).
4. Sukonstruojamas pats optimalus sprendinys.

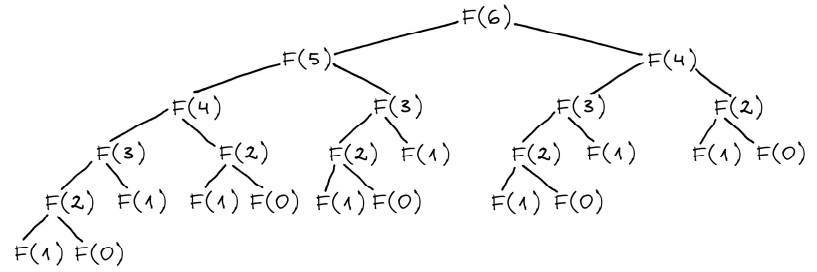
Jeigu reikia rasti ne patį optimalų sprendinį, o tik jo vertę, tuomet paskutinis žingsnis praleidžiamas.

Panagrinėsime labai paprastą uždavinį, sprendžiamą dinaminiu programavimu ir kartu padėsiantį suvokti, kodėl dinaminiu programavimu pagrįsti algoritmai yra efektyvūs. Prisiminkime Fibonačį, triušius ir jo skaičius, apie kuriuos buvo rašyta 4.1 skyrelyje. Fibonačio skaičiai apibrėžiami tokiu būdu: $F_0 = 0$, $F_1 = 1$, $F_n = F_{n-1} + F_{n-2}$, reikia rasti *n*-ąjį Fibonačio skaičių F_n.

Tai nėra optimizavimo uždavinys. Sprendinio vertė jau apibrėžta rekursyviai, tereikia ją suskaičiuoti. 4.1 skyrelyje buvo pateikta rekursinė funkcija, skaičiuojanti Fibonačio skaičius:

```
function F(n : longint) : longint;
begin
    if n = 0 then
        F := 0
    else if n <= 2 then
        F := 1
    else
        F(n - 1) + F(n - 2);
end;
```

4.1 skyrelyje rašėme, kad rekursyvus Fibonačio skaičių skaičiavimas yra eksponentinio sudėtingumo (labai lėtas). Pažvelkime į žemiau pateiktą kreipinio (į rekursinę funkciją) `F(6)` skaičiavimų medį.



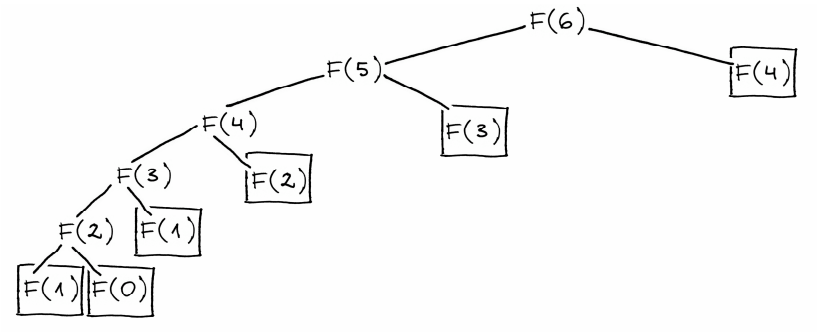
79 pav. Kreipinio F(6) į rekursinę funkciją skaičiavimų medis

Nesunku pastebėti, kad skaičiuojant F_6 darbo atliekama kur kas daugiau negu reikia. Tos pačios mažesnių Fibonačio skaičių reikšmės perskaičiuojamos daug kartų. Pavyzdžiui, skaičiuojant F_6, net 5 kartus tenka skaičiuoti F_2. Nesunku įsivaizduoti, kaip atrodytų F_7 paieškos medis: reikėtų atlikti beveik dvigubai daugiau darbo.

Taigi, būtų natūralu kartą suskaičiuotą reikšmę įsiminti masyve, ir jos daugiau neperskaičiuoti:

```
const MAX = ...;
var Fmas : array [0..MAX] of longint;

function F(n : longint) : longint;
begin
    { dar neapskaičiuotos reikšmės žymimos -1 }
    if Fmas[n] <> -1 then
        F := Fmas[n]
    else if n = 0 then
        F := 0
    else if n = 1 then
        F := 1
    else begin
        Fmas[n] := F(n - 1) + F(n - 2);
        F := Fmas[n];
    end;
end;
```

Toliau pateiktas šios funkcijos rekursijos medis. Kvadratėliais įrėmintos reikšmės iš naujo nebeskaičiuojamos, o paimamos iš lentelės.



80 pav. Funkcijos F, įsimenančios tarpinius sprendinius, skaičiavimų medis, kai į ją kreiptasi F(6)

Kiekviena reikšmė bus skaičiuojama tik vieną kartą, todėl F_n suskaičiuojamas per $O(n)$ laiko. Tačiau dar paprasčiau yra apsieiti be rekursijos ir suskaičiuoti F_n generuojant Fibonačio skaičių seką iš eilės, kiekvieną narį gaunant iš dviejų paskutinių:

```
var Fmas : array [0..MAX] of longint;

function F(n : longint) : longint;
var k : integer;
begin
    Fmas[0] := 0;
    Fmas[1] := 1;
    for k := 2 to n do
        Fmas[k] := Fmas[k - 1] + Fmas[k - 2];
    F := Fmas[n];
end;
```

Šioje funkcijoje F_n reikšmė skaičiuojama **iš apačios į viršų**, t. y. pradedant nuo pačių mažiausių reikšmių ir vis gaunant didesnes. Prieš tai aprašytos funkcijos reikšmes skaičiavo **iš viršaus į apačią**.

Tai, ką atlikome, buvo trečiasis dinaminio programavimo metodo žingsnis: rekursyvus apibrėžimas padėjo sukonstruoti *efektyvų*

algoritmą. Efektyvumą (laiko atžvilgiu) pasiekėme atmetę pakartotinį tų pačių uždavinių sprendimą, įsimindami jų vertes (taigi atminties efektyvumo sąskaita[41]).Tai būdinga visiems dinaminiu programavimu pagrįstiems algoritmams.

Jau buvo minėta, kad dinaminio programavimo išmokstama ne skaitant teoriją, o analizuojant sprendimus, tad tolesniuose skyreliuose ir analizuosime uždavinius, sprendžiamus dinaminio programavimo metodu.

Pradėsime nuo uždavinio, su kurio sąlyga jau esame susipažinę.

12.3 *Kuprinės uždavinys*

Vagis, naktį įsilaužęs į muziejų, rado n vertingų eksponatų. Žinoma kiekvieno eksponato vertė v_k bei svoris s_k, išreikšti sveikaisiais skaičiais. Vagis gali panešti kuprinę, sveriančią ne daugiau kaip S kilogramų.

***Užduotis**. Reikia nustatyti, kuriuos eksponatus jis turėtų susikrauti į kuprinę, kad bendra jų vertė būtų kuo didesnė, o kuprinė – panešama.*

Tai klasikinis optimizavimo uždavinys, sprendžiamas optimizuojant (pavyzdžiui, minimizuojant arba maksimizuojant) pasirinkto svorio *S* kuprinės vertę.

[41] Jei reikalinga tik optimalaus sprendinio vertė, tai galima sudaryti efektyvesnį atminties atžvilgiu algoritmą. Pavyzdžiui, skaičiuojant Fibonačio skaičius iš apačios į viršų, nereikia saugoti atmintyje viso masyvo – pakanka įsiminti du paskutinius suskaičiuotus narius.

Uždavinį būtų galima spręsti perrinkimu – išbandyti visus įmanomus rinkinius – tačiau tai, be abejo, labai neefektyvu. Pastebėkime, kad nors galimų rinkinių labai daug (2^n), tačiau galimų rinkinių svorių pakankamai nedaug – nuo 0 iki $s_1 + s_2 + ... + s_n$. Pasinaudosime šia savybe ir sudarysime efektyvų, dinaminiu programavimu pagrįstą algoritmą.

Dinaminio programavimo taikymas prasideda nuo optimalaus sprendinio struktūros nustatymo. Sunumeruokime eksponatus nuo 1 iki n ir pagalvokime, kokią didžiausią vertę galima pasiekti neviršijant svorio S. n-asis eksponatas gali priklausyti arba nepriklausyti optimaliam sprendiniui:

- jei n-asis eksponatas nepriklauso optimaliam sprendiniui, tai optimalus sprendinys lygus kito, mažesnio uždavinio – optimalaus rinkinio iš pirmųjų $(n - 1)$ eksponatų, neviršijančio svorio S – sprendiniui;

- jei n-asis eksponatas priklauso optimaliam sprendiniui, tai optimalų sprendinį sudaro n-asis eksponatas ir kito, mažesnio, uždavinio – optimalaus rinkinio iš pirmųjų $(n - 1)$ eksponatų, neviršijančio svorio $(S - s_n)$ – sprendinys.

Tai ir yra optimalaus kuprinės uždavinio sprendinio struktūra. Optimalų sprendinį gausime išnagrinėję abu variantus ir išsirinkę didesnės vertės sprendinį.

Pažymėkime $D(k, r)$ didžiausią rinkinio, kurio svoris neviršija r ir kuris sudarytas iš pirmųjų k eksponatų, vertę. Tuomet, remdamiesi ankstesniais samprotavimais, $D(k, r)$ galime išreikšti rekursyviai:

$$D(k, r) = \begin{cases} 0, & jei\ k = 0 \\ D(k-1, r), & jei\ \ s_n > r \\ \max\{D(k-1, r),\ \ v_k + D(k-1, r-s_k)\}, & kitais\ atvejais \end{cases}$$

Ši formulė jau leidžia apskaičiuoti optimalaus sprendinio vertę D(n, S), tačiau efektyviai galime skaičiuoti tik įsimindami dalinių sprendinių vertes (kaip ir Fibonačio skaičių atveju).

Panagrinėkime konkretų pavyzdį. Sakykime, vagis gali panešti 12 kilogramų. Eksponatų svoriai bei vertės pateikti lentelėje:

Eksponatas	Vertė	Svoris
1	1	1
2	5	2
3	8	3
4	11	4
5	20	7

Paruošiame funkcijos D reikšmių lentelę, užpildydami iš anksto žinomas kraštines reikšmes: maksimali vertė visada lygi nuliui, kai nėra nė vieno eksponato ($D(0, S) = 0$), arba kai vagis negali panešti jokio svorio ($D(n, 0) = 0$).

Skaičiuojant $D(n, S)$ reikšmę pagal rekurentinį sąryšį, naudojamos funkcijos reikšmės su mažesniais parametrais (t. y. analogiškų uždavinių su mažesniais parametrais optimalūs sprendiniai). Todėl jei lentelę pildysime po eilutę, pradėdami nuo $k = 0$, o eilutėje – iš kairės į dešinę, pradėdami nuo $r = 0$, tai skaičiuodami konkrečią reikšmę ($D(k, r)$) tikrai būsime jau anksčiau apskaičiavę kitas reikalingas reikšmes ($D(k-1, r)$ ir $D(k-1, r-s_n)$).

Pavyzdžiui, apskaičiuokime langelio $D(3, 5)$ reikšmę, t. y. raskime, kokia gali būti didžiausia kuprinėje esančių eksponatų vertė, jei galime rinktis iš trijų pirmųjų eksponatų, o kuprinės svoris negali viršyti 5 kg.

		Svoris (r)												
		0	1	2	3	4	5	6	7	8	9	10	11	12
Eksponatų sk. (k)	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	1	0	1	1	1	1	1	1	1	1	1	1	1	1
	2	0	1	5	6	6	6	6	6	6	6	6	6	6
	3	0	1	5	8	9	**13**							
	4	0												
	5	0												

Galimi du variantai: arba įtraukti į rinkinį trečiąjį eksponatą, arba ne. Pirmuoju atveju gausime vertę $v_3 + D(2, 5 - s_3) = 8 + D(2, 2) = 8 + 5 = 13$, o antruoju – $D(2, 5) = 6$ (skaičiavimams reikalingos reikšmės lentelėje pažymėtos pilku fonu). Renkamės didesniąją iš šių verčių – 13, trečiąjį eksponatą įtraukdami į optimalų rinkinį.

Taigi reikšmių lentelės užpildymą realizuoti nesudėtinga. Programoje einamąją eilutę (eksponatų kiekį) žymėsime raide k, einamąjį stulpelį (nagrinėjamą svorį) – r, o eksponatų svorius ir vertes saugosime masyvuose *svoris* ir *vertė*. Skaičiuodami konkretaus langelio $[k, r]$ reikšmę, iš pradžių patikriname, ar k-ojo eksponato svoris neviršija nagrinėjamo svorio, t. y. ar $svoris[k] \le r$. Jei viršija – tai $D[k, r] = D[k - 1, r]$, t. y. k-ojo eksponato į rinkinį įtraukti negalime. Priešingu atveju, $D[k, r]$ priskiriame didesnę iš reikšmių $D[k - 1, r]$ ir $(vertė[k] + D[k - 1, r - svoris[k]])$.

```
const MAXN = ...; { maksimalus eksponatų skaičius }
      MAXS = ...; { maksimalus panešamas svoris }
type lentelė = array [0..MAXN, 0..MAXS] of integer;
     masyvas = array [1..MAXN] of integer;

procedure skaičiuok(n, S : integer;
                    var svoris, vertė : masyvas;
                    var D : lentelė);
var k, r : integer;
begin
    { užpildomos kraštinės lentelės reikšmės }
    for r := 0 to S do
        D[0, r] := 0;
    for k := 0 to n do
        D[k, 0] := 0;
    { užpildoma visa likusi lentelės dalis }
    for r := 1 to S do
        for k := 1 to n do
            if svoris[k] <= r then
                { jei k-asis eksponatas tilptų }
                D[k, r] := max[42](
                    D[k - 1, r],
                    vertė[k] +
                        D[k - 1, r - svoris[k]])
            else
                { jei k-asis eksponatas netilptų,
                  jo įtraukti negalima }
                D[k, r] := D[k - 1, r];
end;
```

Štai kaip atrodo iki galo užpildyta nagrinėto pavyzdžio lentelė Pusjuodžiu šriftu pažymėtos reikšmės, gaunamos įtraukiant atitinkamą eksponatą į rinkinį.

[42] Funkcija `max` randa didesnįjį iš dviejų skaičių, jos teksto nepateikiame.

Eksponatų sk. (*k*)	Svoris (*r*)												
	0	1	2	3	4	5	6	7	8	9	10	11	12
0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**
2	0	1	**5**	**6**	**6**	**6**	**6**	**6**	**6**	**6**	**6**	**6**	**6**
3	0	1	5	**8**	**9**	**13**	**14**	**14**	**14**	**14**	**14**	**14**	**14**
4	0	1	5	8	**11**	13	**16**	**19**	**20**	**24**	**25**	**25**	**25**
5	0	1	5	8	11	13	16	**20**	**21**	**25**	**28**	**31**	**33**

Taigi įvykdėme jau tris iš keturių dinaminio programavimo metodo žingsnių: nustatę optimalią sprendinio struktūrą, išreiškėme jo reikšmę rekursyviai ir sudarėme efektyvų algoritmą, kuris, įsimindamas tarpinius sprendinius, apskaičiuoja šią reikšmę. Duoto pavyzdžio atveju maksimali vertė lygi 33. Tačiau vagį, be abejo, domina ne tik vertė, bet ir pats eksponatų rinkinys, kuris sudarytų tokią vertę. Rinkinį nesudėtinga sukonstruoti iš jau suskaičiuotos lentelės. Pradėkime nuo langelio $[n, S]$: jei $D[n, S] = D[n-1, S]$, tai n-ojo eksponato į rinkinį įtraukti nereikia ($D[n, S]$ buvo gautas iš $D[n-1, S]$, taigi neįtraukiant n-ojo eksponato), o jei $D[n, S] > D[n-1, S]$ – įtraukti reikia. Toliau atitinkamai nagrinėjame langelius $[n-1, S]$ arba $[n-1, S - svoris[n]]$, ir taip toliau, kol pasiekiame lentelės kraštą.

```
type logmas = array [1..MAXN] of boolean;

procedure sudaryk_rinkinį(n, S : integer;
                          var svoris : masyvas;
                          var D : lentelė;
                          var imti : logmas);
{ pagal masyvų „D“ ir „svoris“ reikšmes nustatoma,
  kuriuos eksponatus verta imti }
var k, r : integer;
begin
    for k := 1 to n do
        imti[k] := false;
```

```
        k := n;
        r := S;
        while (k > 0) and (r > 0) do begin
            if D[k, r] > D[k - 1, r] then begin
                { vadinasi, vertė D[k, r] gauta įtraukus k-ąjį eksponatą }
                imti[k] := true;
                r := r - svoris[k];
            end;
            k := k - 1;
        end;
    end;
```

Šią procedūrą reikia kviesti įvykdžius procedūrą `skaičiuok`. Nesudėtinga įvertinti algoritmo sudėtingumą: pildant $n \times S$ dydžio lentelę, kiekvienam langeliui sugaištama O(1) laiko, taigi algoritmo sudėtingumas ir atminties ir laiko atžvilgiu yra $O(n \cdot S)$. Beje, jei pats rinkinys nedomina, tai sudėtingumą atminties atžvilgiu galima sumažinti iki $O(S)$, kadangi skaičiuojant konkrečią reikšmę pakanka prisiminti tik einamąją ir prieš tai buvusią lentelės eilutes. Tačiau neapsigaukite: iš tiesų algoritmo sudėtingumas yra polinominis tik jei iš anksto žinoma, jog dydis S pakankamai nedidelis. Bendru atveju (jei S neapribotas), *Kuprinės uždavinys* yra NP sunkus uždavinys.

