

1 skyrius

Algoritmų sudarymo būdai

1.1. Algoritmų sudėtingumo analizė

Algoritmas yra tiksliai apibrėžta skaičiavimo procedūra, kuria, imdami pradinius duomenis ir atlikę baigtinį skaičių operacijų, gauname rezultatus. Skaičiavimo procedūrą galime suprasti kaip kompiuterio programą, užrašytą viena iš programavimo kalbų.

Pradinių duomenų skaičius yra labai svarbi uždavinio charakteristika, nes, aišku, kad kuo daugiau duomenų, tuo daugiau kompiuterio atminties reikia duomenų saugojimui ir, dažniausiai, ilgiau vykdoma skaičiavimo programa. Pavyzdžiui:

- vektoriaus X duomenų dydis yra lygus jo elementų skaičiui n ,
- matricos A , turinčios m eilučių ir n stulpelių, dydis yra mn ,
- grafo $G = (V, E)$, kurio viršūnių V skaičius n , o briaunų aibės E dydis m , pradinių duomenų skaičius yra $m + n$.

Nors duomenų skaičius ir charakterizuoja uždavinio dydį, tačiau vien tik šis skaičius dar neapibūdina algoritmo sudėtingumo. Nagrinėkime du svarbius matricių veiksmus: dviejų matricių sumos $A + B$ ir sandaugos AB skaičiavimą. Tegul matricių dydis yra n eilučių ir n stulpelių, t.y. turime $n \times n$ dydžio matricas.

Nagrinėkime matricių sumos $A + B$ skaičiavimo algoritmą

$$C = A + B, \quad C = (c_{ij}), \quad 1 \leq i, j \leq n,$$

$$c_{ij} = a_{ij} + b_{ij}.$$

Taigi viso atliekame n^2 sumavimo veiksmų. Šį kartą veiksmų skaičius yra tos pačios eilės dydis, kaip ir pradinių duomenų skaičius (matricų koeficientų yra $2n^2$).

Dabar nagrinėkime dviejų matricių sandaugos $C = AB$ skaičiavimo algoritmą

$$c_{ij} = \sum_{k=1}^n a_{ik}b_{kj}, \quad 1 \leq i, j \leq n.$$

Viso atliekame n^3 daugybos ir $n^2(n-1)$ sumavimo veiksmų, arba $2n^3 - n^2$ aritmetinių veiksmų.

Taigi matome, kad dviejų matricių daugybos veiksmas yra sudėtingesnis už jų sumos skaičiavimą. Abiejų operacijų sudėtingumą įvertinome panaudodami kiekybinį matą – aritmetinių veiksmų skaičių. Tačiau ne visiems algoritmams tinka šis matas, pavyzdžiui duomenų rūšiavimo ir paieškos algoritmuose svarbiausi yra duomenų palyginimo ir sekos elementų keitimo vietomis veiksmai. Todėl naudosime tokį bendrą apibrėžimą.

Algoritmo sudėtingumas yra lygus to algoritmo bazinių veiksmų skaičiui.

Svarbu, kad taip apibrėžtas algoritmo sudėtingumas nepriklauso nuo konkretaus kompiuterio ypatybių. Tada uždavinio dydžiu (arba apimtimi) vadiname geriausio žinomo jo sprendimo algoritmo sudėtingumą.

Nagrinėdami bet kokį algoritmą, įvertiname atliekamų aritmetinių veiksmų skaičių arba bent svarbiausią jų dalį. Tada, remdamiesi šia informacija, tiksliai prognozuojame algoritmo realizacijos laiką. Pavyzdžiui, žinome, kad, Gauso algoritmu sprendami N tiesinių lygčių sistemą, atliekame $\frac{2}{3}N^3 + O(N^2)$ aritmetinių veiksmų. Remdamiesi šia formule, įvertiname, jog, sprendami dvigubai didesnę $2N$ tiesinių lygčių sistemą, sugaišime aštuonis kartus daugiau laiko, ir šis teiginys yra teisingas kiekvienam kompiuteriui. Be to, jei sudarysime kitą algoritmą, kurio aritmetinių veiksmų įvertis yra $\frac{1}{2}N^3 + O(N^2)$, tai galime numatyti, jog pakankamai didelę tiesinių lygčių sistemą šiuo algoritmu išspręsime $\frac{4}{3}$ karto greičiau nei Gauso algoritmu.

1.1.1. Algoritmo sudėtingumo įverčių tipai

Tarkime, kad algoritmo T pradinių duomenų skaičius yra n . Tada jo sudėtingumą žymėsime $T(n)$. Tačiau algoritmo sudėtingumas dažnai priklauso ne tik nuo pradinių duomenų skaičiaus, bet ir nuo šių duomenų pasiskirstymo. Pavyzdžiui, rūšiuojant skaičių masyvą, algoritmo vykdymo trukmė smarkiai skirsis, kai pradiniai duomenys yra atsitiktinai pasiskirstę ir kai jie jau beveik surūšiuoti.

Pažymėkime D_n visų pradinių duomenų variantų aibę, kai kiekviename variante yra n duomenų ir jie priklauso A aibei:

$$D_n = \{ d_n = (a_1, a_2, \dots, a_n), \quad a_j \in A, \quad j = 1, 2, \dots, n \}.$$

Tada apibrėžiame tris svarbius algoritmo sudėtingumo įverčius:

- geriausiojo atvejo sudėtingumą

$$T_G(n) = \min_{d_n \in D_n} T(d_n),$$

- blogiausiojo atvejo sudėtingumą

$$T_B(n) = \max_{d_n \in D_n} T(d_n),$$

- vidutinį (arba tikėtiną) algoritmo sudėtingumą

$$T_V(n) = \sum_{d_n \in D_n} p(d_n) T(d_n),$$

čia $p(d_n)$ yra pradinių duomenų d_n atvejo tikimybė. Aišku, kad teisinga tokia normavimo sąlyga

$$\sum_{d_n \in D_n} p(d_n) = 1.$$

Dažniausiai yra sunku įvertinti kiekvieno duomenų pasiskirstymo pasirodymo tikimybę, tada remsimės prielaida, kad visi atvejai yra vienodai tikėtini

$$p(d_n) = \frac{1}{|D_n|}, \quad d_n \in D_n.$$

Asimptotiniai įverčiai

Pateiksime standartinius asimptotinių įverčių žymėjimus.

Tegul $f = f(n)$ ir $g = g(n)$ yra dvi funkcijos, kurios yra apibrėžtos natūrinių skaičių aibėje, o jų reikšmės visada yra teigiami skaičiai.

Asimptotinis viršutinis įvertis

Sakysime, kad $f(n) = \mathcal{O}(g(n))$ (skaitome "o-didysis nuo $g(n)$ "), jei egzistuoja tokia teigiama konstanta c ir toks natūrinis skaičius n_0 , kad galioja nelygybės:

$$0 \leq f(n) \leq c g(n), \quad n \geq n_0.$$

Asimptotinis viršutinis įvertis reiškia, kad funkcija $f(n)$ didėja negreičiau, nei funkcija $c g(n)$.

Pavyzdžiui, imkime $f(n) = (n + 2)^2$ ir parodykime, kad $f = \mathcal{O}(n^2)$. Įsitinkime, kad

$$f(n) \leq 4n^2, \quad \text{kai } n \geq 2.$$

Nesunku patikrinti (pvz. matematinės indukcijos metodu), kad

$$4(n + 1) \leq 3n^2, \quad \text{kai } n \geq 2,$$

todėl gauname tokius įverčius

$$f(n) = n^2 + 4n + 4 \leq n^2 + 3n^2 \leq 4n^2.$$

Konstantą c galime ir sumažinti, jei padidinsime n_0 , pavyzdžiui

$$4(n + 1) \leq n^2, \quad \text{kai } n \geq 5,$$

todėl

$$f(n) = n^2 + 4n + 4 \leq n^2 + n^2 \leq 2n^2, \quad n \geq 5.$$

Tačiau asimptotinio įverčio egzistavimui yra svarbu tik tai, kad radome konstantas c ir n_0 .