12.4 Uždavinys *Ilgiausias didėjantis posekis*

Duota n skaičių seka a_1, a_2, ..., a_n.

Užduotis. *Reikia rasti ilgiausią didėjantį šios sekos posekį.*

Pavyzdžiui, jei duota seka (9, 5, 2, 8, 7, 3, 1, 6, 7, 4, 6, 3), tai ilgiausias didėjantis posekis turi keturis narius. Galimi sprendiniai (2, 3, 6, 7) arba (2, 3, 4, 6).

Pradėsime nuo optimalaus sprendinio struktūros nustatymo. Tai pavyks padaryti pradėjus analizuoti seką nuo pabaigos — tokia strategija dažnai pasiteisina (prisiminkime, jog *Kuprinės uždavinyje* ieškodami optimalaus sprendinio struktūros, eksponatus taip pat pradėjome analizuoti nuo paskutinio).

Tarkime, kad paskutinis sekos narys (skaičius a_n) užbaigia ilgiausią didėjantį posekį. Koks gi sekos narys posekyje eina prieš a_n? Tegu tai a_k ($k < n$). Be abejo, tam, kad posekis būtų didėjantis, a_k turi būti mažesnis už a_n. Be to, a_k turi būti toks sekos narys, kad savo ruožtu sekoje $a_1, a_2, \ldots, a_k$ užbaigtų kuo ilgesnį didėjantį posekį.

Pasitelkę tokius samprotavimus, uždavinio sprendinį išreiškėme mažesnių uždavinių sprendiniais: jei visiems $k = 1, 2, \ldots, n-1$, kuriems $a_k < a_n$, žinotume, koks ilgiausio sekos $a_1, a_2, \ldots, a_k$ posekio, užsibaigiančio nariu a_k, ilgis, tai iš šių posekių išrinkę ilgiausią ir prijungę a_n, tikrai gautume ilgiausią didėjantį posekį, užsibaigiantį nariu a_n (kadangi būtume išbandę visus variantus).

Jei kiekvienam sekos nariui suskaičiuotume, kokį ilgiausią didėjantį posekį šis užbaigia, tai iš jų išrinkę ilgiausią ir gautume ilgiausią didėjantį visos sekos posekį.

Pažymėję ilgiausio posekio, užsibaigiančio nariu a_n, ilgį $L(n)$, ankstesnius samprotavimus galime užrašyti tokia lygybe:

$$L(n) = 1 + \max_{\substack{1 \le k < n \\ a_k < a_n}} L(k)$$

Rekursinis optimalios sprendinio vertės apibrėžimas yra antrasis dinaminio programavimo metodo žingsnis. Pagal šią formulę sudarysime efektyvų algoritmą.

Toliau pateikiama procedūra rasti optimaliam sprendiniui *iš apačios į viršų*. Iš pradžių randama, kokį ilgiausią posekį užbaigia sekos narys a_1, tuomet a_2, ir taip toliau iki a_n. Iš šių išrenkamas ilgiausias visos sekos posekis. Atskirame masyve p saugoma informacija, kuri vėliau padės sukonstruoti optimalų sprendinį: $p[k]$ rodo ilgiausio sekos $a_1, a_2, \ldots, a_k$ posekio, užsibaigiančio nariu a_k, priešpaskutinio nario numerį.

```
const MAX = ...; { maksimalus sekos ilgis }
type masyvas = array [1..MAX] of integer;

procedure ilg_posekis(a : masyvas; n : integer;
                      var posekis : masyvas;
                      var ilgis : integer);
var L, p : masyvas;
    k, kmax, m, nr : integer;
begin
    { optimalaus sprendinio vertė skaičiuojama iš apačios į viršų }
    kmax := 1; { ilgiausio posekio paskutiniojo elemento indeksas }
    for m := 1 to n do begin
        L[m] := 0;
        for k := 1 to m - 1 do
            if (a[k] < a[m]) and (L[k] > L[m])
            then begin
                L[m] := L[k];
                {pažymimas priešpaskutinis šio posekio elementas}
                p[m] := k;
            end;
        { priskaičiuojamas ir m-asis elementas }
        L[m] := L[m] + 1;
        if L[kmax] < L[m] then
            { tai ilgiausias kol kas rastas posekis }
            kmax := m;
    end;

    { sukonstruojamas optimalus sprendinys }
    ilgis := L[kmax];
    for k := ilgis downto 1 do begin
        posekis[k] := a[kmax];
        kmax := p[kmax];
    end;
end;
```

Šio sprendimo sudėtingumas laiko atžvilgiu yra $O(n^2)$, o atminties atžvilgiu – $O(n)$.

Parodysime, kaip randamas ilgiausias sąlygoje pateiktos sekos (*9, 5, 2, 8, 7, 3, 1, 6, 7, 4, 6, 3*) posekis.

k	1	2	3	4	5	6	7	8	9	10	11	12
a_k	9	5	2	8	7	3	1	6	7	4	6	3
$L(k)$	1	1	1	2	2	2	1	3	4	3	4	2
p_k	–	–	–	2	2	3	–	6	8	6	10	3

Kaip minėta pavyzdyje, yra du ilgiausi didėjantys posekiai, kurių ilgis 4 – eilutėje $L(k)$ skaičius 4 įrašytas dviejuose langeliuose. Pasinaudojus masyvo p reikšmėmis nesunku sukonstruoti patį posekį. Pavyzdžiui, konstruosime posekį, užsibaigiantį a_9. Paskutinis posekio narys yra $a_9 = 7$, priešpaskutinio posekio nario numeris lygus $p_9 = 8$, tad šis narys lygus $a_8 = 6$. Prieš jį eina šeštas ($p_8 = 6$) sekos narys $a_6 = 3$, o prieš šį trečias – $a_3 = 2$. Taigi ilgiausias didėjantis nagrinėtos sekos posekis yra (2, 3, 6, 7).

12.5 Uždavinys *Teisingos dalybos*[43]

Dvi draugės – Rusnė ir Emilija – nori pasidalyti n dovanų rinkinį. Kiekviena dovana turi būti atiduota arba Rusnei, arba Emilijai, ir nė viena dovana negali būti padalyta į dvi dalis. Kiekviena dovana turi vertę, išreikštą sveikuoju skaičiumi nuo 0 iki m. Pažymėkime R ir E dovanų, kurias atitinkamai gaus Rusnė ir Emilija, verčių sumas.

[43] Panašus uždavinys buvo pateiktas Vidurio Europos informatikos olimpiadoje, kuri vyko Vengrijoje 1995 m.

Užduotis. *Reikia rasti, kaip padalyti dovanas Rusnei ir Emilijai, kad* $|R - E|$ *būtų minimalus.*

Dovanų vertes pažymėkime v_1, v_2, ..., v_n. Bendra šių dovanų vertė lygi $V = v_1 + v_2 + ... + v_n$. Atkreipkite dėmesį, kad $R + E = V$. Taigi, žinodami vieną iš šių skaičių, galime iš karto apskaičiuoti ir antrą. Taip pat žinant, kurios dovanos bus atiduotos Rusnei, vienareikšmiškai galima pasakyti, kurios atiteks Emilijai. Taigi galima spręsti „pusę“ uždavinio: ieškoti, kaip parinkti dovanas Rusnei, kad jų verčių suma būtų kuo artimesnė $V/2$.

Šį kartą dinaminį programavimą taikysime netiesiogiai: iš pradžių dinaminiu programavimu išspręsime kitą uždavinį, vadinamą *sumos dėstymu,* o pasinaudoję jo sprendimu, nesunkiai padalysime dovanas mergaitėms teisingiausiu įmanomu būdu.

Tarkime, duota n sveikųjų skaičių $v_1, v_2, ..., v_n$ iš intervalo $[0, m]$. Prašoma nustatyti, ar (ir kaip) iš jų galima sudaryti tokį skaičių rinkinį, kad jų suma būtų lygi A. Jei taip, tai sakysime, kad iš skaičių $v_1, v_2, ..., v_n$ galime *sudėti* skaičių A. Šis uždavinys vadinamas **sumos dėstymu**.

Nesunku pastebėti, kad išsprendę sumos dėstymo uždavinį, mokėsime išspręsti ir *Teisingų dalybų* uždavinį: iš dovanų verčių paeiliui bandysime sudėti skaičius, kuo artimesnius $V/2$, ir sustosime, kai tik pavyks.

Galimų rinkinių yra labai daug – 2^n, jų visų išbandyti negalima. Kita vertus, sumų, kurias gali sudaryti kuris nors duotųjų skaičių rinkinys, yra palyginti nedaug – tai skaičiai nuo 0 iki V, kur $V = v_1 + v_2 + ... + v_n$, taigi jų ne daugiau negu $n \cdot m + 1$.

Pasinaudoję šia savybe, sudarysime dinaminiu programavimu pagrįstą algoritmą.

Tarkime, kad iš duotųjų n skaičių galima sudėti skaičių A. Skaičius v_n gali priklausyti šiam rinkiniui arba nepriklausyti (kitų variantų nėra):

- jei skaičius v_n rinkiniui nepriklauso, tai skaičių A turi būti įmanoma sudėti iš pirmųjų $(n-1)$ skaičių;

- jei v_n rinkiniui priklauso, tai iš pirmųjų $(n-1)$ skaičių turi būti įmanoma sudėti likusią skaičiaus dalį $(A - v_n)$.

Taigi abiem atvejais uždavinį galima išreikšti per analogiškų, tačiau su mažesniais parametrais, uždavinių sprendimus. Jei teiginį „skaičių S galima sudėti iš pirmųjų k skaičių“ pažymėsime $G(k, S)$, tai būtų teisingos tokios lygybės[44]:

$$G(k, S) = \begin{cases} true & jei\ S = 0 \\ false & jei\ k = 0 \\ G(k-1, S), & jei\ S < v_k \\ G(k-1, S) \vee G(k-1, S - v_k), & kitais\ atvejais \end{cases}$$

Remiantis šiomis lygybėmis nesunku sudaryti efektyvų *Sumos dėstymo uždavinio* algoritmą – apskaičiuoti funkcijos G reikšmes *iš apačios į viršų*, pildant $n \times A$ dydžio reikšmių lentelę, pradedant nuo mažiausių k (taip, kaip darėme spręsdami *Kuprinės uždavinį*).

[44] Simbolis „$\vee$“ reiškia loginę operaciją „arba“; Paskalio kalboje tai atitiktų loginę operaciją „or“.

Pavyzdžiui, jei duotieji skaičiai yra $v_1 = 3$, $v_2 = 4$, $v_3 = 5$, $v_4 = 7$ ir klausiama, ar iš jų galima sudėti skaičių $A = 14$, tai reikšmių lentelė, gauta iš rekurentinių lygybių, būtų tokia (pažymėtos tik teigiamos funkcijos reikšmės):

		S														
		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
k	0	t														
	1	t			t											
	2	t			t	t			t							
	3	t			t	t	t		t	t	t			t		
	4	t			t	t	t		t	t	t	t	t	t		t

Pildant lentelės langelį $[k, S]$, peržiūrimi langeliai $[k-1, S]$ ir $[k-1, S-v_k]$: jei bent vienam iš jų įrašyta reikšmė *true*, tai į $[k, S]$ taip pat įrašoma *true*:

```
const MAXN = ...; { maksimalus dėmenų skaičius }
      MAXM = ...; { maksimali dėmens vertė }

type masyvas = array [1..MAXN] of integer;
     logmas2 = array [0..MAXN * MAXM,
                      0..MAXN] of boolean;

procedure dėstyk(var v : masyvas; n, A : integer;
                 var G : logmas2);
var k, S : integer;
begin
    { išvalomos masyvo reikšmės }
    for k := 0 to n do
        for S := 0 to A do
            G[k, S] := false;
    { išdėstomos sumos }
    G[0, 0] := true; { inicializuojama kraštiė reikšmė }
    for k := 1 to n do
        for S := 0 to A do
            if G[k - 1, S] then
                G[k, S] := true
```

```
            else if (v[k] <= S) then
                if (G[k - 1, S - v[k]]) then
                    G[k, S] := true;
end;
```

Algoritmo sudėtingumas atminties ir laiko atžvilgiu yra vienodas – $O(n \cdot A)$.

Dabar galime grįžti prie *Teisingų dalybų uždavinio*. Pasinaudoję dinaminiu programavimu pagrįstu sumos dėstymo algoritmu, efektyviai apskaičiuosime, kokių verčių dovanų rinkinius įmanoma sudaryti. Iš šių rinkinių pakanka išrinkti tą, kurio vertė artimiausia $V/2$ (visų verčių sumos pusei). Belieka pasinaudoti apskaičiuotais duomenimis (masyvu G) ir pasirinkti, kurias dovanas reikia skirti Rusnei, o kurias – Emilijai. Sumos dėstymo uždavinio terminais tai reikštų nustatyti, kuriuos iš n dėmenų reikia sudėti, norint gauti skaičių A. Tad nagrinėjame lentelės langelį $G[n, A]$: n-asis dėmuo nereikalingas, jei A galima sudėti iš likusių $n-1$ dėmenų, t. y. $G[n-1, A] = true$. Priešingu atveju, n-asis narys būtinas. Toliau analogiškai tikriname $n-1$ dėmens reikalingumą, nagrinėdami langelius $G[n-1, A]$ arba $G[n-1, A-v_n]$.

```
type logmas = array [1..MAXN] of boolean;

procedure dalybos(var Rusnei : logmas;
                      var v : masyvas; n : integer);
{ rezultatas įrašomas į masyvą „Rusnei“: Rusnei[k] = true,
  jei k-ąją dovaną reikia skirti jai }
var G : logmas2;
    Vsum : longint;
    i, S : integer;
begin
    { suskaičiuojama visų verčių suma }
    Vsum := 0;
    for i := 1 to n do
        Vsum := Vsum + v[i];

    dėstyk(v, n, Vsum div 2, G);
```

```
        { randama artimiausia V/2 reikšmė, kurią galima išdėstyti }
        S := Vsum div 2;
        while not G[n, S] do
            S := S - 1;

        { nustatoma, kurias iš dovanų skirti Rusnei,
          kad jų bendra vertė būtų lygi S }
        for i := 1 to n do
            Rusnei[i] := false;
        i := n;
        for i := n downto 1 do
            { tikrinama, ar S vertės rinkiniui priklauso i-oji dovana }
            if not G[i - 1, S] then begin
                Rusnei[i] := true;
                S := S - v[i];
            end;
    end;
```

Šio sprendimo sudėtingumas sutampa su sumos dėstymo algoritmo sudėtingumu, kur dėstoma suma A neviršija $n \cdot m$, taigi yra toks: $O(n^2 \cdot m)$.

12.6 Uždavinys *Bibliotekoje*[45]

K bibliotekos darbuotojų buvo paskirta užduotis: peržiūrėti visas vienos lentynos knygas ieškant tam tikros informacijos. Šioje lentynoje iš viso yra n knygų. Darbą norima paskirstyti darbuotojams kuo lygesnėmis dalimis, tačiau knygos turėtų išlikti savo vietose, todėl buvo nuspręsta paprasčiausiai išskaidyti visą lentyną į k nesikertančių sričių, ir pavesti kiekvienam darbuotojui ieškoti informacijos tik vienoje srityje. Vis dėlto vienos knygos puslapių skaičiumi gerokai viršija kitas, todėl lentyną į k sričių norima išskaidyti optimaliai –

[45] Analogiškas uždavinys pateiktas S. Skienos knygoje *The Algorithm Design Manual* [6].

taip, kad didžiausias vienam darbuotojui tenkantis puslapių skaičius būtų kuo mažesnis.

Užduotis. *Duoti visų knygų puslapių skaičiai $p_1, p_2, \ldots, p_n$. Reikia rasti, kaip visą darbą darbuotojams paskirstyti optimaliai.*

Pradėkime analizuoti uždavinį nuo kelių paprastų pavyzdžių. Tegu visą darbą reikia padalyti trims darbuotojams, o lentynoje yra devynios knygos. Be abejo, jei visos knygos turėtų vienodą puslapių skaičių, tai lentyną galėtume skaidyti į tris lygias dalis:

100 100 100 | 100 100 100 | 100 100 100

Tačiau lentynos dalijimas lygiomis dalimis tikrai netikęs, jei puslapių skaičius knygose gerokai skiriasi:

100 200 300 | 400 500 600 | 700 800 900

Šiuo atveju pirmam darbuotojui tektų peržiūrėti 100 + 200 + 300 = 600 puslapių, o trečiajam — 700 + 800 + 900 = 2400, taigi net keturis kartus daugiau. Išbandę įvairius variantus, galime padaryti išvadą, kad geriausias įmanomas paskirstymas būtų toks:

100 200 300 400 500 | 600 700 | 800 900

Tuomet darbuotojams tektų peržiūrėti atitinkamai 1500, 1300 ir 1700 puslapių.

Ar yra kokia nors strategija, kurios laikydamiesi lentynoje esančias knygas visuomet padalytume optimaliai? Idealiu atveju visiems darbuotojams darbas paskirstomas lygiomis dalimis, t. y. kiekvienam darbuotojui tenkantis puslapių skaičius lygus visų knygų puslapių skaičių sumai, padalytai iš darbuotojų skaičiaus:

$P_{vid} = (p_1 + p_2 + ... + p_n) / k$. Todėl būtų natūralu apskaičiuoti šią reikšmę ir iš eilės parinkinėti sritis, stengiantis jų dydžius gauti kuo artimesnius P_{vid}, t. y. taikyti godžiąją strategiją.

Vadovaudamiesi šia strategija, gautume optimalų paskirstymą visuose kol kas nagrinėtuose pavyzdžiuose. Tačiau neskubėkime daryti išvadų. Bendru atveju galima gauti ir neoptimalų knygų paskirstymą: jei keliems darbuotojams skiriamas darbas yra šiek tiek mažesnis už vidurkį, tai paskutiniam gali susikaupti nemažai „papildomo“ darbo.

Tarkime, lentynoje iš eilės sudėtos 6 knygos po 80 puslapių, o toliau trys knygos, kurių puslapių skaičiai lygūs 100, 30 ir 200, ir jas reikia paskirstyti trims darbuotojams. Šiuo atveju $P_{vid} = 270$. Taigi vadovaudamiesi godžiąją strategija, pirmam darbuotojui skirtume peržiūrėti pirmas tris knygas (240 puslapių yra artimesnė reikšmė P_{vid}, negu 320), antram – taip pat tris knygas, ir galų gale gautume tokį knygų paskirstymą:

80 80 80 | 80 80 80 | 100 30 200

Daugiausiai darbo – 330 puslapių – tektų trečiajam darbuotojui. Tačiau optimaliu atveju didžiausias peržiūrimų puslapių skaičius būtų lygus 320:

80 80 80 80 | 80 80 100 | 30 200

Taigi optimalus paskirstymas gaunamas pirmajam darbuotojui skiriant keturias knygas. Godžioji strategija šito negalėjo numatyti, kadangi sprendimai priimami pagal labai paprastą kriterijų, nenumatant jų padarinių.

Sudarysime dinaminiu programavimu pagrįstą algoritmą šiam dalijimo uždaviniui spręsti. Jis visuomet ras optimalų paskirstymą, kadangi išanalizuos visus galimus variantus, tačiau tai atliks efektyviai.

Kad būtų paprasčiau, sutarsime knygų nuo i-osios iki j-osios puslapių skaičių sumą žymėti $S(i, j)$, t. y. $S(i, j) = p_i + p_{i+1} + \ldots + p_j$. Didžiausią vienam darbuotojui peržiūrėti tenkantį puslapių skaičių vadinsime *paskirstymo įverčiu*.

Taigi pradėkime nuo optimalios sprendinio struktūros nustatymo. Bet kuriame paskirstyme k-ajam darbuotojui tenka kažkiek knygų iš lentynos pabaigos, t. y. knygos nuo l-iosios iki n-osios, $l \leq n$. Kad ir koks būtų paskirstymas, daugiausiai darbo tenka arba k-ajam darbuotojui, arba kuriam nors kitam. Pirmu atveju (jei daugiausia darbo tenka k-ajam darbuotojui), paskirstymo įvertis lygus $S(l, n)$, o antruoju atveju susiduriame su analogišku, tik mažesniu, uždaviniu – optimaliu lentynos iki l-osios knygos (jos neįtraukiant) paskirstymu pirmiesiems $k - 1$ darbuotojų.

Pažymėkime optimalų n knygų paskirstymo k darbuotojų įvertį $M(k, n)$. Jei žinotume, kad optimalu k-ajam darbuotojui skirti knygas nuo l-osios iki n-osios (t.y. žinotume, kam lygus l), tai

$$M(k, n) = \max\{\, S(l, n), M(k - 1, l - 1) \,\}$$

Tačiau mes iš anksto nežinome, kiek knygų optimalu skirti paskutiniam darbuotojui. Todėl tenka išbandyti visus galimus variantus ir pasirinkti tą, kurio atveju gaunamo paskirstymo įvertis yra mažiausias. Taigi iš tiesų $M(k, n)$ apibrėžiamas taip:

$$M(k, n) = \min_{1 \leq l \leq n} \max\{\, S(l, n), M(k - 1, l - 1) \,\}$$

t. y. minimumas yra skaičiuojamas iš visų maksimumų, gaunamų, kai l kinta nuo 1 iki n.

Kad funkcijos reikšmes galėtume skaičiuoti pagal rekursyvų apibrėžimą, jį būtina papildyti kraštinėmis reikšmėmis: paskirstymo įvertis visada lygus nuliui, jei lentyna tuščia; jei yra tik vienas darbuotojas, tai jam atitenka visas darbas, kuris lygus $S(1, n)$.

$$M(k,n)=\begin{cases}0, & \textit{jei } n=0;\\ S(1,n), & \textit{jei } k=1;\\ \min\limits_{l=1}^{n} \max\{\, S(l,n), M(k-1,l-1)\,\}, & \textit{kitais atvejais.}\end{cases}$$

Rekursyviai apibrėžę optimalaus sprendinio vertę, jau galime sudaryti ją efektyviai apskaičiuojantį algoritmą. Tačiau nepamirškime, jog mus domina ne tik sprendinio vertė, bet ir pats sprendinys, t. y. optimalus lentynos paskirstymas. Skirtingai nuo ligi šiol nagrinėtų uždavinių, sprendinį sukonstruoti iš apskaičiuotos funkcijos reikšmių lentelės būtų per sudėtinga. Todėl skaičiuodami kaupsime papildomus duomenis: jei skaičiuodami $M(k, n)$ reikšmę nustatysime, kad k-ajam darbuotojui optimalu paskirti knygas nuo l-osios iki n-osios, tai dydį l pasižymėsime atskirame masyve ($D[k, n] := l$).

Toliau pateikiamas procedūros, apskaičiuojančios, kaip optimaliai paskirstyti knygų peržiūrėjimo darbą darbuotojams, tekstas.

```
const MAXN = ...; { maksimalus knygų skaičius }
      MAXK = ...; { maksimalus darbuotojų skaičius }
      BEGALINIS = MAXINT;

type masyvas = array [0..MAXN + MAXK] of integer;
     masyvas2 = array [1..MAXK, 0..MAXN] of integer;
```

```
procedure paskirstyk(k, n : integer;
                     p : masyvas; { psl. skaičius }
                     var įvertis : integer;
                     var nuo : masyvas);

    { apskaičiuoja knygų nuo i-osios iki j-osios puslapių skaičių sumą }
    function S(i, j : integer) : integer;
    var h : integer;
    begin
        S := 0;
        for h := i to j do
            S := S + p[h];
    end;

var i, j, l, v : integer; { pagalbiniai kintamieji }
    D, M : masyvas2;

begin
    { užpildomos kraštinės reikšmės }
    for i := 1 to k do
        M[i, 0] := 0;
    for j := 1 to n do begin
        M[1, j] := S(1, j);
        D[1, j] := 1;
    end;

    { apskaičiuojama likusi lentelės dalis }
    for i := 2 to k do
        for j := 1 to n do begin
            M[i, j] := BEGALINIS;
            { renkamasis minimumas... }
            for l := 1 to j do begin
                { ...iš maksimumų }
                v := max[46](S(l, j), M[i - 1, l - 1]);
                if v < M[i, j] then begin
                    M[i, j] := v;
                    D[i, j] := l;
                end;
            end;
        end;
```

[46] Funkcija `max` randa didesnįjį iš dviejų skaičių, jos nepateiksime.

```
        { sukonstruojamas optimalus sprendinys }
        įvertis := M[k, n];
        j := n;
        for i := k downto 2 do begin
            nuo[i] := D[i, j];
            j := D[i, j] - 1;
            { jei i-ajam darbuotojui skiriamos knygos nuo D[i, j], tai
              likusiems i - 1 darbuotojų reikia paskirstyti D[i, j] - 1 knygų}
        end;
        nuo[1] := 1;
    end;
```

Procedūra grąžina kelis rezultatus:

- gautojo (optimalaus) paskirstymo *įvertį*, t. y. kiek daugiausiai darbo teks vienam iš darbuotojų;

- lentynoje esančių knygų paskirstymą. Šis pateikiamas masyve *nuo*. *j*-ajam (bet kuriam, išskyrus paskutinį) darbuotojui paskirtos knygos yra intervalas [*nuo*[*j*], *nuo*[*j* + 1]), o *k*-ajam (paskutiniam) – [*nuo*[*k*], *n*].

Toliau pateikiame lenteles M ir D, gaunamas jau mūsų nagrinėto pavyzdžio atveju, kai $k = 3$, $n = 9$, o puslapių skaičiai pateikti lentelėje:

p_1	p_2	p_3	p_4	p_5	p_6	p_7	p_8	p_9
100	200	300	400	500	600	700	800	900

Skaičiuojant langelio $[k, n]$ reikšmę, išbandomos visos l reikšmės nuo 1 iki n, masyve M įsimenama mažiausia reiškinio $\max\{ S(l, n), M(k-1, l-1) \}$ reikšmė ir pasižymima masyve D.

M		n									
		0	1	2	3	4	5	6	7	8	9
k	1	0	100	300	600	1000	1500	2100	2800	3600	4500
	2	0	100	200	300	600	900	1100	1500	2100	2400
	3	0	100	200	300	400	600	900	1100	1500	1700

D		n									
		0	1	2	3	4	5	6	7	8	9
k	1	–	1	1	1	1	1	1	1	1	1
	2	–	1	2	3	4	4	5	6	6	7
	3	–	1	2	3	4	5	6	7	7	8

Aptarkime procedūros `paskirstyk` sudėtingumą. Algoritmas apskaičiuoja kiekvieną $k \times n$ dydžio lentelės langelį. Kiek gi laiko sugaištama vieno langelio reikšmei apskaičiuoti? Vidutiniu atveju išbandomų l reikšmių skaičius tiesiškai priklauso nuo n, o su kiekviena l reikšme skaičiuojama funkcijos S, sumuojančios puslapių skaičių iš tam tikro intervalo, reikšmė. Pastarosios funkcijos sudėtingumas taip pat tiesiškai priklauso nuo n, t. y. yra $O(n)$. Taigi:

- vienam langeliui sugaištama $O(n^2)$ laiko;
- bendras algoritmo sudėtingumas yra $O(n \cdot k \cdot n^2) = O(n^3 \cdot k)$.

Nors tai palankus (polinominis) sudėtingumas šiam gana sudėtingam uždaviniui, jį galima pagerinti efektyviau skaičiuojant funkcijos S reikšmes. Paprasčiausia būtų apskaičiuoti visas galimas jos reikšmes iš anksto ir įsiminti masyve, vėliau prireikus $S(i, j)$ reikšmės, tereiktų jos reikšmę paimti iš masyvo, taigi S sudėtingumas būtų $O(1)$. Visoms reikšmėms apskaičiuoti prireiktų $O(n^2)$ laiko ir tiek pat atminties. Bendro algoritmo sudėtingumas laiko atžvilgiu būtų $O(n^2 + n^2 \cdot k) = O(n^2 \cdot k)$.

Tačiau dar efektyvesnis, ir kur kas elegantiškesnis sprendimas yra iš anksto susiskaičiuoti knygų nuo 1-osios iki *i*-osios puslapių sumas visiems *i*, t. y. tegu $r_i = p_1 + p_2 + ... + p_i$. Jas suskaičiuoti galima per O(n), pastebėjus, kad $r_i = r_{i-1} + p_i$. Tuomet, jei mus domina knygų nuo *i*-osios iki *j*-osios puslapių suma, ją galima apskaičiuoti per O(1) (konstantinį laiką), atliekant vieną aritmetinę operaciją:

$$p_i + p_{i+1} + ... + p_j = (p_1 + p_2 + ... + p_j) - (p_1 + p_2 + ... + p_{i-1}) = r_j - r_{i-1}.$$

Žemiau pateiksime (pažymėdami specialiu komentaru pakeistas eilutes) efektyviau realizuotą procedūrą `paskirstyk`, kurios sudėtingumas yra O($n^2 \cdot k$) vietoje O($n^3 \cdot k$).

```
procedure paskirstyk(k, n : integer;
                     p : masyvas; { psl. skaičius }
                     var įvertis : integer;
                     var nuo : masyvas);

var i, j, l, v : integer; { pagalbiniai kintamieji }
    D, M : masyvas2;
    r : masyvas;          { pagalbinis masyvas}

begin
    { užpildomas masyvas r }                          // **
    r[0] := 0;                                        // **
    for j := 1 to n do                                // **
        r[j] := r[j - 1] + p[j];                      // **
    { užpildomos kraštinės reikšmės }
    for i := 1 to k do
        M[i, 0] := 0;
    for j := 1 to n do begin
        M[1, j] := r[j];                              // **
        D[1, j] := 1;
    end;

    { apskaičiuojama likusi lentelės dalis }
    for i := 2 to k do
        for j := 1 to n do begin
            M[i, j] := BEGALINIS;
```

```
                { renkamasis minimumas... }
                for l := 1 to j do begin
                    { ...iš maksimumų }
                    v := max(r[j] - r[l - 1],          // **
                             M[i - 1, l - 1]);         // **
                    if v < M[i, j] then begin
                        M[i, j] := v;
                        D[i, j] := l;
                    end;
                end;
            end;

    { sukonstruojamas optimalus sprendinys }
    { }
    ...
end;
```

12.7 Uždavinys *Sodas*[47]

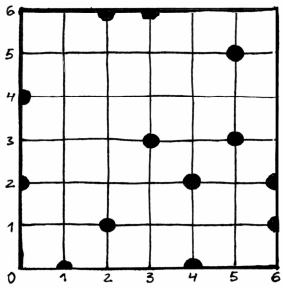
Kvadratiniame m × m dydžio sode auga n medžių. Laikoma, kad medis yra taškas, neturintis ilgio ir pločio. Koordinačių sistemos pradžia yra apatinis kairysis sodo kampas, o ašys yra lygiagrečios sodo tvoroms. Medžių vietą nusako jų koordinatės (x, y), išreikštos sveikaisiais skaičiais.