Tokie įverčiai nebūtinai yra vieninteliai, pavyzdžiui teisingas ir įvertis

$$(n + 2)^2 = \mathcal{O}(n^3),$$

tačiau jis yra grubesnis už anksčiau gautąjį įvertį.

Algoritmų sudėtingumo asimptotiniai įverčiai yra svarbūs, kai duomenų skaičius n yra didelis. Tada rinksimės tokius algoritmus, kurių sudėtingumo funkcija didėja lėčiau. Jei duomenų yra nedaug, tai greitesnis gali būti ir algoritmas, kurio asimptotinis įvertis yra blogesnis. Pavyzdžiui nagrinėkime dvi funkcijas

$$f(n) = 20n, \quad g(n) = \frac{1}{5}n^2.$$

Aišku, kad $f = \mathcal{O}(n)$ yra lėčiau didėjanti funkcija, nei $g = \mathcal{O}(n^2)$, bet

$$\frac{1}{5}n^2 \leq 20n, \quad \text{kai } n \leq 100.$$

Asimptotinis apatinis įvertis

Sakysime, kad $f(n) = \Omega(g(n))$ (skaitome "omega-didžioji nuo $g(n)$ "), jei egzistuoja tokia teigiama konstanta c , kad be galo dideliame skaičiui skirtingų natūrinių n galioja nelygybės:

$$f(n_j) \geq c g(n_j), \quad j = 1, 2, \dots, \infty,$$

čia $n_{j+1} > n_j$. Jeigu asimptotinis viršutinis rėžis parodo algoritmo skaičiavimo apimties viršutinį rėžį, tai apatinis rėžis padeda įvertinti neišvengiamus algoritmo realizacijos kaštus.

Kai kada ir asimptotinio apatinio įverčio apibrėžime yra reikalaujama, kad nelygybė galiotų visoms pakankamai didelėms n reikšmėms:

$$f(n) \geq c g(n), \quad n \geq n_0.$$

Tačiau tada negalime gauti gero apatinio įverčio daugeliui algoritmų, kurių skaičiavimo apimtis yra daug mažesnė daliai pradinių duomenų, bet ne visiems.

1.1 pavyzdys. Asimptotinio apatinio įverčio radimas. Tarkime, kad algoritmo sudėtingumo funkcija yra tokia:

$$T(n) = \begin{cases} n + 7, & n = 2k, \quad k = 1, 2, \dots, \\ n^2 + n - 3, & n = 2k + 1. \end{cases}$$

Tada gauname, kad $T(n) = \Omega(n^2)$ pirmojo apibrėžimo prasme, bet naudodami antrąjį apibrėžimą galime įrodyti tik silpnesnį įvertį $T(n) = \Omega(n)$.

Asimptotiškai griežtas įvertis

Sakysime, kad $f(n) = \Theta(g(n))$, jei egzistuoja tokios teigiamosios konstantos c_1, c_2 ir toks natūralusis skaičius n_0 , kad galioja nelygybės:

$$0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n), \quad n \geq n_0.$$

Nykstamai maža funkcija

Sakysime, kad $f(n) = o(g(n))$ (skaitome "o-mažoji nuo $g(n)$ "), jei bet kokiai teigiamajai konstantai c egzistuoja toks natūralusis skaičius n_0 , kad galioja nelygybės:

$$0 \leq f(n) < c g(n), \quad n \geq n_0.$$

Ekvivalentų apibrėžimą gauname naudodami funkcijų ribas:

$$\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = 0.$$

1.1.2. Algoritmo sudėtingumo įvertinimo metodai

Sudarydami daugelio uždavinių sprendimo algoritmus naudojame rekursijos ir uždavinio dalijimo į mažesnius uždavinius metodus. Tirdami tokių algoritmų sudėtingumą turime spręsti tiesines skirtumų lygtis. Susipažinsime su svarbiausiais tokių lygčių sprendimo metodais.

Nagrinėkime algoritmą, kurio sudėtingumo funkcija yra

$$T(n) = \begin{cases} c, & \text{jei } n = 1, \\ T(n-1) + d, & \text{jei } n > 1. \end{cases}$$

Tokiu algoritmu skaičiuojame faktorialo $n!$ reikšmės, naudodami rekursiją.

Gautąją lygtį spręskime lygties eilės mažinimo metodu. Įstatę pagrindinę formulę vietoj $T(n-1)$, gauname lygybę:

$$T(n) = T(n-1) + d = T(n-2) + 2d.$$

Tęsdami šį procesą įvertiname $T(n)$:

$$T(n) = T(1) + (n-1)d = (n-1)d + c,$$

taigi įrodėme, kad $T(n) = \mathcal{O}(n)$.

Mažoruojančių funkcijų metodas. Dažnai pavyksta rasti tik funkciją $f(n)$, kuri įvertina $T(n)$ iš viršaus:

$$T(n) \leq f(n), \quad \text{kai } n \geq n_0.$$

Toks metodas vadinamas *mažoruojančių funkcijų* metodu.

Lygties eilės mažinimo metodas. Tai gana bendras metodas, jis leidžia užrašyti bendrąjį skirtumų lygčių sprendinį baigtinių sumų pavidalu. Nagrinėkime algoritmą, kurio sudėtingumo funkcija yra:

$$T(n) = \begin{cases} c, & \text{jei } n = 1, \\ aT\left(\frac{n}{b}\right) + d(n), & \text{jei } n > 1. \end{cases}$$

čia a, b ir c yra konstantos, o $d(n)$ – funkcija. Tokie algoritmai gaunami, pvz. *skaldyk ir valdyk* metodu, kurį suformuluosime kitame skirsnyje ir dažnai naudosime sprenddami įvairius uždavinius. Šiuo metodu n dydžio uždavinį išskaidome į a mažesnių $\frac{n}{b}$ dydžio uždavinių, kuriuos išsprendę ir atlikę papildomai $d(n)$ veiksmų randame viso uždavinio sprendinį.

Tarsime, kad $n = b^m$, nes kaip tik tokie yra n , kai naudojame skaldyk ir valdyk metodu sukonstruotus algoritmus. Pertvarkykime lygtį mažindami jos eilę, tuo tikslu įrašykime $T(\frac{n}{b})$ nario išraišką iš pagrindinės lygties, šį procesą kartokime $(m - 1)$ kartą:

$$\begin{aligned} T(n) &= aT\left(\frac{n}{b}\right) + d(n) = a\left(aT\left(\frac{n}{b^2}\right) + d\left(\frac{n}{b}\right)\right) + d(n) \\ &= a^2T\left(\frac{n}{b^2}\right) + ad\left(\frac{n}{b}\right) + d(n) \\ &= a^3T\left(\frac{n}{b^3}\right) + a^2d\left(\frac{n}{b^2}\right) + ad\left(\frac{n}{b}\right) + d(n) = \dots \\ &= ca^m + \sum_{j=0}^{m-1} a^j d(b^{m-j}). \end{aligned}$$

Gavome bendrąją lygties sprendinį. Išnagrinėsime kelis atskirus atvejus, dažnai sutinkamus įvertinant įvairių algoritmų sudėtingumą.

Tarkime, kad $a = 2$ ir $b = 2$, o funkcija $d(n) = gn$:

$$T(n) = 2T\left(\frac{n}{2}\right) + gn.$$

Tada gauname tokią algoritmo sudėtingumo funkciją

$$T(n) = cn + \sum_{j=0}^{m-1} gn = cn + gnm = cn + gn \log n. \quad (1.1)$$

Asimptotinio įverčio $T(n) = \mathcal{O}(n \log n)$ pagrindinį narį apibrėžė nehomogeninė funkcija $d(n) = gn$, kuri įvertino atskirų sprendinių sujungimo kaštus.

Tarkime, kad funkcija $d(n)$ didėja lėčiau už tiesinę, t.y. $d(n) = gn^\alpha$, $\alpha < 1$. Tada ir viso algoritmo sudėtingumo funkcija yra tiesinė:

$$\begin{aligned} T(n) &= cn + g 2^{m\alpha} \sum_{j=0}^{m-1} 2^{(1-\alpha)j} = cn + g 2^{m\alpha} \frac{2^{(1-\alpha)m} - 1}{2^{(1-\alpha)} - 1} \\ &= cn + g \frac{n - n^\alpha}{2^{(1-\alpha)} - 1} = \mathcal{O}(n). \end{aligned}$$

Jeigu $a = b^\beta$, o $\beta > 1$ ir $d(n) = gn$, tai tada algoritmo sudėtingumą lemia koeficientas a (patikrinkite šį teiginį):

$$T(n) = \mathcal{O}(n^\beta).$$

Rakite algoritmo sudėtingumo funkciją, kai ir rezultatų sujungimo kaštai didėja netiesiškai $d(n) = gn^\beta$.

1.2. Variantų perrinkimas

Spręsdami daugelį uždavinių naudojame bendrą principą – uždavinį dalijame į mažesnes užduotis, kurias išsprendę gauname viso uždavinio sprendinį. Variantų perrinkimo principas ypač išpopuliarėjo, kai atsirado kompiuteriai. Naudodami šį metodą susiduriame su dviem svarbiausiais uždaviniais:

1. Kaip padalinti uždavinį į baigtinį skaičių mažesnių užduočių (variantų).
2. Kaip sumažinti nagrinėjamų variantų skaičių, nes tiesioginis visų variantų patikrinimas gali būti neįvykdomas net su greičiausiais superkompiuteriais.

Antrąją problemą sprendžiame naudodami *dinaminio programavimo, šakų ir režijų* metodus, kuriuos aptarsime kituose poskyriuose. Šiame poskyryje nagrinėsime tik vieną pavyzdį, kai svarbiausia yra sudaryti visų variantų aibę, o variantų perrinkimas yra atliekamas greitai.

1.2 pavyzdys. Kaip sužinoti Petro vaikų amžių? Susitiko du senokai nesimatę draugai – Jonas ir Petras. Pokalbio metu paaiškėjo, kad ši diena yra ypatinga Petrui, nes visi trys jo vaikai, Inga, Julija ir Justas, šiandien švenčia savo gimtadienius. Petras pasiūlė Jonui, geram matematikos žinovui, pabandyti atspėti, koks yra kiekvieno vaiko amžius.

Norėdamas palengvinti užduotį jis nurodė, kad Justas yra nejaunesnis už seseris, o Inga neturi jaunesnės sesers. Taip pat Petras pasakė, kad sudauginę visų trijų vaikų metus gauname skaičių 36. Šiek tiek pagalvojęs Jonas pareiškė, kad jam dar neužtenka informacijos. Tada Petras nurodė, kad vaikų metų suma sutampa su namo, prie kurio jie stovi, langų skaičiumi. Jonas vėl pagalvojo ir pasakė, kad naujoji informacija tikrai labai svarbi, bet jos visgi dar nepakanka, kad galėtų

pasakyti atsakymą. Todėl reikėtų mažos pagalbos. Naujoji Petro pastaba buvo trumpa: vyriausiojo vaiko akys yra mėlynos. Sužinojęs tai, Jonas iš karto pasakė kiekvieno vaiko amžių!

Pabandykime ir mes išspręsti šį uždavinį. Iš pirmosios sąlygos sužinojome, kad trijų vaikų metų sandauga yra lygi 36. Nesunku patikrinti, kad yra tik aštuoni skirtingi variantai, kai išpildyta ši sąlyga, jie pateikti 1.1 lentelėje.

1.1 lentelė. Aštuoni variantai, kai vaikų metų sandauga yra lygi 36

| Vardas | V1 | V2 | V3 | V4 | V5 | V6 | V7 | V8 |
|--------|----|----|----|----|----|----|----|----|
| Inga | 1 | 1 | 1 | 1 | 1 | 2 | 2 | 3 |
| Julija | 1 | 2 | 3 | 4 | 6 | 2 | 3 | 3 |
| Justas | 36 | 18 | 12 | 9 | 6 | 9 | 6 | 4 |

Kadangi turime padaryti prielaidą, kad Jonas žino, kiek langų turi namas, prie kurio susitiko draugai, tai iš antrosios sąlygos jis sužinojo ir kam lygi vaikų metų suma. Tačiau tokios informacijos jam vis dar neužteko, kad galėtų pasakyti atsakymą. Apskaičiuokime kiekvieno varianto vaikų metų sumą:

$$\begin{aligned}
 1 + 1 + 36 &= 38, & 1 + 2 + 18 &= 21, \\
 1 + 3 + 12 &= 16, & 1 + 4 + 9 &= 14, \\
 1 + 6 + 6 &= 13, & 2 + 2 + 9 &= 13, \\
 2 + 3 + 6 &= 11, & 3 + 3 + 4 &= 10.
 \end{aligned}$$

Dabar tampa aišku, kad ši metų suma yra lygi 13, nes visais kitais atvejais, pvz. jei vaikų metų suma būtų lygi 14 ar 21, Jonas jau žinotų ir kiekvieno vaiko amžių. Liko du variantai – (1, 6, 6) ir (2, 2, 9). Kadangi tik antruoju atveju Justas yra vyriausias vaikas (tai, kad jo akys mėlynos, aišku, neturi jokios reikšmės), darome išvadą, kad Petras augina dvynukes Ingą ir Juliją, kurioms sukako dveji metukai, ir devynių metų sūnų Justą.

1.3. Rekursijos metodas

Rekursija yra patogus daugelio matematinių objektų apibrėžimo būdas, ji plačiai naudojama ir sudarant bei realizuojant įvairius algoritmus. Nesiekdami griežtumo sakysime, kad objektas yra apibrėžtas rekursijos būdu, jei apibrėžime vėl naudojamas tas pats objektas.

1.3 pavyzdys. Faktorialas. Natūraliojo skaičiaus $n \in \mathbb{N}$ faktorialas yra apibrėžiamas taip

$$n! = \begin{cases} n \cdot (n-1)!, & \text{jei } n > 0, \\ 1, & \text{jei } n = 0. \end{cases}$$

Iš šio apibrėžimo gauname tokią lygybę

$$n! = n \cdot (n-1) \cdot (n-2) \cdots 2 \cdot 1.$$

1.4 pavyzdys. Fibonačio skaičiai. Fibonačio skaičiai pirmą kartą buvo panaudoti sudarant kiškių populiacijos matematinius modelius. Šį uždavinį 13 amžiaus pradžioje nagrinėjo italų matematikas Leonardas Fibonačis (it. *Leonardo Fibonacci*). Skaičiai sutinkami įvairiuose informatikos algoritmuose bei algoritmų sudėtingumo analizėje. Jie apibrėžiami tokiu būdu:

$$f_n = \begin{cases} f_{n-1} + f_{n-2}, & \text{jei } n > 1, \\ f_0 = 1, f_1 = 1. \end{cases}$$

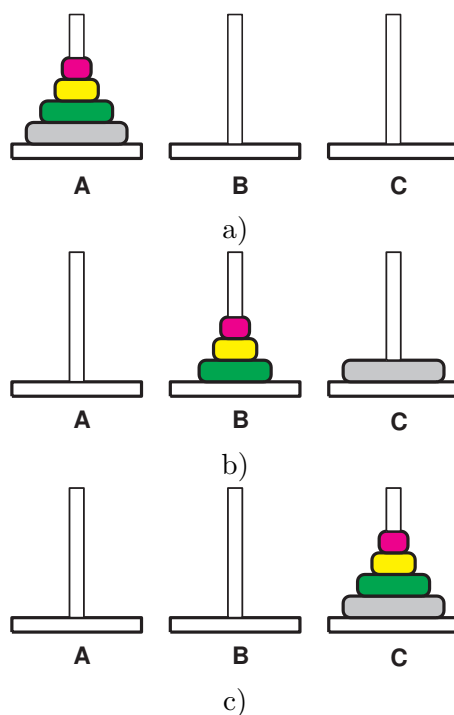
Abiejuose pavyzdžiuose matome tipiską rekursijos struktūrą:

- objektas, priklausantis nuo parametro, yra apibrėžiamas naudojant tą patį objektą ar objektus, tik su kitomis parametro reikšmėmis;
- nurodoma rekursijos pabaiga ir užduodamos pradinės sąlygos, šių sąlygų skaičius sutampa su rekursijos gyliu.