Užduotis. *Reikia rasti didžiausio stačiakampio, kuriame nebūtų medžių, plotą. Stačiakampio kraštinės turi būti lygiagrečios atitinkamoms sodo tvoroms (kraštinėms).*

Ieškomo stačiakampio kraštinėse gali augti medžiai, taip pat stačiakampio kraštinė gali sutapti su sodo tvora.

---

[47] Panašus uždavinys buvo pateiktas Vidurio Europos informatikos olimpiadoje 1995 metais.

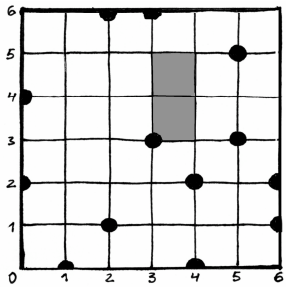


81 pav. Sodo pavyzdys; sode auga trylika medžių

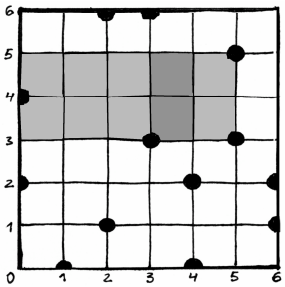
Įveskime keletą sąvokų. $P(x, y)$ pažymėsime tokį didžiausią *vienetinio pločio* stačiakampį, kurio viršutinio dešiniojo kampo koordinatės yra (x, y), o kairiosios kraštinės vidiniuose taškuose nėra medžių. Šio stačiakampio aukštį žymėsime $H_P(x, y)$. Nesunku matyti, kaip efektyviai apskaičiuoti H_p reikšmes:

$$H_p(x, y) = \begin{cases} 1, & \text{jei } y = 1 \text{ arba jei taške } (x-1, y-1) \text{ auga medis} \\ H_p(x, y-1) + 1, & \text{kitais atvejais} \end{cases}$$

Pažymėkime $T(x, y)$ didžiausią medžių neturintį stačiakampį, kuriam priklauso $P(x, y)$ ir kurio aukštis sutampa su $P(x, y)$ aukščiu.



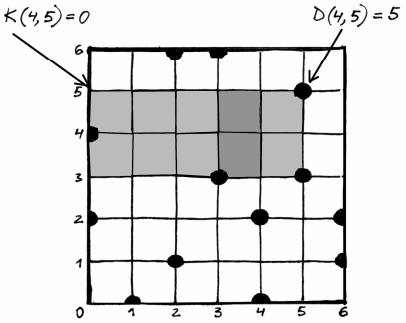
82 pav. Stačiakampis P(4, 5) pažymėtas pilkai; jo aukštis $H_P(4, 5) = 2$



83 pav. T(4, 5) – maksimalaus ploto stačiakampis, kuriam priklauso P(4, 5)

Stačiakampio T(x, y) kairiojo viršutiniojo kampo koordinatę x pažymėkime K(x, y), o dešiniojo viršutinio – D(x, y). Žinodami tai, iš karto galėsime apskaičiuoti T(x, y) plotą:

$$S_T(x, y) = (D(x, y) - K(x, y)) \cdot H_P(x, y).$$



84 pav. $S_T(4, 5) = (D(4, 5) - K(4, 5)) \cdot H(4, 5) = (5 - 0) \cdot 2 = 10$

Tarkime, kad žinome, kaip efektyviai apskaičiuoti funkcijų K ir D reikšmes. Tuomet užtenka peržiūrėti visus galimus stačiakampius T(x, y) (t. y. išbandyti visas galimas x ir y poras, kurių bus m^2) ir išrinkti didžiausią – jis ir bus ieškomasis sprendinys.

Toliau pateikta procedūra naudoja dvimatį loginį masyvą `medis`, kurio kiekvienas elementas `medis[x, y]` rodo, ar taške (x, y) auga medis.

```
const MAXM = ...; { maksimalus sodo dydis }
type lgmasyvas = array [0..MAXM, 0..MAXM] of boolean;
     kvmasyvas = array [1..MAXM, 1..MAXM] of integer;

function max_sodas(m : integer; { sodo dydis }
                   { medis[x, y] = true, jei (x, y) auga medis }
                   var medis : lgmasyvas) : integer;
var x, y, plotas : integer;
    Hp, K, D : kvmasyvas;
```

```
begin
    { apskaičiuojame Hp reikšmes }
    for y := 1 to m do
        for x := 1 to m do
            if y = 1 then
                Hp[x, y] := 1
            else if medis[x - 1, y - 1] then
                Hp[x, y] := 1
            else
                Hp[x, y] := Hp[x, y - 1] + 1;

    { apskaičiuojame K ir D reikšmes kiekvienam stačiakampiui T
      (šių procedūrų tekstas bus pateiktas vėliau) }
    skaičiuok_K(m, Hp, K);
    skaičiuok_D(m, Hp, D);

    { belieka peržiūrėti visus stačiakampius ir išrinkti didžiausią }
    max_sodas := 0;
    for y := 1 to m do
        for x := 1 to m do begin
            plotas := Hp[x, y] * (D[x, y] - K[x, y]);
            if plotas > max_sodas then
                max_sodas := plotas;
        end;
end;
```

Pagalvokime, kaip efektyviai apskaičiuoti masyvų K ir D reikšmes. K ir D reikšmės kiekvienai eilutei (t. y. kiekvienai koordinatei y) bus skaičiuojamos atskirai, tad panagrinėsime, kaip apskaičiuoti K ir D reikšmes, kai koordinatė y fiksuota.

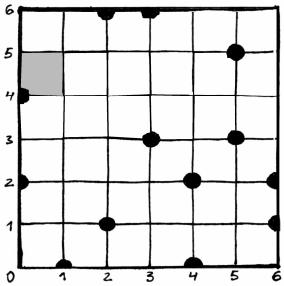
Pradėkime nuo masyvo D. Efektyviai reikšmėms apskaičiuoti bus naudojama dėklo[48] duomenų struktūra. Dėkle saugomos tos x koordinatės, kurioms D(x, y) dar neapskaičiuotas. Koordinatės x peržiūrimos iš kairės į dešinę (t. y. nuo 1 iki m) ir paeiliui dedamos į dėklą. Tačiau prieš tai patikrinama, galbūt $H_p(s, y) > H_p(x, y)$,

[48] Dėklo duomenų struktūra aprašyta 4.1 skyrelyje.

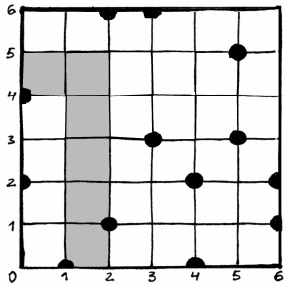
kur s – paskutinis dėkle esantis elementas. Jei $H_p(s, y) > H_p(x, y)$, tai stačiakampio, kuriam priklauso $H_p(s, y)$, daugiau į dešinę pratęsti negalima, taigi rastas dešinysis stačiakampio $T(s, y)$ kraštas: $D(s, y) = x - 1$. Tokiu atveju iš dėklo pašalinama koordinatė s, nes $D(s, y)$ jau apskaičiuota. Jei iš dėklo pašalinta koordinatė, vėl tikrinama, ar $H_p(s, y) > H_p(x, y)$, kur s – jau atnaujintas paskutinis dėklo elementas. Galbūt ir šiam elementui bus rastas $D(s, y)$, o pats elementas - pašalintas iš dėklo. Koordinatė x į dėklą įtraukiama tik tada, kai $H_p(s, y) \leq H_p(x, y)$ arba kai dėklas jau tuščias.

Peržiūrėjus visas x koordinates, dėkle liks tik tos x koordinatės, kurių stačiakampio $T(x, y)$ dešinysis kraštas sutampa su kvadrato kraštu.

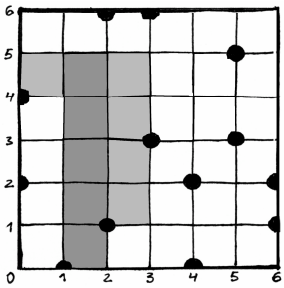
Pateiktame pavyzdyje parodysime, kaip skaičiuojamos funkcijos D reikšmės, konkrečiu atveju – kai $y = 5$.



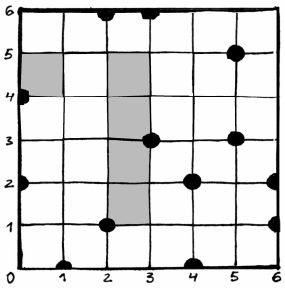
85-1 pav. $H_P(1, 5) = 1$;
Dėklas = [1]



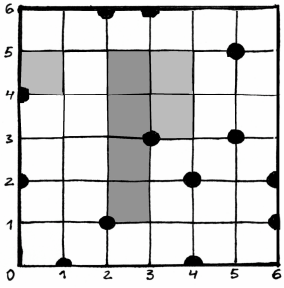
85-2 pav. $H_P(2, 5) = 5$;
$H_P(1, 5) \leq H_P(2, 5)$;
Dėklas = [1, 2]



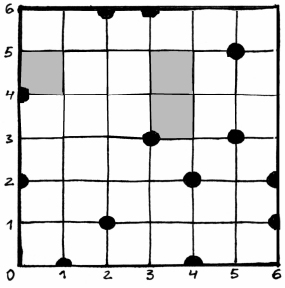
85-3 pav. $H_P(3, 5) = 4$; $H_P(2, 5) > H_P(3, 5)$; *radome* $D(2, 5) = 2$; *Dėklas* = [*1*]



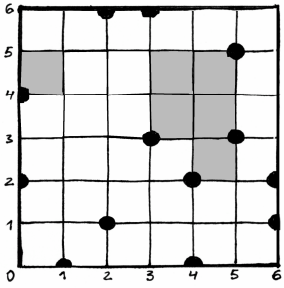
85-4 pav. $H_P(3, 5) = 4$; $H_P(1, 5) \leq H_P(3, 5)$; *Dėklas* = [*1, 3*]



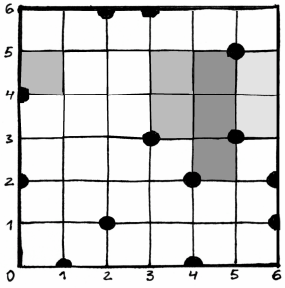
85-5 pav. $H_P(4, 5) = 2$; $H_P(3, 5) > H_P(4, 5)$; *radome* $D(3, 5) = 3$; *Dėklas* = [*1*]



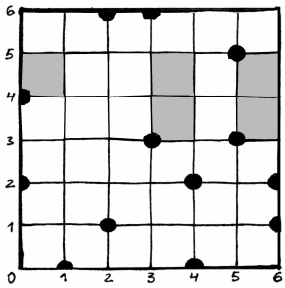
85-6 pav. $H_P(4, 5) = 2$; $H_P(1, 5) \leq H_P(4, 5)$; *Dėklas* = [*1, 4*]



85-7 pav. $H_P(5, 5) = 3$; $H_P(4, 5) \leq H_P(5, 5)$; *Dėklas* = [*1, 4, 5*]



85-8 pav. $H_P(6, 5) = 2$; $H_P(5, 5) > H_P(6, 5)$; *radome* $D(5, 5) = 5$; *Dėklas* = [*1, 4*]



85-8 pav. Dėklas = [1, 4, 6]
D(1, 5) = D(4, 5) = D(6, 5) = m = 6

K reikšmės skaičiuojamos analogiškai, tik koordinatės peržiūrimos iš dešinės į kairę.

```
type masyvas = array [1..MAXM] of integer;

procedure skaičiuok_D(m : integer;
                      var Hp : kvmasyvas;
                      var D : kvmasyvas);
var dėklas : masyvas;
    sk, x, y, s : integer;
begin
    sk := 0; { Elementų skaičius dėkle }
    for y := 1 to m do begin
        for x := 1 to m do begin
           if sk > 0 then begin
                s := dėklas[sk];
                while (sk > 0) and
                      (Hp[x, y] < Hp[s, y]) do
                begin
                    { rastas dešinysis T(s, y) kraštas (x - 1) }
                    D[s, y] := x - 1;
                    sk := sk - 1;
                    if sk > 0 then s := dėklas[sk];
                end;
           end;
           { koordinatė x dedama į dėklą }
           sk := sk + 1;
           dėklas[sk] := x;
        end;
```

```
        { jei dėkle likus koordinatė x, tai T(x, y) tęsiasi
          iki pat dešiniojo sodo krašto }
        while sk > 0 do begin
            s := dėklas[sk];
            D[s, y] := m;
            sk := sk - 1;
        end;
    end;
end;

procedure skaičiuok_K(m : integer;
                      var Hp : kvmasyvas;
                      var K : kvmasyvas);
var dėklas : masyvas;
    sk, x, y, s : integer;
begin
    sk := 0; { Elementų skaičius dėkle }
    for y := 1 to m do begin
        for x := m downto 1 do begin
           if sk > 0 then begin
               s := dėklas[sk];
               while (sk > 0) and
                     (Hp[x, y] < Hp[s, y]) do
               begin
                   { rastas kairysis T(s, y) kraštas (x) }
                   K[s, y] := x - 1;
                   sk := sk - 1;
                   if sk > 0 then s := dėklas[sk];
               end;
           end;
           { koordinatė x dedama į dėklą }
           sk := sk + 1;
           dėklas[sk] := x;
        end;
        { jei dėkle likus koordinatė x, tai T(x, y) tęsiasi
          iki pat kairiojo sodo krašto }
        while sk > 0 do begin
            s := dėklas[sk];
            K[s, y] := 0;
            sk := sk - 1;
        end;
    end;
end;
```

Šio sprendimo sudėtingumas pagal laiką ir atmintį – $O(m^2)$. Nesunkiai galime modifikuoti sprendimą taip, kad sudėtingumas pagal atmintį sumažėtų iki $O(m + n)$. Medžių koordinates galima saugoti vienmačiame $O(n)$ įrašų masyve, o kvadratą nagrinėti po vieną eilutę: apskaičiuoti H_p, K, ir D einamajai y koordinatei, išrinkti didžiausią iki šiol rastą stačiakampį ir toliau nagrinėti kitą y koordinatę. Skaičiuojant $K(x, y)$ ir $D(x, y)$ reikšmes, K ir D reikšmių su kitomis y koordinatėmis neprireikia, o skaičiuojant $H_p(x, y)$ reikšmę naudojamos tik $H_p(x, y - 1)$ reikšmės.

12.8 Kada taikyti dinaminį programavimą

Išsprendėme kelis uždavinius pritaikę dinaminį programavimą. Bendru atveju sunku įvertinti, ar uždavinį galima spręsti taikant dinaminį programavimą. Tačiau dažnai tokie uždaviniai pasižymi bendromis savybėmis. Šiame skyrelyje jas ir apžvelgsime.

Prieš taikant dinaminį programavimą reikėtų užduoti šiuos klausimus:

Ar tai optimizavimo uždavinys? Ar šiam uždaviniui galima rasti daug sprendinių, iš kurių mus domina tik vienas (ilgiausias, trumpiausias ar panašiai)? Dauguma dinaminiu programavimų sprendžiamų uždavinių yra būtent optimizavimo uždaviniai.