Rekursiją naudosime sudarydami duomenų rūšiavimo, informacijos paieškos algoritmus, ja grindžiami dinaminiai duomenų struktūrų pagrindiniai metodai.

1.3.1. Rekursijos algoritmų analizė

Šiame paragrafe pateiksime pavyzdžių iš kitų informatikos, skaičiuojamosios matematikos sričių, taip pat parodysime, kaip rekursija padeda išspręsti loginių žaidimų ir galvosūkių užduotis.



1.1 pav. Hanojaus bokštai: a) pradinė padėtis, b) žiedų padėtis po antrojo žingsnio, c) žaidimas baigtas

Hanojaus bokštai. Jau senovės Kinijoje buvo sprendžiamas toks žaidimas: turime tris virbus A , B ir C . Ant A virbo yra suverti n skirtingo diametro žiedai: apačioje yra didžiausias žiedas, ant jo užėtas mažesnis ir taip toliau iki mažiausio, kuris patalpintas viršuje. Šiuos žiedus reikia perkelti ant kito virbo C , kai ant viršaus galima dėti tik mažesnio diametro žiedą (žr. 1.1a paveikslą).

Šio žaidimo strategiją labai patogiu apibrėžti naudojant rekursiją:

1. Pirmiausia laikydamiesi taisyklių perkeliame $(n - 1)$ viršutinius A žiedus ant B virbo, o C virbu naudojames kaip pagalbinį.
2. Tada perkeliame nuo A ant C paskutinį didžiausią žiedą (žr. 1.1b paveikslą).
3. Paskutiniame žingsnyje tuo pačiu būdu perkeliame žiedus nuo B ant C virbo, o A naudojame kaip pagalbinį (žr. 1.1c paveikslą).

Algoritmo struktūra yra labai paprasta ir akivaizdi, ji pateikta 1.2 paveiksle.

Hanojaus bokštų uždavinio sprendimo algoritmas

```
HanojausBokštai (n, A, B, C)
(1) if ( n > 0 )
(2)   HanojausBokštai (n-1, A, C, B);
(3)   move (A, C);
(4)   HanojausBokštai (n-1, B, A, C);
(5) end if
end HanojausBokštai
```

1.2 pav. Hanojaus bokštų uždavinio sprendimo algoritmas

1.5 pavyzdys. Trijų žiedų uždavinio analizė. Naudodami rekursinį algoritmą, tris žiedus perkeliame tokia tvarka:

1. Žiedas nuo A virbo perkeliamas ant C virbo.
2. Žiedas nuo A virbo perkeliamas ant B virbo.
3. Žiedas nuo C virbo perkeliamas ant B virbo.
4. Žiedas nuo A virbo perkeliamas ant C virbo.
5. Žiedas nuo B virbo perkeliamas ant A virbo.
6. Žiedas nuo B virbo perkeliamas ant C virbo.
7. Žiedas nuo A virbo perkeliamas ant C virbo.

Trijų žiedų uždavinį nesunkiai išspęstume ir nenaudodami rekursinio algoritmo, bet pastarasis efektyviai sprendžia uždavinį ir kai turime daug žiedų.

Yra daug uždavinių, kai rekursija yra patogi apibrėžiant naujus objektus, bet jų realizavimo algoritmai nėra efektyvūs. Todėl svarbu išmokti, kaip kitais būdais galima realizuoti rekursijos funkcijas.

Vėl nagrinėkime faktorialo skaičiavimo užduotį. Tada $n!$ reikšmę galime skaičiuoti rekursiniu algoritmu, pateiktu 1.3 paveiksle.

Rekursinis faktorialo skaičiavimo algoritmas

```
int Faktorialas(n)
(1) if ( n == 0 ) return (1);
    else
(2)   return ( n * Faktorialas(n-1) );
    end if
end Faktorialas
```

1.3 pav. Faktorialo skaičiavimas rekursiniu algoritmu

Kadangi šiame algoritme naudojame lygiai tuos pačius loginius operato-rius, kaip ir faktorialo apibrėžime, tai jo teisingumo tikrinti jau nebereikia. Faktorialo reikšmę galime skaičiuoti ir iteraciniu algoritmu, pateiktu 1.4 pa-veiksle.

Iteracinis faktorialo skaičiavimo algoritmas

```
int Faktorialas2 (n)
(1) s = 1;
(2) for ( i = 2; i <= n ; i++ )
(3)   s = s * i;
    end if
(4) return ( s );
end Faktorialas2
```

1.4 pav. Faktorialo skaičiavimas iteraciniu algoritmu

Abiejuose algoritmuose atliekame po tiek pat aritmetinių veiksmų, tačiau iteracinio algoritmo vykdymo trukmė bus trumpesnė, nes rekursijos papildomieji kaštai yra didesni už ciklo operatoriaus papildomuosius kaštus.

Dabar nagrinėkime rekursinį Fibonačio skaičių radimo algoritmą.

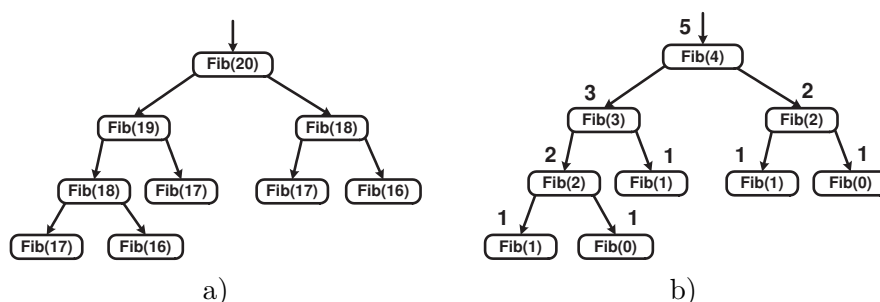
Rekursinis Fibonačio skaičių algoritmas

```

int Fibonaci(n)
(1) if ( n < 2 ) return (1);
    else
(2)  return ( Fibonaci(n-1) + Fibonaci(n-2) );
    end if
end Fibonaci

```

Algoritmo vykdymo eiga yra pavaizduota 1.5 paveiksle, kai $n = 20$ ir $n = 4$. Matome, kad medžio viršūnių reikšmės kartojasi ir todėl skaičiavimų apimtis greitai didėja.



1.5 pav. Fibonačio skaičių radimo algoritmas: a) $n = 20$, b) $n = 4$

Nesunkiai galime įvertinti tokio algoritmo sudėtingumą. Pažymėkime $T(n)$ Fibonačio skaičiaus f_n skaičiavimo kaštus. Bazinė algoritmo operacija laikykime bet kokią aritmetinį veiksmą ir ignoruokime papildomus kaštus, kurie atsiranda realizuojant rekursijos kreipinius. Tada gauname tokią skirtumų lygtį ir pradines sąlygas

$$\begin{cases} T(n) = T(n-1) + T(n-2) + 1, \\ T(0) = 1, \quad T(1) = 1. \end{cases}$$

Tokių lygčių sprendimo metodus nagrinėsime 2.5 paragrafe. Dabar tik užrašysime bendrąjį sprendinį:

$$T(n) = c_1 \left(\frac{1 + \sqrt{5}}{2} \right)^n + c_2 \left(\frac{1 - \sqrt{5}}{2} \right)^n \approx c_1 \left(\frac{1 + \sqrt{5}}{2} \right)^n.$$

Taigi rekursinio algoritmo sudėtingumas yra *eksponentinis*. Padidėjus vienetu duomenų skaičiui n , algoritmo vykdymo eiga pailgėja $(1 + \sqrt{5})/2 =$

1.618 kartą. Taikymuose naudotini *polinominio* sudėtingumo algoritmai, kai $T(n) = \mathcal{O}(n^\alpha)$, o laipsnis α nedidelis.