Ar uždavinyje aprašyto objekto elementai yra surikiuoti? Daugelio objektų elementai yra surikiuoti iš kairės į dešinę (t. y. tarp dviejų objektų įvestas santykis *kairiau*), arba apibrėžta kokia nors kitokia tvarka. Pavyzdžiui, muziejaus eksponatai (*Kuprinės uždavinyje*),

dovanos (*Teisingų dalybų uždavinyje*), medžiai[49] (*Sodo uždavinyje*), simbolių eilutės simboliai, iškiliojo daugiakampio viršūnės, lapai paieškos medyje ir pan. Tikėtina, kad optimizavimo uždavinį, kuriame objektų elementai yra surikiuoti, galima efektyviai išspręsti dinaminio programavimo metodu.

Jei objekto elementai nėra surikiuoti, tikriausiai teks atsisakyti dinaminio programavimo. Mat tokiu atveju uždavinį sprendžiant dinaminio programavimo metodu, laiko bei atminties sąnaudos būtų eksponentinės eilės (tai reiškia, kad sprendimas būtų visiškai neefektyvus).

Ar galima suskaidyti uždavinį į smulkesnius uždavinius, o tuos į dar smulkesnius, kol pasiekiamos elementarios ribinės situacijos? Sakykime, uždavinyje aprašytas objektas turi n elementų. Ar, paėmę mažiau nei n elementų, gausime tą patį uždavinį, tik su mažesniais parametrais? Jei ne – pritaikyti dinaminio programavimo nepavyks.

Ar smulkesnių uždavinių sprendiniai turi įtakos didesnių uždavinių sprendimui? Kokia informacija apie sprendinius mažesniems nei n elementų objektams yra būtina, norint rasti sprendinį objektui su n elementų? Ar turėdami sprendinius visiems mažesniems nei n elementų objektams bei n-ąjį elementą galime, gauti sprendinį objektui iš n elementų? Jei ne – dinaminio programavimo pritaikyti taip pat nepavyks.

Ar skaidant į smulkesnius uždavinius, tie smulkesni uždaviniai ima kartotis? Jei ne – dinaminio programavimo taikyti neverta. Nes *dinaminio programavimo* efektyvumas laiko atžvilgiu bus toks pat, kaip ir pilno perrinkimo atveju, tačiau pareikalaus kur kas daugiau

[49] *Sodo uždavinyje* medis A(x_1, y_1) yra *kairiau* nei medis $B(x_2, y_2)$, jei $x_1 < x_2$ arba $x_1 = x_2$ ir $y_1 < y_2$.

atminties. Dinaminio programavimo esmę sudaro dalinių uždavinių sprendinių įsiminimas, kai dėl to nebereikia iš naujo nespręsti tų pačių uždavinių. Tačiau jei daliniai uždaviniai[50] nesikartoja, tai nieko nelaimėsime taikydami dinaminį programavimą.

Ar sprendimui pakaks atminties? Taikant dinaminį programavimą dažnai reikia atsižvelgti į atminties sąnaudas. Jos būna kur kas didesnės nei sprendžiant uždavinį, pavyzdžiui, *grįžimo metodu*. Reikalingas atminties kiekis kartais gali nulemti, ar tam uždaviniui pavyks pritaikyti dinaminį programavimą.

Reikėtų atkreipti dėmesį į spręstų uždavinių sudėtingumą pagal atmintį. Pavyzdžiui, *Kuprinės uždavinio* sprendimo sudėtingumas atminties atžvilgiu yra $O(n \cdot S)$. Jei eksponatų svoriai būtų dideli, dinaminio programavimo pritaikyti nepavyktų. Tad pradinių duomenų ribojimai yra labai svarbūs įvertinant, ar uždavinio sprendimui galima taikyti dinaminį programavimą.

Beje, jei ieškoma tik sprendinio vertė, o ne pats sprendinys, dažnai galima sutaupyti atminties. Pavyzdžiui, *Kuprinės uždavinyje* užtektų saugoti ne visą lentelę, o tik dvi einamąsias lentelės eilutes, kadangi skaičiuojant k-osios lentelės eilutės reikšmes naudojamos tik reikšmės iš (k-1)-osios eilutės.

---

50 Yra tokia uždavinio sprendimo strategija *Skaldyk ir valdyk*, kai uždavinys padalijamas į mažesnius uždavinius, visi mažesni uždaviniai išsprendžiami taikant rekursiją ir sujungus gautus sprendinius gaunamas pradinio uždavinio sprendinys; tik šiuo atveju mažesni uždaviniai nesikartoja ir tarpusavyje neturi nieko bendra; *Greitojo rikiavimo algoritmas* yra tokios strategijos pavyzdys: rikiuojama seka dalijama į dvi dalis ir kiekviena dalis rikiuojama atskirai, tačiau vienos sekos dalies rikiavimas neturi įtakos kitos dalies rikiavimui.

13 STRATEGINIŲ STALO ŽAIDIMŲ ALGORITMAI

I had one item on my agenda today – not to lose.
Šiandien mano dienotvarkėje buvo tik vienas punktas – nepralaimėti.
Šachmatų čempionas Garis Kasparovas apie žaidimą šachmatais prieš kompiuterinę programą *Deep Junior*

Smagu ir įdomu leisti laisvalaikį žaidžiant šaškėmis, šachmatais, Go ar net kryžiukais ir nuliukais, turint po ranka tik languotą popieriaus lapą. Smagiausia žaisti dviese, tačiau ne mažiau įdomu pabandyti sužaisti stalo žaidimą su kompiuteriu, kai vieno iš dviejų žaidėjų ėjimus atlieka programa. Galimas ir dar vienas žaidimo būdas: parašyti programas, atliekančias ėjimus, ir surengti programų turnyrą.

Kuo daugiau patirties turi žaidėjas ir kuo geresnę žaidimo strategiją jis sugalvos, tuo daugiau turi šansų laimėti. O kaip gi su programa, atliekančia žaidėjo ėjimus? Juk kiekvieno žaidimo taisyklės ir strategijos yra skirtingos. Ar gali būti kokie nors principai, bendri visiems žaidimams? Ar yra kas bendra tarp, pavyzdžiui, žaidimo šachmatais ir kryžiukais ir nuliukais? Pasirodo, yra! Visų strateginių stalo žaidimų algoritmai programuojami remiantis tais pačiais principais, kuriuos aprašysime šiame skyrelyje.

Strateginiai stalo žaidimai žaidžiami ant žaidimo lentos ar kitaip pažymėto žaidimo ploto su specialiais žaidimo komponentais (kauliukais, figūrėlėmis, kortelėmis). Žaidimą žaidžia du žaidėjai, kurie paeiliui atlieka ėjimus. Žaidimas yra baigtinis, t. y. jis būtinai baigsis po baigtinio ėjimų skaičiaus.

Strateginių žaidimų algoritmas – tai algoritmas, realizuojantis tam tikrą žaidimo strategiją, kuria remiantis parenkamas tolesnis ėjimas. Vartotojo sąsaja ir teisėjavimas neįeina į šią sąvoką. Tai suprantama – parašyti vartotojo sąsają šachmatų žaidimo programai galėtų daugelis išmanančių kompiuterinės grafikos pradmenis. Tačiau

sukurti ir realizuoti strategiją, kuria žaidžiant pavyktų laimėti prieš rimtą varžovą, yra daug sudėtingesnis uždavinys.

Bendru atveju *strateginių stalo žaidimų* sąvoka gali būti platesnė, pavyzdžiui, žaidimą gali žaisti ne du, o daugiau žaidėjų, ėjimus jie gali atlikti ne tik paeiliui, bet ir praleisti ėjimą ar kelis kartus eiti iš eilės, remdamiesi konkrečiomis žaidimo taisyklėmis. Vaikams skirtuose žaidimuose ėjimus dažnai nulemia ir atsitiktinumas (pavyzdžiui, mesto kauliuko atsivertę taškai). Tačiau šiame skyrelyje nagrinėsime tik strateginius stalo žaidimus, paremtus klasikine stalo žaidimų samprata.

13.1 Trumpa stalo žaidimų istorija

Stalo žaidimai buvo populiarūs jau seovės civilizacijose. Seniausias žinomas stalo žaidimas rastas senovės Egipte – žaidimas *Senetas*. Piešiniai, kuriuose žaidžiamas šis žaidimas, buvo rasti ketvirto tūkstantmečio prieš mūsų erą kapavietėse. Piešiniuose matyti, kad Senetą galėjo žaisti du žaidėjai, kurie pradiniu momentu turėjo nuo 5 iki 10 figūrėlių. Deja, tikslių šio žaidimo taisyklių nėra išlikusių. Manoma, kad žaidimo lenta buvo 3×10 dydžio, keletas jos langelių turėjo specialius simbolius. Žaidėjas mesdavo keturias lazdeles (kiekviena kurių turėjo dvi skirtingas puses) ir priklausomai nuo to, kas atsiversdavo, atlikdavo ėjimą. Manoma, kad populiarus žaidimas *Triktrak* (angl. *Backgammon*) kilo iš Seneto.

86 pav. Seneto žaidimas

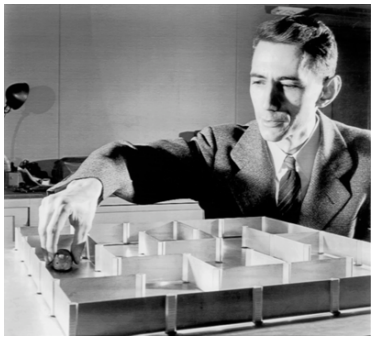
Kitas senovės egiptiečių žaidimas yra *Mehenas*. *Mehenas* yra mitologinės būtybės vardas. Pirmosios nuorodos į šį žaidimą yra apie 3000 metus prieš mūsų erą. Žaidimo lenta priminė susirangiusią gyvatę (spiralės formos) ir buvo sudaryta iš stačiakampiukų. Manoma, kad žaidimas buvo žaidžiamas su liūtus ar liūtes primenančių figūrėlių rinkiniais iš 3–6 figūrėlių.

Viduramžiais išpopuliarėjo žaidimai kauliukais, kurie dažniausiai buvo gaminami iš elnio ragų, taip pat kortomis. Buvo populiarūs ir kiti stalo žaidimai. Jų figūrėlės dažnai būdavo pusrutulio formos, gaminamos iš gintaro, kaulo, stiklo ar net arklio dantų. Vis didesnį populiarumą ėmė įgauti šachmatai. Stalo žaidimai buvo žaidžiami ne tik Azijoje, bet ir Europoje, Skandinavijoje. 1732 metais, keliaudamas po Laplandiją, žymus švedų botanikas Karlas Linėjus (*Carolus Linnaeus*) aprašė žaidimą *Tabula*. Lapiai jį žaidė 9×9 dydžio lentoje, o žaidimo figūrėles vadino „Švedais“ ir „Rusais“.

87 pav. Žaidžiamas stalo žaidimas Tabula

Stalo žaidimai labai išpopuliarėjo XX a. viduryje, ypač po II pasaulinio karo. O XX a. viduryje susidomėta ir strateginių žaidimų programavimu. Dar 1950 metais Klodas Elvudas Šenonas (Claude Elwood Shannon) paskelbė novatorišką darbą *Kompiuterio programavimas žaisti šachmatais (angl. Programming a Computer for Playing Chess*), kuriame aprašė, kaip skaičiavimo mašinai (ar kompiuteriui) galėtų būti parašyta programa, protingai žaidžianti šachmatais. Šenonas atkreipė dėmesį, kad teoriškai egzistuoja idealus sprendinys (t. y. ėjimų seka, kurią atlikdamas žaidėjas pasieks geriausią rezultatą), tačiau praktiškai jo

rasti neįmanoma. Jis pateikė dvi euristika paremtas strategijas, kuriomis programuojant strateginius stalo žaidimus remiamasi iki šiol:



88 pav. Klodas Elvudas Šenonas (Claude Elwood Shanon) 1916 – 2001

A Generuoti visus įmanomus ėjimus iki tam tikro gylio (t. y. generuoti visus galimus pirmuosius ėjimus, visus galimus antruosius ėjimus ir t. t.), tuomet euristiškai įvertinti kiekvieną gautą situaciją lentoje, sudaryti žaidimo medį ir išrinkti geriausią ėjimą;

B Generuoti tik perspektyvius ėjimus (žaidimo medžio šakas) ignoruojant ėjimus, kurie laikomi neperspektyviais.

Pirmoji šachmatais žaidžianti programa buvo parašyta ir įdiegta 1956 m. Taisyklės buvo supaprastintos, žaidžiama 6×6 lentoje. Kompiuteris buvo tik 11 kHz dažnio ir turėjo 600 žodžių atminties. Buvo realizuota A tipo Šenono aprašyta strategija, generuojami 4 lygių ėjimai. Tam prireikdavo net 12 minučių. Ši programa įveikdavo tik silpnus žaidėjus.

Po dvejų metų pirmą kartą buvo panaudotas Alfa–Beta atkirtimas, kuri leido padvigubinti analizuojamų ėjimų skaičių. Žaidimo programos buvo nuolat tobulinamos, pradėti rengti šachmatų programų turnyrai. Septintojo dešimtmečio viduryje (1975 m.) pradėta kurti *Cray Blitz* programa, kuri ilgą laiką buvo greičiausia šachmatais žaidžianti programa ir net septynerius metus buvo kompiuterių programų šachmatų čempionė.

89 pav. G. Kasparovas žaidžia prieš Deep Junior programą, 2003 m.

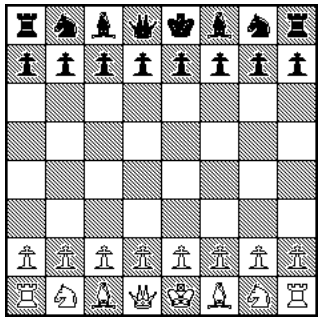
1988 metais pirmą kartą kompiuterinė sistema *Deep Thought*, skirta žaisti šachmatais, nugalėjo didmeistrį Bentą Larseną (*Bent Larsen*). Programa, kurią kūrė studentai, jau sugebėdavo išanalizuoti iki 750 000 pozicijų per sekundę ir iki dešimties ėjimų į priekį. Ši sistema pralošė šachmatų čempionui Bobiui Fišeriui (*Bobby Fischer)*. Vis dėlto, kuo toliau, tuo sunkiau didmeistriams sekėsi įveikti šachmatais žaidžiančias programas. Pirmoji kompiuterinė sistema, kuriai 1996 metais pralaimėjo partiją tuometinis pasaulio čempionas Garis Kasparovas, buvo *Deep Blue*. Tačiau tuomet turnyrą pasaulio čempionui vis tik pavyko laimėti. *Deep blue* buvo labai atnaujinta, neoficialiai netgi pervadinta į *Deeper blue*, ir 1997 metais Kasparovas jau pralaimėjo rezultatu 2,5:3,5. Tai buvo pirmas kartas, kai kompiuterinei sistemai pavyko įveikti pasaulio šachmatų čempioną matuojant žaidimo laiką įprastu būdu. 2003-iaisiais Gariui Kasparovui nepavyko įveikti ir *Deep Junior* programos – turnyras baigėsi lygiosiomis.

Šachmatai – tai tik vienas žaidimų, kuriam žaisti kuriamos kompiuterinės programos. Kasmet vyksta olimpiados[51], turnyrai, kuriuose tarpusavyje žaidžia programos ir kuriuose gali dalyvauti visi norintys.

[51] Žaidimų programų olimpiadas kasmet organizuoja ICGA (*International Computer Games Association*); Olandijos informatikos olimpiados kasmet organizuoja žaidimų programų turnyrą *CodeCup*.

13.2 Žaidimų medžiai, MiniMax paieška

Analizuodami strateginius stalo žaidimus sutarsime, kad abu varžovai **žaidžia optimaliai**, t. y. kiekvieną kartą renkasi patį palankiausią sau ėjimą.



90 pav. Pradinė šachmatų žaidimo pozicija

Bet kuris stalo žaidimas prasideda nuo **pradinės pozicijos**. Sutarsime, kad to paties žaidimo pradinė pozicija yra visada ta pati. Pavyzdžiui, žaidžiant šachmatais, pradinę poziciją sudaro juodos ir baltos figūros, išdėstytos tam tikra tvarka 8×8 lentoje, žaidžiant Go arba kryžiukais ir nuliukais, pradinę poziciją sudaro tuščia lenta.