1.2 lentelėje pateikti skaičiavimo eksperimento rezultatai: T_n yra f_n skaičiavimo laikas, $\rho_n = \frac{T_n}{T_{n-1}}$ yra dviejų gretimų skaičių skaičiavimo laikų santykis.

1.2 lentelė. Fibonačio skaičių rekursyvaus algoritmo sudėtingumo analizė

| n | T_n | ρ_n |
|-----|-------|----------|
| 40 | 2,88 | 1,610 |
| 41 | 4,68 | 1,625 |
| 42 | 7,57 | 1,617 |
| 43 | 12,25 | 1,618 |

Fibonačio skaičius galime rasti ir iteraciniu algoritmu.

Iteracinis Fibonačio skaičių algoritmas

```

Fibonaci2 (n)
(1) s2 = 1;
(2) for ( i = 2; i <= n; i++ )
    (3) s1 = s2;
    (4) s2 = s2 + s1;
    end for
(5) return ( s2 );
end Fibonaci2

```

Tokio algoritmo sudėtingumas tik $T(n) = n$ veiksmų, taigi Fibonačio skaičius randame labai efektyviai. Šis pavyzdys rodo, kad rekursija ne visada naudotina realizuojant algoritmus, kuriais sprendžiame uždavinius apibrėžtus rekursijos būdu.

1.3.2. Kada reikalinga rekursija?

Naudojant rekursiją yra patogu kontroliuoti užduočių atlikimo eiliškumą, kai dalies užduočių vykdymą atidedame vėlesniam laikui. Ji yra nebūtina, kai užduočių medyje bus vykdoma tik viena iš jo šakų, nors konkrečiai, kuris pomedis bus reikalingas paaiškėja tik algoritmo vykdymo metu. Šiuos teiginius iliustruosime dvejetainio paieškos medžio pavyzdžiu.

Dvejetainio medžio viršūnių aplankymo algoritmai. Reikia aplankyti visas dvejetainio medžio viršūnes ir atspausdinti jose saugomus elementus. Tris svarbius algoritmus gauname panaudodami rekursiją. Jeigu medžio viršūnėse užrašyta aritmetinė išraiška, tai šie algoritmai atspausdins *prefix*, *infix* ir *postfix* išraiškos formas.

Pirmajame algoritme pirmiausia atspausdiname šakninėje viršūnėje saugomą informaciją, o po to aplankome kairiąją ir dešiniąją medžio šakas.

Prefix (tiesioginis) algoritmas

```
Prefix (node* tree)
(1) if ( tree != NULL )
      (2) print(tree->data);
      (3) Prefix (tree->left);
      (4) Prefix (tree->right);
      end if
end Prefix
```

Infix arba *vidiniame* algoritme pirmiausia aplankome kairiąją medžio šaką, po to spausdiname šaknies informaciją ir galiausiai aplankome dešiniąją šaką.

Infix algoritmas

```
Infix (node* tree)
(1) if ( tree != NULL )
      (2) Infix (tree->left);
      (3) print(tree->data);
      (4) Infix (tree->right);
      end if
end Infix
```

Postfix arba *atvirkštiniame* algoritme pirmiausia aplankome kairiąją medžio šaką, po to – dešiniąją šaką ir galiausiai spausdiname šaknies informaciją.

Postfix algoritmas

```

Postfix (node* tree)
(1) if ( tree != NULL )
    (2) Postfix (tree->left);
    (3) Postfix (tree->right);
    (4) print(tree->data);
end if
end Postfix

```

Elemento paieška dvejetainiame medyje. Tarkime, kad turime dvejetainį paieškos medį ir reikia patikrinti ar jame saugomas elementas D . Tai galime atlikti naudodami tokį algoritmą:

Elemento paieškos algoritmas

```

Find (node* tree, int D)
(1) while ( (tree != NULL) && (tree->key != D) )
    (2) if (tree->key < D) Find (tree->right, D);
    (3) else Find (tree->left, D);
end while
(4) return (tree);
end Find

```

Algoritme panaudojame rekursiją, tačiau jo analizė rodo, kad kiekvieną kartą iš dviejų medžio šakų pasirenkama tik viena. Todėl efektyvesnė yra tokia algoritmo modifikacija, kurioje jau nenaudojame rekursijos:

Elemento paieškos algoritmas 2

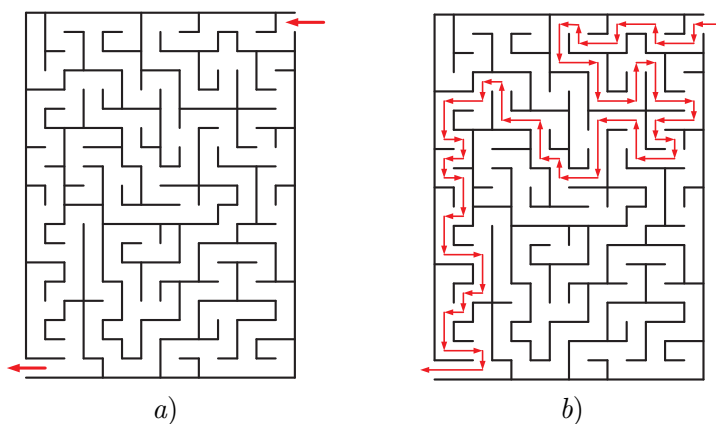
```

Find2 (node* tree, int D)
(1) while ( (tree != NULL) && (tree->key != D) )
    (2) if (tree->key < D) tree = tree->right;
    (3) else tree = tree->left;
end while
(4) return (tree);
end Find2

```

1.3.3. Variantų perrinkimas su grįžimu atgal

Daugeliui iš mūsų teko paklaidžioti pasiklydus miške, nepažįstamame mieste ar laisvalaikiu spręsti uždavinį apie kelio labirinte radimą (žr. 1.6 paveikslą).



1.6 pav. Kaip pereiti per labirintą pradedant viršutiniame dešiniajame kampe ir baigiant kairiajame apatiniame kampe? Paveikslo *a* dalyje pavaizduotas labirintas, *b* dalyje pateiktas kelio pavyzdys.

Šiame skirsnyje spręsimė ne vieną atskirą uždavinį, o sudarysimė algoritmų šabloną *Tikrink* (angl. *template*), tinkamą daugelio panašių uždavinių sprendimui. Duomenų struktūroje pozicija saugome informaciją apie uždavinio sprendimo eigą, n žymi rekursijos gylį, o sprendinio paieška tęsiama iš P taško (pvz. P yra labirinto kambario koordinatės), S yra aibė ėjimų, kuriuos galima atlikti iš P .

Variantų perrinkimo algoritmas

```

int Tikrink (int n, Point P, Inf pozicija)
if ( End (pozicija) == Pabaiga )
    Spausdink (pozicija);
    return(1);
else
    S = BandymųAibė(P, pozicija)
    while ( S != 0 )
        U = NaujasBandymas(S);
        NaujaPozicija(U, pozicija);
        if ( Tikrink (n+1, U, pozicija) == 1 ) return (1);
        else
            SenaPozicija(U, pozicija);
    end while
    return (0);
end if else

```

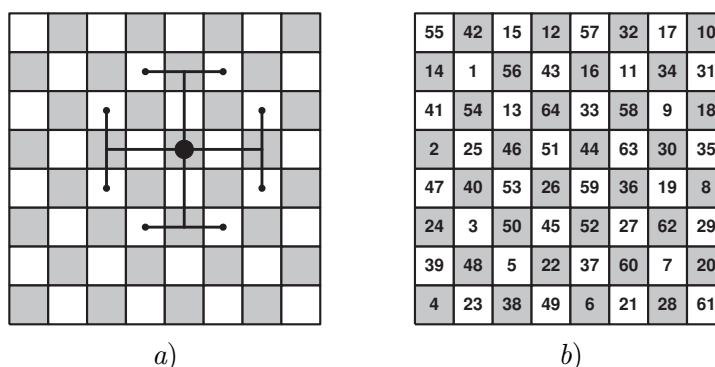
Trumpai aptarsime algoritmo struktūrą. Funkcija BandymųAibė generuoja visus naujus ėjimus, kuriuos galima atlikti iš taško P . Leistinių ėjimų aibę apibrėžia uždavinio sąlygos ir duomenų struktūroje pozicija saugoma informacija apie jau aplankytas paieškos vietas. Kai ši aibė tampa tuščia, tada procedūra Tikrink grąžina reikšmę 0. Algoritmo cikle generuojame naują poziciją ir atliekame sprendinio paiešką šia kryptimi (vykdomas naujas rekursijos kreipinys). Galimos dvi paieškos baigtys:

a) eidami šia kryptimi pasiekiamo paieškos tikslą, t.y. procedūra Tikrink grąžina reikšmę 1, tada šią informaciją perduodame ir kviečiančiajam procesui;

b) paieška yra nesėkminga ir procedūra Tikrink grąžina reikšmę 0, tada algoritme grįžtame vieną žingsnį atgal ir tikriname naują leistiną ėjimą iš taško P .

Pateiksime du pavyzdžius, kai variantų perrinkimas padeda rasti uždavinio sprendinį.

1.6 pavyzdys. Žirgo maršrutas šachmatų lentoje. Turime $n \times n$ dydžio šachmatų lentą, kurios $P = (p, q)$ langelyje stovi žirgas. Leistini žirgo ėjimai yra pavaizduoti 1.7 a paveiksle. Reikia rasti tokį žirgo maršrutą, kai jis apeina visą šachmatų lentą, kiekviename langelyje pabūdamas tik vieną kartą. Žirgo maršruto pavyzdys yra parodytas 1.7 b paveiksle.



1.7 pav. Žirgo maršrutai šachmatų lentoje: a) leistini ėjimai, b) maršruto pavyzdys.

Šis uždavinys yra atskiras atvejis bendresnio *keliaujančio pirklio* uždavinio, kurio sprendimo metodus nagrinėsime kituose vadovėlio skyriuose.

1.7 pavyzdys. Kelio paieška labirinte. Nagrinėkime labirintą, kuris pavaizduotas 1.6 paveikslo *a* dalyje. Reikia pereiti per labirintą pradedant viršutiniame dešiniajame kampe ir baigiant kairiajame apatiniame kampe. Šį kelią nesunkiai randame naudodami variantų perrinkimo su grįžimu atgal algoritmą, maršrutas pavaizduotas paveikslo *b* dalyje.

Euristikos

Kai visų variantų perrinkimas yra neįmanomas dėl per daug didelio jų skaičiaus, naudojame *euristikas*, t.y. algoritmus, leidžiančius greitai patikrinti perspektyvius variantus, bet negarantuojančius tikslaus sprendinio radimo. Kai kuriems uždaviniams euristiniais algoritmais pavyksta rasti optimalius sprendinius, o kitiems uždaviniams gauname gerus sprendinio artinius.

Sprendžiant paieškos ar minimizacijos uždavinius populiarūs euristiniai algoritmai yra gaunami naudojant *godžią* strategiją. Kiekvieną kartą renkames tą variantą ar paieškos kryptį, kuri šiame žingsnyje atneša didžiausią pelną. Aišku, kad tokia lokali pasirinkimo strategija nebūtinai yra optimali viso uždavinio sprendimo atžvilgiu, bet jos realizavimo sąnaudos yra labai mažos.

1.8 pavyzdys. Žirgo maršruto radimas euristiniu algoritmu.

Pasirinkdami eilinį žirgo ėjimą prioritetą teiksime tam langeliui, į kurį patekti yra mažiausiai variantų. Jei tokių langelių yra keli, tai renkames bet kurį iš jų.

Aptarsime algoritmo realizaciją. Priminsime, kad matricos $H(p, q)$ elemente saugojome informaciją, kuriame ėjime žirgas aplankė (p, q) langelį. Dabar papildomai saugosime informaciją ir apie tai, kiek liko variantų, leidžiančių patekti į laisvą lentos langelį (elementui $H(p, q)$ priskirime atitinkamą neigiamą skaičių). Tada naujo žirgo ėjimo paieška (mažiausio skaičiaus išrinkimas) ir informacijos modifikavimas yra atliekami tik baigtinėje langelių aibėje (ne daugiau nei aštuoniuose), todėl viso euristinio algoritmo sudėtingumas yra tik $\mathcal{O}(n^2)$.

Šiuo euristiniu algoritmu buvo apskaičiuotas žirgo maršrutas, pavaizduotas 1.7 b paveiksle.

1.4. Skaldyk ir valdyk metodas

Daugelio uždavinių efektyvius sprendimo algoritmus sudarome naudodami tokį metodą:

- uždavinį skaidome į du ar daugiau mažesnius uždavinius;
- randame šių uždavinių sprendinius;
- iš jų sudarome viso uždavinio sprendinį.

Dalinius uždavinius vėl galime spręsti tokiu pačiu metodu, taip skaidome tol, kol gautieji uždaviniai yra lengvai išsprendžiami. Toks rekursyvus algoritmas vadinamas *skaldyk ir valdyk* metodu (angl. *divide and conquer algorithm*).

Aišku, tai yra tik bendra metodika, o sprendžiant konkretų uždavinį reikia paaiškinti, kaip realizuojami šie trys žingsniai. Pastebėsime, kad dažnai tam pačiam uždaviniui *skaldyk ir valdyk* metodu galime sudaryti kelis skirtingus sprendimo algoritmus. Tokių algoritmų pavyzdžius pateiksime sprenddami duomenų rūšiavimo uždavinį.

1.4.1. Algoritmo bendroji schema

Suformuluosime algoritmo bendrąją schemą, kai sprendžiame uždavinį, kurį apibūdina aibė A , o jo sprendinys užrašomas `SprendTipas` duomenų struktūra.

Skaldyk ir valdyk algoritmas

```

SprendTipas SkaldykIrValdyk (Inf A)
(1) if ( |A| > ε )
    (2) Skaldyk (A, A1, A2, ..., AM);
    (3) for (j=1; j<= M; j++)
        (4) Sj = SkaldykIrValdyk (Aj);
        end for
    (5) S = Valdyk (S1, S2, ..., SM);
else
    (6) S = Sprendinys(A)
end if else
return (S);
end SkaldykIrValdyk

```

1.9 pavyzdys. Gretimų taškų radimas. Turime $n = 2^m$ taškų aibę

$$P(n) = \{P_j = (x_j, y_j), j = 1, 2, \dots, n\}.$$

Tegul taškai sunumeruoti abscisių koordinatės didėjimo tvarka:

$$x_1 \leq x_2 \leq \dots \leq x_n.$$

Atstumą tarp dviejų taškų P_i ir P_j apibrėžiame lygybe

$$d(P_i, P_j) = ((x_i - x_j)^2 + (y_i - y_j)^2)^{1/2}.$$

Reikia rasti tokius du taškus P_q ir P_r , tarp kurių atstumas yra mažiausias

$$d(P_q, P_r) = \min_{1 \leq i, j \leq n} d(P_i, P_j).$$

Iš viso galime sudaryti $\frac{n(n-1)}{2}$ skirtingas taškų poras, todėl pilno porų perrinkimo algoritmo sudėtingumas yra $\mathcal{O}(n^2)$.