Iš pradinės pozicijos pirmasis žaidėjas gali atlikti tam tikrus ėjimus. Visas šiais ėjimais gautas pozicijas vadinsime **pirmojo lygio pozicijomis**. Pozicijas, gautas atlikus vieną ėjimą iš pirmojo lygio pozicijų – **antrojo lygio** ir t. t.

Pavyzdžiui, žaidžiant kryžiukais ir nuliukais, pirmasis žaidėjas (jis būtinai žaidžia kryžiukais) gali atlikti 9 skirtingus pirmuosius ėjimus: padėti kryžiuką į bet kurį iš 9 langelių. Antrasis žaidėjas kiekvienu atveju gali pasirinkti vieną iš aštuonių ėjimų. Toliau pirmasis žaidėjas kiekvienu atveju gali atlikti septynis skirtingus ėjimus. Taip bus gaunamos trečiojo lygio pozicijos.

Analogiškai galima toliau generuoti ėjimus kol bus pasiektos **baigiamosios žaidimo pozicijos**, t. y. pozicijos, kai vienas žaidėjų laimi arba pasiekiamos lygiosios. Kryžiukų ir nuliukų atveju daugiausia gali būti 9 lygiai.

Iš žaidimo pozicijų sudaromas **žaidimo medis,** kurio viršūnės atitinka žaidimo pozicijas. Medžio šaknis atitiks pradinę žaidimo poziciją, jos antrinės viršūnės – visas pozicijas, kurias galima pasiekti vienu ėjimu iš startinės pozicijos (pirmojo lygio pozicijas) ir t. t.

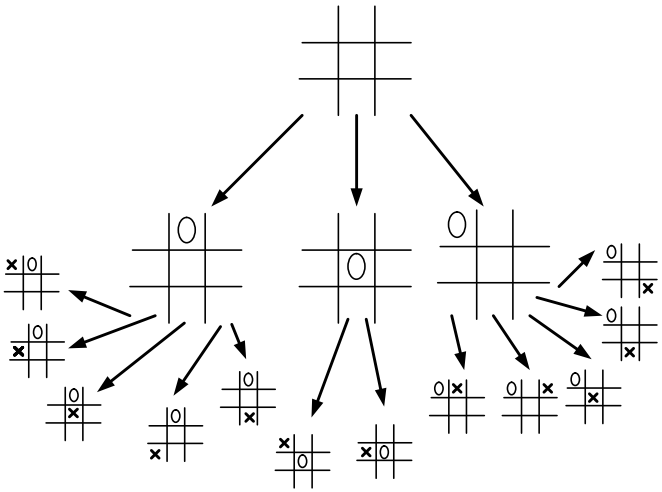
Gali būti, kad tą pačią poziciją galima gauti atlikus kelias skirtingas ėjimų sekas. Tačiau konstruojant žaidimo medį į tai neatsižvelgiama: jei ta pati žaidimo pozicija buvo gauta dvejomis skirtingomis ėjimų sekomis, tai ji žymima dviem medžio viršūnėmis.

Visų pirmojo lygio pozicijų atstumas iki medžio šaknies bus lygus 1, antrojo lygio pozicijų – 2, o k-ojo lygio pozicijų – k.

Žaidimo pozicija vadinama **laiminčia pozicija žaidėjui X**, jei žaidėjas X galės taip parinkti tolimesnius savo ėjimus, kad jis laimės nepriklausomai nuo to, kaip žais jo varžovas arba jeigu tai yra baigiamoji žaidimo pozicija, kurioje laimi žaidėjas X. Atkreipsime dėmesį, kad žaidėjo X laiminčioje pozicijoje ėjimo teisė gali priklausyti bet kuriam iš dviejų žaidėjų.

Žaidimo pozicija vadinama **pralaiminčia pozicija žaidėjui X**, jei nesvarbu, kaip žaidėjas X žaistų, varžovas gali parinkti tokius tolimesnius savo ėjimus, kad jis (varžovas) būtinai laimės, arba jei tai yra baigiamoji žaidimo pozicija, kurioje žaidėjas X pralaimi. Žaidėjo X pralaiminčioje pozicijoje ėjimo teisė taip pat gali priklausyti bet kuriam iš dviejų žaidėjų.

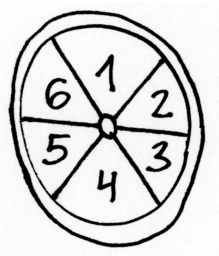
Be abejo, kiekvienas žaidėjas stengiasi atlikti tokius ėjimus, kurie nuvestų į laiminčias žaidimo pozicijas.



91 pav. Kryžiukų ir nuliukų žaidimo medžio pavyzdys; paveiksle parodyti tik du medžio lygiai; pateiktas nepilnas medis (atmestos simetriškos pozicijos)

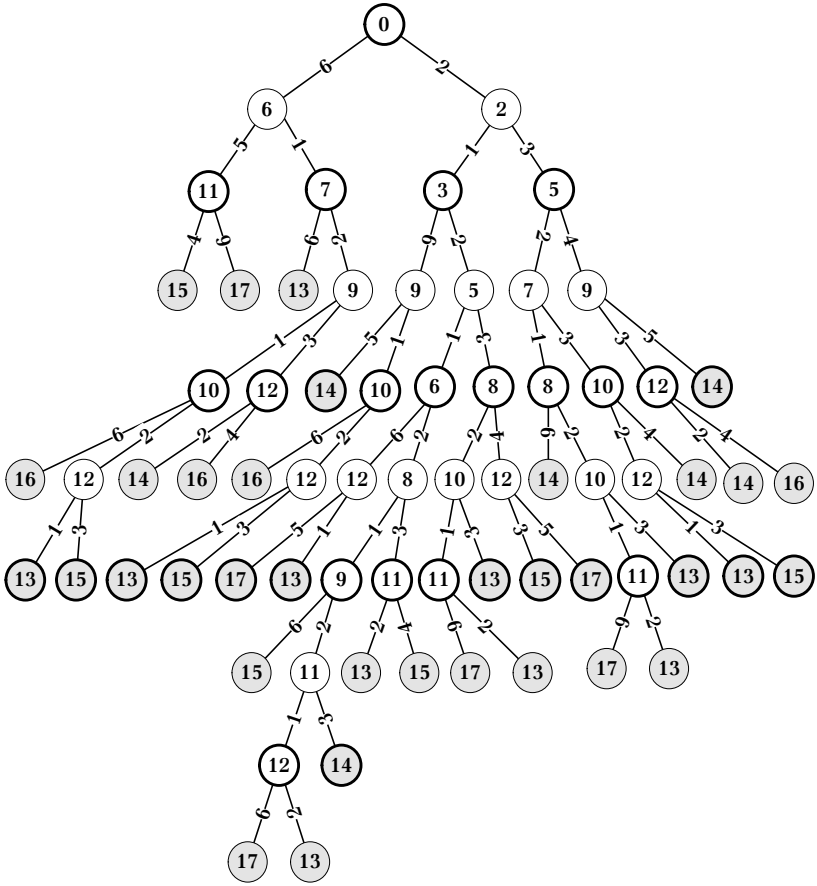
Paanalizuokime konkretų uždavinį:

Lošimas disku[52]*: du žaidėjai paeiliui sukioja diską per vieną segmentą į kairę arba į dešinę. Žaidėjas, pasukęs diską, perskaito jo viršutiniame segmente atsiradusį skaičių n ir jį prideda prie sumos s (bendros abiems žaidėjams). Lošimo pradžioje s = 0, o diskas atsisukęs ties skaičiumi 1. Lošimas baigiamas, kai s pasiekia arba viršiją iš anksto sutartą skaičių m (pavyzdžiui, m = 13). Laimi tas, kas atlieka paskutinį ėjimą, t. y. pasiekia arba viršija m.*



92 pav. Disko pozicija pradiniu momentu

[52] Panašus uždavinys buvo pateiktas Lietuvos informatikos olimpiadoje III etape 1998 metais.

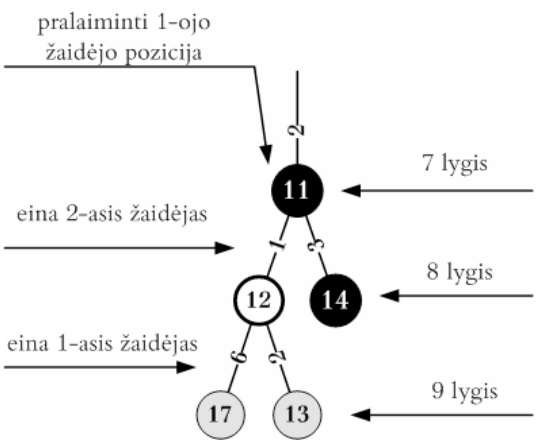


93 pav. Pilnas Lošimo disku medis, kai m = 13; medžio lapai (t. y. baigiamosios žaidimo pozicijos pavaizduotos pilka spalva, lyginių lygių pozicijos (iš kurių ėjimus atlieka pirmasis žaidėjas), apvestos storesne linija; apskritimo viduje įrašyta s reikšmė, o ant ėjimus žyminčių briaunų – žaidėjo ėjimai, t. y. atsukamos disko pozicijos.

Analizuosime žaidimą *Lošimas disku*, kai, pavyzdžiui, $m = 13$.

93 paveiksle pateiktas pilnas žaidimo medis. Medis turi 9 lygius – tai reiškia, kad galimi ne daugiau kaip 9 ėjimai. Bet kurią žaidimo poziciją nusako pora (s, d), kur s – bendra abiems žaidėjams suma, o d – skaičius, užrašytas ant į viršų atsisukusio disko segmento. Visas medžio pozicijas bandysime skirstyti į laiminčias ir pralaiminčias pirmajam žaidėjui (laiminti pirmojo žaidėjo pozicija tuo pačiu yra pralaiminti antrojo žaidėjo pozicija ir atvirkščiai).

Pradėkime nuo 9-ojo lygio. Abi pozicijos yra baigiamosios ir laiminčios tam žaidėjui, kuris atliko ėjimą, t. y. pirmajam. Lipkime aukštyn į 8-ąjį lygį, kuriame galimos dvi pozicijos. Iš pirmosios pozicijos ($s = 12$; $d = 1$) pirmasis žaidėjas gali atlikti tik laiminčius ėjimus, taigi ši pozicija yra laiminti pirmajam žaidėjui.

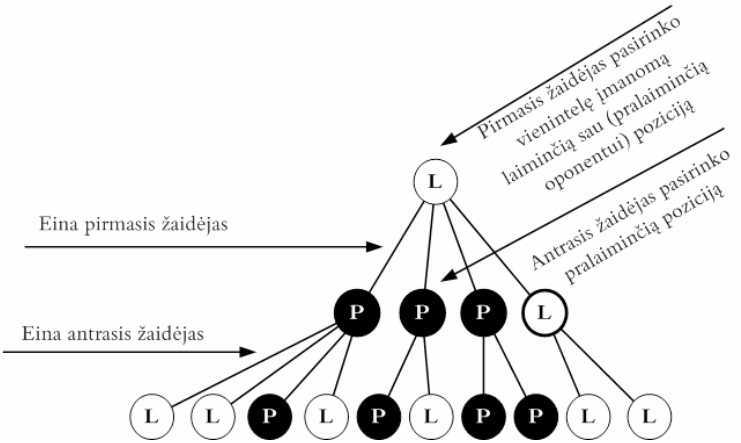


94 pav. Žaidimo medžio fragmentas, pralaiminčios pirmojo žaidėjo pozicijos pažymėtos juodai

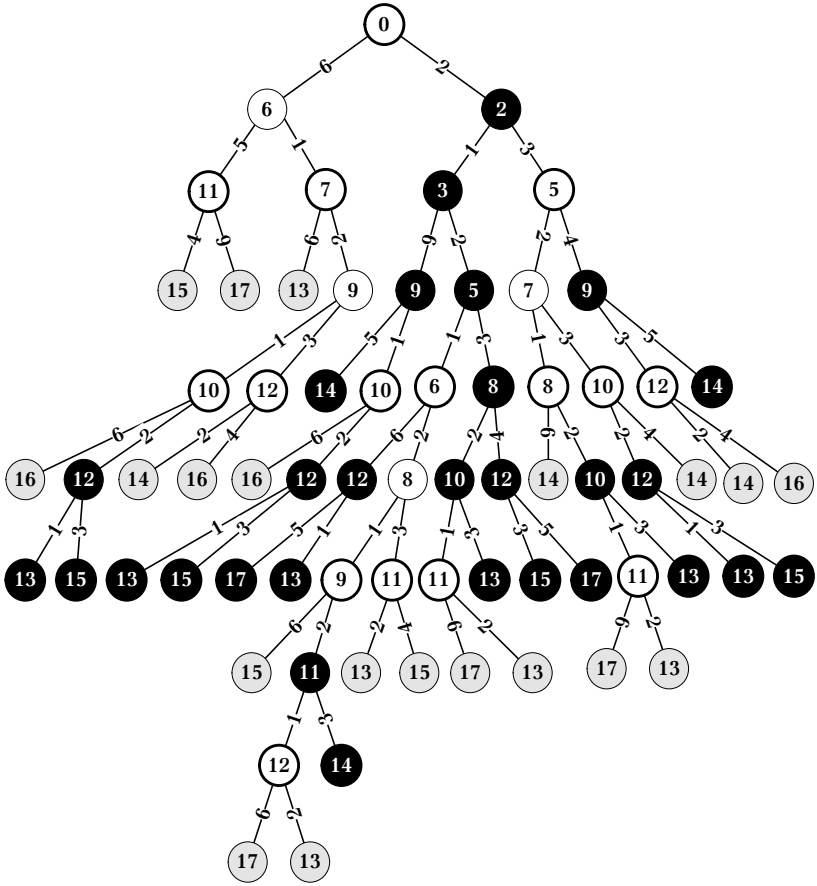
Tuo tarpu antroji pozicija ($s = 14$; $d = 3$) yra baigiamoji žaidimo pozicija ir gaunama ėjimą atlikus antrajam žaidėjui, taigi ji yra pralaiminti pirmojo žaidėjo pozicija.

7-ajame lygyje yra net 8 žaidimo pozicijos ir tik viena jų ($s = 11$; $d = 2$) nėra baigiamoji. Visos baigiamosios pozicijos yra laiminčios pirmajam žaidėjui, nes jos gaunamos pirmajam žaidėjui atlikus ėjimą. O kaip gi su pozicija ($s = 11$; $d = 2$)?

Iš šios pozicijos toliau ėjimą atliktų antrasis žaidėjas. Pasukęs diską, jis gali arba atsukti vienetą ir pakliūti į laiminčią pirmojo žaidėjo poziciją, arba atsukti trejetą ir laimėti pats. Žinoma, žaisdamas optimaliai, jis atsuks trejetą. Taigi ši pozicija yra pralaiminti pirmojo žaidėjo pozicija (94 pav.)



95 pav. Medžio pozicijos analizuojamos iš apačios į viršų; pirmojo žaidėjo laimėjimo pozicijos žymimos L raide, pralaimėjimo – P raide; jeigu eina pirmasis žaidėjas ir galima nors viena laiminti poziciją, jis ją ir renkasi; jeigu eina antrasis žaidėjas ir galima nors viena pralaiminti pirmojo žaidėjo pozicija, tai ji ir pasirenkama



96 pav. Tas pats žaidimo medis kaip ir 93 pav., tik visos pozicijos suskirstytos į laiminčias ir pralaiminčias pirmajam žaidėjui; pralaiminčios pirmojo žaidėjo pozicijos pažymėtos juodai; matome, kad jei abu žaidėjai žaidžia optimaliai, pirmasis žaidėjas būtinai laimės

Šis pavyzdys ir paaiškina optimalios žaidimo strategijos esmę: jei tarp pozicijų, į kurias žaidėjas (nesvarbu kuris) gali pakliūti atlikęs ėjimą, yra **nors viena laiminti** to žaidėjo atžvilgiu, tai tas žaidėjas būtinai ją ir rinksis (95 pav.).