Padalinkime visus taškus į dvi aibes:

$$P_L\left(\frac{n}{2}\right) = \{P_j, j = 1, 2, \dots, \frac{n}{2}\},$$

$$P_R\left(\frac{n}{2}\right) = \{P_j, j = \frac{n}{2} + 1, 2, \dots, n\}.$$

Tada teisingas vienas iš šių teiginių:

1. Abu taškai P_q, P_r priklauso aibei $P_L\left(\frac{n}{2}\right)$.
2. Abu taškai P_q, P_r priklauso aibei $P_R\left(\frac{n}{2}\right)$.
3. Taškas P_q priklauso aibei $P_L\left(\frac{n}{2}\right)$, o taškas P_r priklauso aibei $P_R\left(\frac{n}{2}\right)$.

Sudarome tokį uždavinio sprendimo algoritmą (norėdami suprastinti žymėjimus tarsime, kad procedūra gražina tik minimalų atstumą tarp taškų P_q, P_r):

```
float GretimiTaškai (Set P, int n)
(1) if ( n == 2 )
    (2) p = d (P1, P2);
else
    (3) pL = GretimiTaškai (PL,  $\frac{n}{2}$ );
    (4) pR = GretimiTaškai (PR,  $\frac{n}{2}$ );
    (5) pT = min(pL, pR);
    (6) p = Pasienis (PL, PR, pT);
```

```

end if else
  return ( $p$ );
end GretimiTaškai

```

Dabar pateiksime procedūros Pasienis realizaciją. Užtenka nagrinėti tik taškus priklausančius pasienio juostai:

$$P_B(n) = \{P_j : x_{n/2+1} - p_T < x_j < x_{n/2} + p_T, j = s, s+1, \dots, f\}.$$

```

float Pasienis (Set  $P_L$ , Set  $P_R$ , float  $p_T$ )
(1) for (i=s; i <= n/2; i++)
  (2) for (j=n/2+1; (| $x_i - x_j$ | <  $p_T$ ) && (j <= f); j++)
    (3)  $p = d(P_i, P_j)$ ;
    (4)  $p_T = \min(p, p_T)$ ;
  end for
end for
  return ( $p_T$ );
end Pasienis

```

1.4.2. Algoritmo sudėtingumo analizė

Dažniausiai uždavinį skaidome į du dalinius uždavinius. Šiuo žingsniu siekiame visą uždavinį padalinti į vienodos apimties dalis.

Įvertinsime *skaldyk ir valdyk* algoritmo sudėtingumą, kai n dydžio uždavinį dalijame į a mažesnių uždavinių, kurių dydis yra n/b , o atskirų sprendinių sujungimo kaštai yra $d(n)$. Kai uždavinys yra mažas, t.y. $n = 1$, jį išsprendžiame koku nors paprastu algoritmu, o veiksmų skaičių pažymėsime c . Tokio algoritmo sudėtingumo funkcija tenkina uždavinį:

$$T(n) = \begin{cases} c, & \text{jei } n = 1, \\ aT(n/b) + d(n), & \text{jei } n > 1. \end{cases}$$

Kai $n = b^m$, jo sprendinį apskaičiuosime 1.1.2 paragrafe:

$$T(n) = ca^m + \sum_{j=0}^{m-1} a^j d(b^{m-j}).$$

Dažniausiai duomenų aibę dalinsime i dvi dalis, o rezultatų sujungimo kaštai bus proporcingi duomenų skaičiui, tada $a = 2, b = 2, d = gn$, ir tokio *skaldyk ir valdyk* algoritmo sudėtingumo funkcija $T(n) = \mathcal{O}(n \log n)$.

1.5. Godieji algoritmai

Dažnai sprendžiame optimizavimo uždavinius, kai leistinų sprendinių aibėje ieškome sprendinio, minimizuojančio pasirinktą tikslo funkciją. Nagrinėjamų variantų skaičius gali būti toks didelis (nepolinominio sudėtingumo funkcija nuo uždavinio parametrų skaičiaus), kad jų visų negalime perrinkti net ir šiuolaikiniais superkompiuteriais.

Tokių uždavinių sprendinio paiešką dažniausiai išskaidome į n etapų ir kiekviename žingsnyje renkamės iš nedidelio baigtinio skaičiaus variantų m . *Godieji* algoritmai rekomenduoja rinktis lokaliai geriausią variantą duotojo žingsnio metu. Tada lokalaus pasirinkimo sudėtingumas $\mathcal{O}(m)$, o viso algoritmo sudėtingumas - tik $\mathcal{O}(nm)$ veiksmų.

Aišku, dažniausiai negalime garantuoti, jog taip spęsdami uždavinį randame tikslų sprendinį, tokie godieji algoritmai yra euristikos, leidžiančios labai sparčiai apskaičiuoti tikslaus sprendinio artinius. Euristikos pavyzdį sukonstravome ieškodami žirgo maršruto šachmatų lentoje.

Bet egzistuoja daug svarbių taikomųjų uždavinių, kai godieji algoritmai apibrėžia tikslų sprendinį. Tokius pavyzdžius nagrinėsime spęsdami grafų teorijos uždavinius, o šiame skirsnyje tik suformuluosime teorines sąlygas, kai godusis algoritmas tiksliai sprendžia optimizavimo uždavinį.

Pirmiausia parodysime, kaip sudaromi godieji algoritmai. Čia spęsimė uždavinius, kurių sudėtingumas yra eksponentinė duomenų skaičiaus funkcija, taigi godusis algoritmas negali garantuoti tikslaus sprendinio radimo.

1.10 pavyzdys. Gražos atidavimo uždavinys. Automatas gražą atiduoda monetomis, kurių nominalai yra:

$$V_1 > V_2 > \dots > V_m.$$

G vertės sumą reikia sudaryti taip, kad monetų skaičius būtų mažiausias. Taigi sprendžiame uždavinį:

$$\min_{(n_1, \dots, n_m) \in D} (n_1 + n_2 + \dots + n_m),$$

$$D = \{n_1 V_1 + n_2 V_2 + \dots + n_m V_m = G, \quad n_j \geq 0\}.$$

Tarkime, kad $V_m = 1$, tada visada galime parinkti bent vieną monetų kombinaciją, kurios suma lygi G . Algoritmo idėja labai paprasta: pirmiausia stengiamės gražą atiduoti didžiausio nominalo monetomis,

tokių monetų skaičius yra $n_1 = \lfloor G/V_1 \rfloor$, po to likusios sumos $G_1 = G - n_1 V_1$ didžiąją dalį atiduome V_2 nominalo monetomis $n_2 = \lfloor G_1/V_2 \rfloor$ ir t.t. Jeigu kuriame nors algoritmo žingsnyje $G_j < V_{j+1}$, tai V_{j+1} nominalo monetų nenaudojame.

Imkime monetų rinkinį

$$V_1 = 25, \quad V_2 = 11, \quad V_3 = 5, \quad V_4 = 1.$$

Godžiuoju algoritmu apskaičiuojame, kad 63 centų gražą reikia atiduoti taip

$$63 = 2 \cdot 25 + 1 \cdot 11 + 2 \cdot 1,$$

taigi naudojame penkias monetas. Nesunku patikrinti, kad toks sprendinys yra optimalus (išnagrinėkite atvejus, kai $n_1 < 2$), bet tada paaiškėja, kad egzistuoja ir dar vienas penkių monetų rinkinys, leidžiantis atiduoti tokią gražą

$$63 = 1 \cdot 25 + 3 \cdot 11 + 1 \cdot 5.$$

Jei $G = 15$, tai godžiuoju algoritmu gražą sudarome taip

$$15 = 1 \cdot 11 + 4 \cdot 1,$$

t.y. vėl naudojame penkias monetas. Tačiau egzistuoja geresnis sprendinys, kai klientui atiduome tris penkių centų monetas.

1.11 pavyzdys. Diskretusis kuprinės užpildymo uždavinys.

(Angl. 0 – 1 *knapsack problem*). Turime n daiktų, kurių tūriai yra v_1, v_2, \dots, v_n , o kaina p_1, p_2, \dots, p_n . Reikia rasti tokį daiktų rinkinį, kuris tilptų į V tūrio kuprinę ir jų vertė būtų didžiausia.