Likusios pozicijos lipant medžiu aukštyn analogiškai suskirstomos į laiminčias ir pralaiminčias pirmojo žaidėjo pozicijas (žr. 96 pav.).

Užkopę iki pirmojo lygio matome, kad abiems žaidėjams žaidžiant optimaliai, laimės pirmasis.

Išanalizavę žaidimo medį (t.y. kiekvienai pozicijai priskyrę atributą, ar ji laiminti, ar pralaiminti) iš kiekvienos pozicijos galime pasirinkti palankiausią ėjimą. Deja, realybėje pilno žaidimo medžio dažniausiai suformuoti nepavyksta, tad apsiribojama tam tikru gyliu, iki kurio bus išskleidžiamas medis.

Pasiekus tą gylį, žaidimo pozicijos (pavadinkime jas ribinėmis), nors ir nėra baigiamosios, įvertinamos euristiškai. Pozicijos vertinamos skaičiais pasirinktame intervale [*min_vertė*, *max_vertė*]. Sutariama, kad pasirinkto žaidėjo (nevarbu kurio) laiminti pozicija bus vertinama maksimaliu intervalo skaičiumi *max_vertė*, o pralaiminti – minimaliu intervalo skaičiumi *min_vertė*. Šis žaidėjas dar vadinamas *maksimizuojančiu* žaidėju, o jo priešininkas – *minimizuojančiu* žaidėju.

Tad kuo geresnė yra pozicija maksimizuojančiam žaidėjui, tuo aukštesnį įvertinimą ji turėtų gauti. Ir atvirkščiai – kuo mažesnis pozicijos įvertinimas, tuo pozicija palankesnė minimizuojančiam žaidėjui. *Ėjimo įverčiu* vadinsime pozicijos, į kurią pakliūnama tuo ėjimu, įvertis.

Kol kas daugiau nesigilinsime į tai, kaip gaunami euristiniai pozicijų įverčiai[53], tiesiog laikysime, kad pasiekę ribinę poziciją galime gauti jos įvertį.

O kaip gi randamas bet kurios (nebūtinai ribinės ar baigiamosios) žaidimo pozicijos įvertis? Tai atlieka **MiniMax** algoritmas: rekursiškai iš viršaus į apačią formuojamas žaidimo medis ir vertinamos pozicijos. Jei pozicija *P* yra baigiamoji arba ribinė, tai pozicija P įvertinama tiksliai (pirmuoju atveju) arba euristiškai (antruoju atveju). Jei pozicija *P* nėra baigtinė, ir dar galima rekursiškai vertinti gilesnes pozicijas, tai išbandomi visi galimi ėjimai ir išrenkamas bei grąžinamas geriausio ėjimo įvertis.

Jei iš pozicijos P atliekant visus įmanomus ėjimus galima pakliūti į pozicijas P_1, P_2, ...P_k, o šių pozicijų įverčiai jau yra žinomi: v_{p1}, v_{p2}, v_{p3}, ... v_{pk}, tuomet pozicijos P įvertis yra toks:

$$V_p = \begin{cases} \max\{v_{p_1}, v_{p_2}, ..., v_{p_k}\}, \textit{ jei eis maksimizuojantis žaid.} \\ \min\{v_{p_1}, v_{p_2}, ..., v_{p_k}\}, \textit{ jei eis minimizuojantis žaid.} \end{cases}$$

Kadangi šį algoritmą galima taikyti įvairiausiems žaidimas, tai neprisirišime prie konkretaus žaidimo ir algoritmą pateiksime **pseudokodu**[54].

```
MiniMax(gylis, pozicija, žaidėjas)
// „gylis“ nurodo iki kokio lygio skleisime žaidimo medį
// „pozicija“ parodo nuo kokios pozicijos analizuosime žaidimą

    jei žaidėjas yra maksimizuojantis
        tai grąžink Max(gylis, pozicija)
        kitu atveju grąžink Min(gylis, pozicija)
```

[53] Apie euristinį pozicijų vertinimą detaliau kalbama 13.3 skyrelyje.

[54] Pseudokodas yra algoritmo aprašymas, kai naudojami programavimo kalbų struktūriniai elementai, tačiau praleidžiami kalbai būdingi sintaksės elementai.

```
Max(gylis, pozicija)
    jei žaidimas baigtas arba (gylis = 0)
        tai grąžink įvertinimas(pozicija, mini_zaid)
        // įvertinama pozicija, kai paskutinį ėjimą atliko
        // minimizuojantis žaidėjas
    kitu atveju
        geriausias := MIN_VERTĖ
        kiekvienam galimam ėjimui e
            atlik ėjimą(e, pozicija)
            įvertis := Min(gylis - 1, pozicija)
            jei įvertis > geriausias tai
                geriausias = įvertis
            atšauk ėjimą(e, pozicija)
        grąžink geriausias

Min(gylis, pozicija)
    jei žaidimas baigtas arba (gylis = 0)
        tai grąžink įvertinimas(pozicija, max_zaid) //**[55]
        // įvertinama pozicija, kai paskutinį ėjimą atliko
        //maksimizuojantis žaidėjas
    kitu atveju
        geriausias := MAX_VERTĖ                    //**
        kiekvienam galimam ėjimui e
            atlik ėjimą(e, pozicija)
            įvertis := Max(gylis - 1, pozicija)    //**
            jei įvertis < geriausias tai           //**
                geriausias = įvertis
            atšauk ėjimą(e, pozicija)
        grąžink geriausias
```

Realiose situacijose žaidimo pozicija dažnai nusakoma sudėtinga duomenų struktūra ir ji neperduodama per parametrus, bet pasiekiama kaip globalusis kintamasis.

Pritaikysime šį algoritmą *Lošimui su disku*. Kol kas detaliau nesiaiškinome, kaip euristiškai vertinti pozicijas, tačiau, kaip matėme, kai

[55] Tokiu komentaru pažymėtos tos procedūros `Min` eilutės, kurios skiriasi nuo procedūros `Max`.

$m = 13$, žaidimo medį galima išskleisti pilnai. Tad realizuodami algoritmą taip pat laikysime, kad žaidimo medį galima pilnai išskleisti. Žaidžiant šį žaidimą galimi tik du skirtingi pozicijų įverčiai – laiminti ir pralaiminti (lygiųjų būti negali), tad pozicijas vertinsime ne skaičiais, o loginėmis reikšmėmis: „minimali“ pozicijos vertė bus *false*, „maksimali“ – *true*.

```
const M = 13; { šį skaičių pasiekęs ar viršijęs, žaidėjas laimi }

type Tpozicija = record
         s, d : integer;
         { s ir d nusako konkrečią žaidimo poziciją }
     end;

procedure atlik_ėjimą(sukti_pirmyn : boolean;
                      var p : Tpozicija);
begin
    if sukti_pirmyn then
        p.d := (p.d + 4) mod 6 + 1
    else
        p.d := p.d mod 6 + 1;
    p.s := p.s + p.d;
end;

procedure atšauk_ėjimą(sukti_pirmyn : boolean;
                       var p : Tpozicija);
begin
    p.s := p.s - p.d;
    if sukti_pirmyn then
        p.d := p.d  mod 6 + 1
    else
        p.d := (p.d + 4) mod 6 + 1;
end;

function Min(pozicija : TPozicija) : boolean; forward;

function Max(pozicija : TPozicija) : boolean;
{ randa pozicijos įvertį (ar tai laiminti pirmojo žaidėjo pozicija),
  jei ėjimą iš jos atlieka pirmasis (maksimizuojantis) žaidėjas }

var sukti_pirmyn, įvertis : boolean;
```

```
begin
    if pozicija.s >= M then { jei žaidimas baigtas }
        Max := false
        { nes paskutinį ėjimą atliko antrasis žaidėjas }
    else begin
        Max := false;
        for sukti_pirmyn := false to true do begin
             atlik_ėjimą(sukti_pirmyn, pozicija);
             įvertis := Min(pozicija);
             if (Max = false) and (įvertis = true)
             then { jei Max < įvertis }
                 Max := įvertis;
             atšauk_ėjimą(sukti_pirmyn, pozicija);
        end;
    end;
end;
```

```
function Min(pozicija : TPozicija) : boolean;
{ randa pozicijos įvertį (ar tai laiminti pirmojo žaidėjo pozicija),
  jei ėjimą iš jos atlieka antrasis (minimizuojantis) žaidėjas }
var sukti_pirmyn, įvertis : boolean;
begin
    if pozicija.s >= M then { jei žaidimas baigtas }
        Min := true
        { nes paskutinį ėjimą atliko pirmasis žaidėjas }
    else begin
        Min := true;
        for sukti_pirmyn := false to true do begin
            atlik_ėjimą(sukti_pirmyn, pozicija);
            įvertis := Max(pozicija);
            if (Min = true) and (įvertis = false)
            then { jei Min > įvertis }
                Min := įvertis;
            atšauk_ėjimą(sukti_pirmyn, pozicija);
         end;
    end;
end;
```

```
function MiniMax(žaidėjas : integer;
                 pozicija : Tpozicija) : boolean;
{ randa pozicijos įvertį („true“, jei tai laiminti pirmojo žaidėjo pozicija
  ir „false“ priešingu atveju }

begin
    if žaidėjas = 1 then
        { jei ėjimą atliks pirmasis (maksimizuojantis) žaidėjas }
        MiniMax := Max(pozicija)
    else
        MiniMax := Min(pozicija);
end;
```

13.3 Euristinis pozicijų vertinimas bei iteratyvus paieškos gilinimas

Kaip jau minėta, dažnai pilno žaidimo medžio suformuoti nepavyksta ir pasiekus ribinį gylį žaidimo pozicijas tenka vertinti euristiškai. Jeigu medį galima išskleisti pilnai, užtenka trijų tipų pozicijų: laimėjimų, pralaimėjimų ir lygiųjų. Tuo tarpu euristinis pozicijos vertinimas yra kur kas sudėtingesnis. Tai atliekant reikia panaudoti kuo daugiau žinių apie konkretų žaidimą.

Euristinis pozicijos įvertis gaunamas žaidimo poziciją įvertinant skaičiumi. Maksimizuojantis ir minimizuojantis žaidėjai turi vertinti žaidimą analogiškai. Pavyzdžiui, vienas žaidėjas mano, kad yra geroje pozicijoje, tai jo oponentas turi manyti, kad jis (t. y. oponentas) yra prastoje pozicijoje.

Euristiškai vertinant pozicijas dažniausiai sudaroma tokio tipo įvertinimo funkcija:

$$\textit{Įvertis}(\textit{pozicija}) = K_1Ž_1 + K_2Ž_2 + K_3Ž_3 + ... + K_nŽ_n,$$

kur $Ž_i$ yra skaičiais išreikštos tam tikros žinios apie žaidimą, įvertinančios poziciją kažkokiu konkrečiu aspektu, o K_i – koeficientai, suteikiantys žinioms skirtingą svorį.

Euristinės įvertinimo funkcijos dažniausiai turi dėmenis, kuriuose kaupiamos tokio tipo žinios apie žaidimą:

- **materialūs įverčiai**; pavyzdžiui, šachmatų ar šaškių figūrų skaičius lentoje arba savo ir priešininko figūrų kiekių skirtumas (šachmatuose kiekvienos rūšies figūrai dažnai suteikiamas svoris);

- **erdvė**; kai kuriuose žaidimuose yra labai svarbu, kiek erdvės kontroliuoja vienas ar kitas žaidėjas, taigi žaidėjo kontroliuojamą erdvę galima išreikšti skaičiumi ir įtraukti į įvertinimo funkciją;

- **mobilumas**; kiek ėjimų galima atlikti iš esamos pozicijos; t. y. tikėtina, kad jei turite daugiau galimybių paeiti, didesnė tikimybė, kad bent vienas šių ėjimų nuves į gerą poziciją; šis įvertis nepasitvirtino kuriant šachmatų žaidimo algoritmus, tačiau pasirodė labai naudingas žaidžiant, pavyzdžiui, *Otelo*;

- **tempas**; kai kuriuose žaidimuose yra svarbu tai, kuris žaidėjas turi iniciatyvą, o kuris tik atsako į priešininko ėjimus (pavyzdžiui, galbūt jūsų figūra yra puolama ir jai teks trauktis);

- **grėsmės**; kiek jūsų figūrų yra puolama, gal gali atsitikti dar kažkas negero, pavyzdžiui, pėstininkas taps valdove arba oponentas užims dalį jūsų teritorijos;

- **forma**; kai kuriuose žaidimuose yra gan svarbu, kaip išsidėstę figūros; pavyzdžiui, šachmatuose gretimuose stulpeliuose vienas paskui kitą stovintys pėstininkai laikomi stipresne kombinacija, nei tame pačiame stulpelyje esantys pėstininkai;

- **išskirtinės situacijos**; beveik kiekviename žaidime pasitaiko išskirtinių situacijų, kuriose žmogus žino, kaip geriausia sužaisti; kartais verta paaukoti figūrą ir taip laimėti dar daugiau; tam tikros išskirtinės situacijos taip gali būti įtrauktos į euristinį vertinimą.

Deja, kuo daugiau žinių apie žaidimą įtraukiame į programą, tuo lėčiau programa veikia. Greitą ir prastai žaidžiančią programą galima pagerinti į algoritmą įtraukiant daugiau žinių apie žaidimą. Tačiau šios papildomos žinios gali padaryti programą lėtesne (per leistiną laiką spėjančią išanalizuoti mažiau žaidimo lygių) ir tada ji gali žaisti netgi prasčiau nei prieš tobulinimą. Taigi reikia išlaikyti tinkamą balansą tarp efektyvumo ir žinių.

O kaip gi apskaičiuoti ribinį gylį, kurį pasiekus reiktų daugiau nebesiplėsti ir pradėti vertinti pozicijas euristiškai? Programuojant žaidimų algoritmus taikoma gan paprasta strategija: *MiniMax* algoritmas įvykdomas analizuojant pozicijas iki pirmojo gylio, po to (jeigu dar užtenka laiko) – *MiniMax* algoritmas vykdomas iš naujo, tik pozicijos analizuojamos iki antrojo gylio, tuomet – jei dar užtenka laiko – iki trečiojo gylio ir taip toliau, kol išnaudojami laiko ar atminties limitai. Tai ir yra iteratyvus paieškos gilinimas. Toliau pateiktas algoritmas atlieka iteratyvų paieškos gilinimą maksimizuojančiam žaidėjui.

```
Rask_ėjimą(pozicija, žaidėjas)
    gylis = 0
    kol neužtrukta daug laiko
        gylis = gylis + 1
        pozicija := PRADINĖ
        geriausias_įvertis := MIN_VERTĖ
        kiekvienam galimam žaidėjo ėjimui e
            atlik ėjimą(e, pozicija)
            įvertis :=
                Minimax(gylis, pozicija, žaidėjas);
            jei įvertis > geriausias_įvertis
                tai geriausias_įvertis := įvertis
                    geriausias_ėjimas := e
            atšauk ėjimą(e, pozicija)
    grąžink geriausias_ėjimas
```

Tai gali pasirodyti labai neefektyvu, mat vykdant kitą iteraciją ankstesni skaičiavimai nepanaudojami. Iš tiesų tai nėra taip neefektyvu kaip atrodo iš pirmo žvilgsnio, kadangi nagrinėjamų pozicijų skaičius didinant gylį auga eksponentiškai. Taigi laikas, reikalingas algoritmo vykdymui iki gylio n, yra dažniausiai daug didesnis už laiką, reikalingą paieškai iki gylio $n-1$. Tarkime, kad kiekviename lygyje iš kiekvienos pozicijos galima atlikti m ėjimų.

Tuomet paieškai iki gylio n užtrunkama

$$O(1 + m + m^2 + m^3 + ... + m^n) = O((m^{n+1} - 1)/(m - 1)) = O(m^n).$$

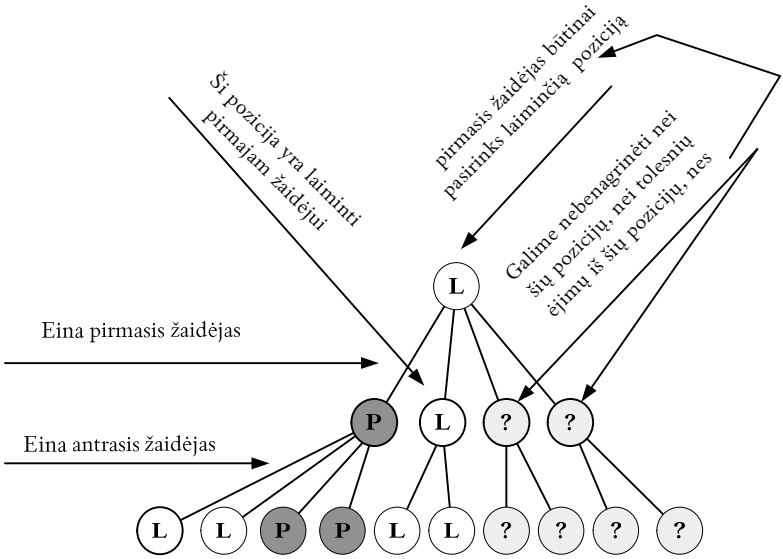
Jei paieška bus gilinama iteratyviai, tuomet bus užtrunkama

$$O(1 + (1 + m) + (1 + m + m^2) + ... + (1 + m + m^2 + ... + m^n)) = O((m^{n+2} - 1)/(m - 1)^2 - (n - 1)/(m - 1)) = O(m^n).$$

13.4 Alfa-Beta atkirtimas

Minimax paieška nėra efektyvi, nes išnagrinėjami visi galimi ėjimai, o jų kiekis, ieškant gilyn, auga eksponentiškai. Tačiau ar būtina išanalizuoti visus galimus ėjimus ir rasti kiekvieno ėjimo (tiksliau

pozicijos, į kurią pakliūvama atlikus tą ėjimą) įvertį? Panagrinėkime 97 paveiksle pateiktą situaciją.



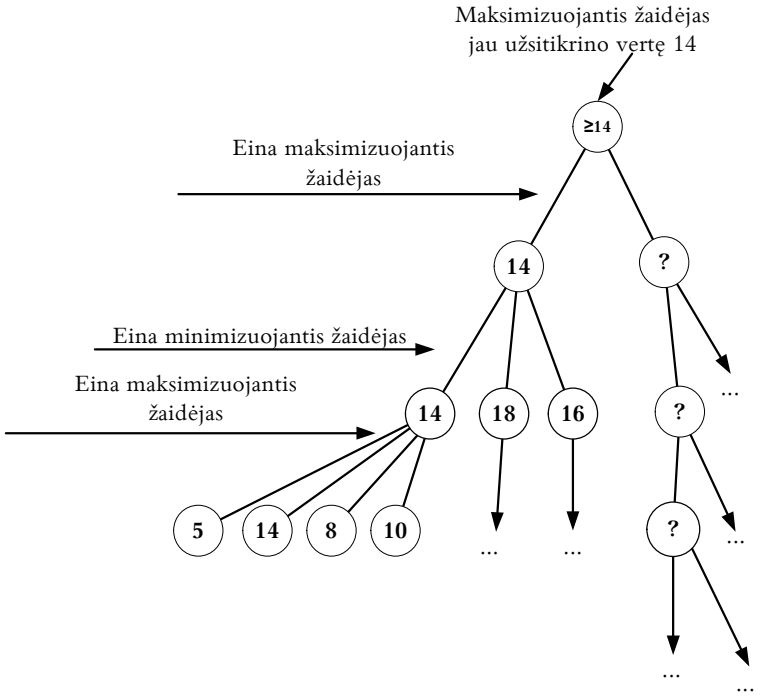
97 pav. Pavyzdys, kai nebūtina analizuoti visų žaidimo pozicijų; laiminti pirmojo žaidėjo pozicija pažymėta L raide, pralaiminti – P raide

Matome, kad dar neišanalizavus viso žaidimo medžio galima nustatyti, kurį ėjimą pasirinks vienas ar kitas žaidėjas ir nebeanalizuoti dalies ėjimų. Kitaip sakant galima *atkirsti* kai kurias žaidimo medžio šakas. Atkirtimas gali būti taikomas ir tiems žaidimų medžiams, kuriuos pavyksta pilnai išskleisti, ir tiems, kurių pozicijos vertinamos euristiškai.

Atkirtimą realizuoja *Alfa-Beta* algoritmas, kuris grindžiamas tokia idėja: jei jau rastas neblogas ėjimas *e*, ir matosi, kad kitas šiuo metu analizuojamas ėjimas nuves į blogesnę poziciją, nei būtų galima

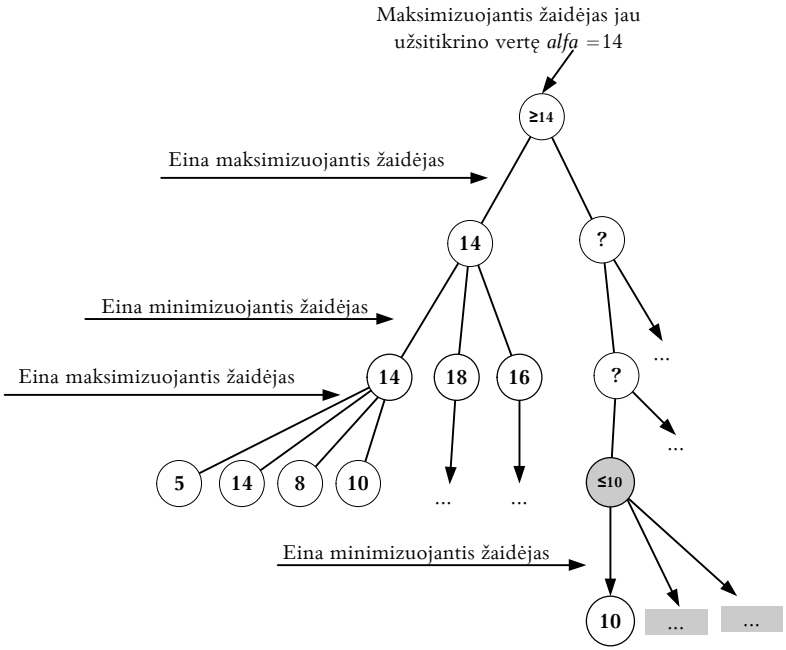
pasiekti pasirinkus *e*, tuomet pozicijos, (ir tolesnių ėjimų iš jos) į kurią pakliūnama tuo kitu ėjimu, galima nebenagrinėti.

Pastebėsime, kad *Alfa-Beta* algoritmas savo esme yra *Minimax* algoritmas, papildytas dviem atkirtimo kriterijais. *Minimax* algoritmas analizuoja visus galimus ėjimus iš konkrečios pozicijos ir parenka ėjimą su palankiausiu įverčiu, o *Alfa-Beta* atkirtimo algoritmas neanalizuoja ėjimų, jei mato, kad jų įverčiai bus prastesni už palankiausią iki šiol rastą įvertį.



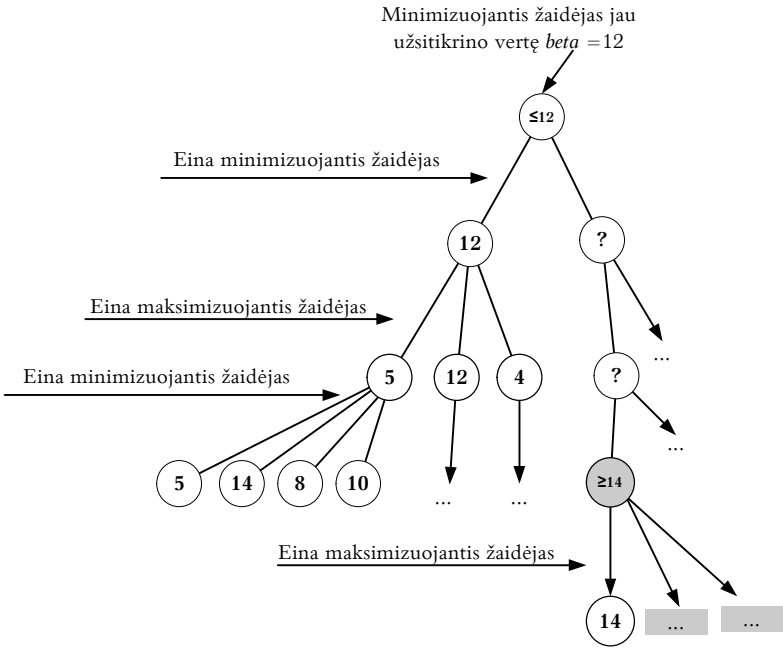
98 pav. Maksimizuojantis žaidėjas gali parinkti tokią ėjimų seką, kad nesvarbu kaip žaistų jo priešininkas, maksimizuojantis žaidėjas pasieks poziciją, kurios vertė lygi ≥ 14, t. y. jis užsitikrino vertę 14

Sutarsime sakyti, kad žaidėjas *užsitikrino* vertę *v*, jeigu jis gali parinkti tokią ėjimų seką, kad nesvarbu kaip žaistų jo priešininkas, žaidėjas pasieks poziciją, kurios vertė lygi *v* arba dar palankesnė, t. y. didesnė už *v*, jei tai maksimizuojantis žaidėjas arba mažesnė už *v*, jei tai minimizuojantis žaidėjas (98 pav.).



99 pav. Maksimizuojantis žaidėjas gali parinkti tokią ėjimų seką, kad nesvarbu kaip žaistų jo priešininkas, maksimizuojantis žaidėjas pasieks poziciją, kurios vertė lygi ≥ 14, t. y. jis užsitikrino vertę 14

Alfa-Beta algoritmas operuoja parametrais *alfa* ir *beta*. Parametre *alfa* saugoma minimali vertė, kurią jau užsitikrino maksimizuojantis žaidėjas konkrečiai pozicijai *P*, o parametre *beta* – maksimali vertė, kurią tai pačiai pozicijai užsitikrino minimizuojantis žaidėjas. Pradiniu momentu *alfa* reikšmė lygi *min_vertė*, o *beta* – *max_vertė*.



100 pav. Minimizuojantis žaidėjas gali parinkti tokią ėjimų seką, kad nesvarbu kaip žaistų jo priešininkas, minimizuojantis žaidėjas pasieks poziciją, kurios vertė lygi ≤ 12, t. y. jis užsitikrino vertę 12

Kadangi maksimizuojantis žaidėjas renkasi ėjimus su kuo didesniu įverčiu, o minimizuojantis – su kuo mažesniu, tai maksimizuojantis žaidėjas siekia, kad *alfa* reikšmė būtų kuo didesnė, o jo oponentas – kad *beta* reikšmė būtų kuo mažesnė.

Sakykime, jau išnagrinėta dalis žaidimo medžio ir maksimizuojantis žaidėjas užsitikrino *alfa* vertę. Tačiau tikintis dar geresnio rezultato, nagrinėjama ir likusi medžio dalis.

Tarkime, kad bet kur toliau medyje minimizuojančiam žaidėjui atlikus ėjimą iš pozicijos P pakliūta į poziciją P', o pastarosios vertė

$v_{P'}$ yra mažesnė už *alfa*. Tuomet akivaizdu, kad minimizuojantis žaidėjas iš pozicijos P rinkdamasis ėjimą su kuo mažesne verte pasieks, kad $v_P < alfa$. Taigi pozicija P maksimizuojančiam žaidėjui nebeįdomi ir likusių ėjimų iš P galima nebenagrinėti (99 pav.).

Tarkime, kad bet kur toliau medyje maksimizuojančiam žaidėjui atlikus ėjimą iš pozicijos P pakliūta į poziciją P', o pastarosios vertė $v_{P'}$ yra didesnė už *beta*, tuomet akivaizdu, kad maksimizuojantis žaidėjas iš pozicijos P rinkdamasis ėjimą su kuo didesne verte pasieks, kad $v_P > beta$. Taigi pozicija P minimizuojančiam žaidėjui nebeįdomi ir likusių ėjimų iš P galima nebenagrinėti (100 pav.).

```
Alfa-Beta(gylis, pozicija, žaidėjas)
{ „gylis“ nurodo iki kokio lygio skleisime žaidimo medį
  „pozicija“ parodo nuo kokios pozicijos analizuosime žaidimą }
    jei žaidėjas yra maksimizuojantis
        tai grąžink
            Alfa-Beta-Max(gylis, pozicija, MAX_VERTĖ)
            { perduodama MAX_VERTĖ, taigi šioje pozicijoje nebus
              vykdomas atkirtimas }
        kitu atveju grąžink
            Alfa-Beta-Min(gylis, pozicija, MIN_VERTĖ)
            { perduodama MIN_VERTĖ, taigi šioje pozicijoje nebus
              vykdomas atkirtimas }

Alfa-Beta-Max(gylis, pozicija, beta)
    jei žaidimas baigtas arba (gylis = 0)
        tai grąžink įvertinimas(pozicija, mini_zaid)
    kitu atveju
        alfa = MIN_VERTĖ // alfa – lokalus kintamasis
        kiekvienam galimam ėjimui e
            atlik ėjimą(e, pozicija)
            įvertis := Alfa-Beta-Min(
                gylis - 1, pozicija, alfa)
            atšauk ėjimą(e, pozicija)
            jei beta <= įvertis
                tai grąžink alfa // atliekamas atkirtimas
            jei įvertis > alfa
                tai alfa = įvertis
        grąžink alfa
```

```
Alfa-Beta-Min(gylis, pozicija, alfa)
{ alfa ir beta yra lokalūs kintamieji }
    jei žaidimas baigtas arba (gylis = 0)
       tai grąžink įvertinimas(pozicija, max_zaid)
    kitu atveju
        beta = MAX_VERTĖ // beta — lokalus kintamasis
        kiekvienam galimam ėjimui e
            atlik ėjimą(e, pozicija)
            įvertis := Alfa-Beta-Max(
                gylis - 1, pozicija, beta)
            atšauk ėjimą(e, pozicija)
            jei įvertis <= alfa
                tai grąžink beta // atliekamas atkirtimas
            jei įvertis < beta
                tai beta = įvertis
        grąžink beta
```

Alfa-Beta atkirtimo efektyvumas labai priklauso nuo to, kokia tvarka nagrinėjami ėjimai. Jeigu visą laiką pirmiau aptinkami blogaiausi ėjimai, tokiu atveju algoritmas veiks lygiai taip pat, kaip ir *Minimax* algoritmas ir iš esmės atkirtimas niekada nebus atliktas. Todėl tvarka, kuria peržiūrimi ėjimai labai svarbi atliekant *Alfa-Beta* atkirtimą.

LITERATŪROS ŠALTINIAI

1. Brassard G., Bratley P. (1996). *Fundamentals of Algorithms*. Prentice Hall, New Jersey.
2. Cormen T. H., Leiserson C. E., Rivest R. L. (1990). *Introduction to Algorithms*. MIT Press, Cambridge.
3. Eppstein D. (1997). *Strategy and board game programming*. http://www.ics.uci.edu/~eppstein/180a/w99.html.
4. Graham R. L., Knuth D. E., Patashnik O. (1990). *Concrete mathematics*. Addison-Wesley.
5. Kubilius J. (1980). *Tikimybių teorija ir matematinė statistika*. Mokslas, Vilnius.
6. Skienna S. S. (1998). *The Algorithm Design Manual*. Springer-Verlag, New York.
7. Skienna S. S., Revilla M. A. (2003). *Programming Challenges*. Springer.
8. Tannenbaumas P., Arnoldas R. (1995). *Kelionė į šiuolaikinę matematiką*. TEV, Vilnius.
9. Vilenkinas N. (1979). *Kombinatorika*. Šviesa, Kaunas.
10. Žilinskas A., Leonavičius D., Valavičius E. (2000). *Informatika*. Aldorija, Vilnius.

RODYKLĖ

E

F

G

H

I

J

K

L

M

N

O

P

R

S

Ž