Sprendžiame optimizavimo uždavinį:

$$\max_{(n_1, \dots, n_m) \in D} (n_1 p_1 + n_2 p_2 + \dots + n_m p_m),$$

$$D = \{n_1 v_1 + n_2 v_2 + \dots + n_m v_m \leq V, \quad n_j \in (0, 1)\}.$$

Taigi optimizavimo uždavinio parametrai n_j gali būti lygūs tik vienetui arba nuliui. Pirmiausia apibrėžiame santykinę kiekvieno daikto vertę $s_j = \frac{p_j}{v_j}$. Visus daiktus rūšiuojame šios vertės mažėjimo tvarka, tarsime, kad

$$s_1 \geq s_2 \geq \dots \geq s_n.$$

Kuprinę stengiamės užpildyti didžiausios santykinės vertės daiktais. Juos imame vieną po kito ir tikriname ar daikto tūris yra nedidesnis už likusią laisvą kuprinės dalį. Jei sąlyga yra išpildyta, tai daiktą talpiname į kuprinę, jei ne – imame sekantį daiktą.

Nagrinėkime tokį pavyzdį. Turime aštuonis daiktus (v_j, p_j) :
 $(25, 50)$, $(20, 80)$, $(20, 50)$, $(15, 45)$, $(30, 105)$, $(35, 35)$, $(20, 10)$, $(10, 45)$.
 Apskaičiuojame jų santykinės vertes

$$S = \left\{ 2, 4, 2\frac{1}{2}, 3, 3\frac{1}{2}, 1, \frac{1}{2}, 4\frac{1}{2} \right\}$$

ir surūšiuojame daiktus verčių didėjimo tvarka

$(10, 45)$, $(20, 80)$, $(30, 105)$, $(15, 45)$, $(20, 50)$, $(25, 50)$, $(35, 35)$, $(20, 10)$.
 Imkime kuprinę, kurios tūris $V = 80$. Naudodami godųjų algoritmą į kuprinę įdedame pirmuosius keturis daiktus, jų bendras tūris 75, o vertė 275 litai. Tai nėra geriausias sprendinys, nes imdami pirmuosius tris ir penktąjį daiktus užpildome visą kuprinę, o tokio krovinio vertė 280 litų. Taigi godusis algoritmas vėl yra tik euristika.

Šiame pavyzdyje į kuprinę galėjome įdėti tik visą daiktą. Jei galima imti ir dalį daikto, tai įsitikinsime, kad godusis algoritmas visada apibrėžia tikslų uždavinio sprendinį.

1.12 pavyzdys. Tolydusis kuprinės užpildymo uždavinys.

(Angl. *fractional knapsack problem*). Užduoties sąlyga tokia pati kaip ankstesniame pavyzdyje, tik dabar daiktai yra dalūs ir galime imti jų dalį. Tai gali būti aukso plytelės ir aukso smėlis, arba supakuotas ir birus cukrus. Tada vykdydami godųjų kuprinės pildymo algoritmą imsime visą kiekį vertingų daiktų, o mažiausiai vertingo, bet dar tilpusio į kuprinę produkto imsime tik tiek, kiek liko laisvos vietos kuprinėje.

Imkime daiktų rinkinį, pateiktą ankstesniame pavyzdyje ir surūšiuotą jų santykinų verčių didėjimo tvarka $(10, 45)$, $(20, 80)$, $(30, 105)$, $(15, 45)$, $(20, 50)$, $(25, 50)$, $(35, 35)$, $(20, 10)$. Tada į kuprinę, kurios tūris $V = 80$, įsidėsime pirmuosius keturis daiktus (jų tūris 75, o vertė 275 litai) ir ketvirtadalį penktojo produkto (jo tūris 5, o vertė 12,5 litų). Taigi krovinyms užpildo visą kuprinę, o jo vertė 287,5 litai.

Godžiųjų algoritmų naudojimo sritis

Šiame skyrelyje aptarsime, kada tikslinga naudoti godžiuosius algoritmus. Pirmiausia pastebėsime, kad tie uždaviniai, kurių tikslų sprendinį ran-

dame godžiuoju algoritmu, būtina tenkina sąlygą, kad šį sprendinį galime skaičiuoti išskaidę uždavinį į mažesnių užduočių seką ir suradę jų optimalius sprendinius. Pavyzdžiui, jei optimaliai užpildėme kuprinę (tiek diskrečiuoju, tiek ir tolydžiuoju atveju), tai išėmę iš jos vertingiausią daiktą vėl gauname panašų uždavinį: reikia užpildyti $(V - v_1)$ tūrio kuprinę likusiais daiktais.

Tokia optimalaus sprendinio priklausomybė nuo mažesnių šio uždavinio dalių optimalių sprendinių yra būdinga ir dinaminio programavimo metodu sprendžiamiems uždaviniams. Tačiau abu metodai visiškai skirtingai renkasi lokalaus uždavinio sprendinį. *Godžiajame* algoritme visada renkamės lokaliai vertingiausią sprendinį, kitame žingsnyje vėl imame vertingiausią sprendinį iš likusių variantų ir t.t. Taip rinkdamiesi lokalių uždavinių sprendinius tikimės, kad išlieka galimybė rasti globaliai optimalų sprendinį. *Dinaminio programavimo* metode prieš pasirinkdami lokalaus uždavinio sprendinį įvertiname tokio pasirinkimo pasekmes visiems tolimesniems variantams.

Algoritmo teisingumas. Godieji algoritmai yra labai efektyvūs, nes jų sudėtingumo funkcija yra polinominė (dažniausiai kvadratinė arba kubinė) duomenų skaičiaus n funkcija. Bet ne visada galime garantuoti, kad šiuo algoritmu randame optimalų uždavinio sprendinį. Norėdami tai įrodyti dažniausiai naudojame tokią schemą:

1. Įrodome, kad ir po godžiojo pasirinkimo pirmajame žingsnyje liko galimybė rasti optimalų viso uždavinio sprendinį, t.y. būtina egzistuoja bent vienas sprendinys, kuris yra suderintas su pirmajame žingsnyje atliktu pasirinkimu.
2. Įrodome, kad atlikę pirmąjį žingsnį vėl gavome tokį patį optimizacijos uždavinį tik mažesnėje pasirinkimų aibėje. Tada algoritmo teisingumo įrodymą užbaigiame matematinės indukcijos būdu.

Įrodykime, kad godžiuoju algoritmu randame optimalų tolydaus kuprinės pildymo uždavinio sprendinį. Jei į kuprinę tilpo tik vertingiausias daiktas ar jo dalis, tai godžiojo algoritmo teisingumas yra akivaizdus. Todėl nagrinėkime atvejį, kai godžiojo algoritmo sprendinys yra toks:

$$\{(v_1, p_1), (v_2, p_2), \dots, (v_{m-1}, p_{m-1}), (w_m, p_m)\}, \quad w_m \leq v_m, \quad m \geq 2..$$

a) Pirmajame godžiojo žingsnyje pasirinkome santykinai vertingiausią daiktą (v_1, p_1) . Tarkime, kad optimaliame rinkinyje šio daikto kiekis yra $w_1 < v_1$, o viso rinkinio vertė P litų. Tada išėmę iš kuprinės

$$\Delta w_1 = \min(w_m, v_1 - w_1)$$

kiekį mažiausiai vertingo produkto ir įdėję tokį patį kiekį pirmojo produkto, gausime naują rinkinį, kurio vertė yra nemažesnė už ankstesnio, nes

$$(s_1 - s_m)\Delta w_1 \geq 0.$$

Matome, kad tada, kai dalies produktų santykinės vertės sutampa, gali egzistuoti ir keli uždavinio sprendiniai.

b) Po pirmojo žingsnio vėl gavome kuprinės užpildymo vertingiausiais produktais uždavinį, tik sumažėjo kuprinės tūris ($V - v_1$) ir trumpesniu tapo daiktų sąrašas. Remdamiesi matematinės indukcijos metodu tvirtiname, kad godžiuoju algoritmu radome optimalų sprendinį.

