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

Nagrinėkime matricų sumos $A + B$ skaičiavimo algoritmą

$$C = A + B, \quad C = (c_{ij}), \;\; 1 \leqslant i, j \leqslant n\,,$$
$$c_{ij} = a_{ij} + b_{ij}\,.$$

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 matricų sandaugos $C = AB$ skaičiavimo algoritmą

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

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

Taigi matome, kad dviejų matricų 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 spręsdami N tiesinių lygčių sistemą, atliekame $\frac{2}{3}N^3 + O(N^2)$ aritmetinių veiksmų. Remdamiesi šia formule, įvertiname, jog, spręsdami 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:

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

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:

$$0 \leqslant f(n) \leqslant c\, g(n), \quad n \geqslant n_0\,.$$

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

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

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

Nesunku patikrinti (pvz. matematinės indukcijos metodu), kad

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

todėl gauname tokius įverčius

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

Konstantą c galime ir sumažinti, jei padidinsime n_0, pavyzdžiui

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

todėl

$$f(n) = n^2 + 4n + 4 \leqslant n^2 + n^2 \leqslant 2n^2\,, \quad n \geqslant 5\,.$$

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

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

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 dideliam skaičiui skirtingų natūrinių n galioja nelygybės:

$$f(n_j) \geqslant c\, g(n_j), \quad j = 1, 2, \ldots, \infty\,,$$

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:

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

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:

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

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:

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

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 $n!$ reikšmęs, 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:

$$T(n) \leqslant f(n), \quad \text{kai } \ n \geqslant n_0\,.$$

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:

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

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 spręsdami 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:

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

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

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

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

Tada gauname tokią algoritmo sudėtingumo funkciją

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

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 rėžių* 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:

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

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 devynerių 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 Fibonaci*). Skaičiai sutinkami įvairiuose informatikos algoritmuose bei algoritmų sudėtingumo analizėje. Jie apibrėžiami tokiu būdu:

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

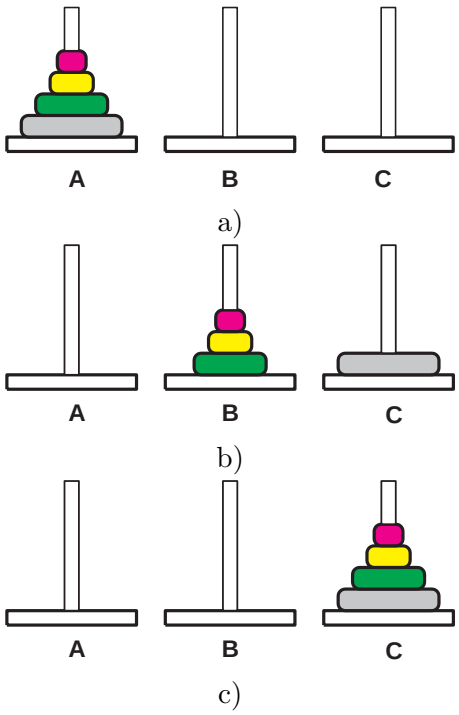
Abiejuose pavyzdžiuose matome tipišką rekursijos struktūrą:

- 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 dinaminių 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 patogu apibrėžti naudojant rekursiją:

1. Pirmiausia laikydamiesi taisyklių perkeliame $(n-1)$ viršutinius A žiedus ant B virbo, o C virbu naudojamės kaip pagalbiniu.

2. Tada perkeliame nuo A ant B paskutinį didžiausią žiedą (žr. 1.1b paveikslą).

3. Paskutiniame žingsnyje tuo pačiu būdu perkeliame žiedus nuo C ant B 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, B);
(4)    HanojausBokštai (n-1, C, B, A);
(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 efktyviai 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 algoritmas

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

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

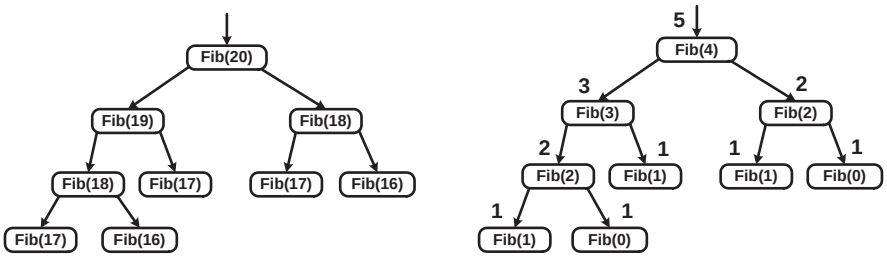
Abiejuose algoritmuose atliekame po tiek pat aritmetinių veiksmų, tačiau iteracinio algoritmo vykdymo trukmė bus trumpesnė, nes rekursijos papildomieji kaštai yra didesni už ciklo operatoriaus papildomuosius kaštus.

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.



a) b)

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 skačiavimo kaštus. Bazine algoritmo operacija laikykime bet kokį 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:

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

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, $\rho_n = \dfrac{T_n}{T_{n-1}}$ 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
```

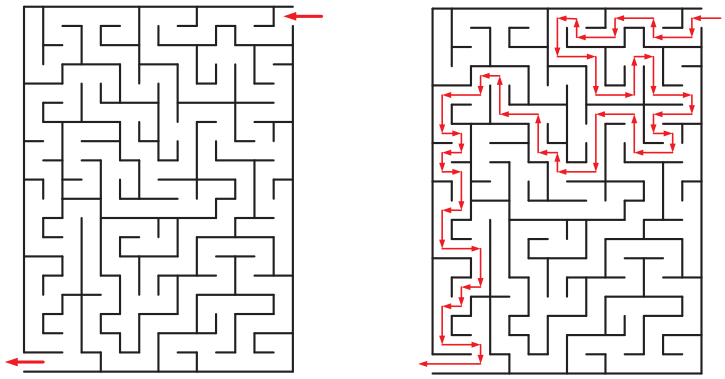
Algoritme panaudojame rekursiją, tačiau jo analizė rodo, kad kiekvieną kartą iš dviejų medžio šakų pasirenkama tik viena. Todėl efektyvesnė yra tokia algoritmo modifikacija, kurioje jau nenaudojame rekursijos:

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).



a) *b*)

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 spręsime ne vieną atskirą uždavinį, o sudarysime 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
```

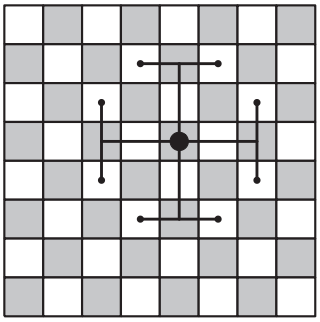
Trumpai aptarsime algoritmo struktūrą. Funkcija BandymųAibė generuoja visus naujus ėjimus, kuriuos galima atlikti iš taško P. Leistinų ėjimų aibę apibrėžia uždavinio sąlygos ir duomenų struktūroje pozicija saugoma informacija apie jau aplankytas paieškos vietas. Kai ši aibė tampa tuščia, tada procedūra 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 pasiekiame 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.



55	42	15	12	57	32	17	10
14	1	56	43	16	11	34	31
41	54	13	64	33	58	9	18
2	25	46	51	44	63	30	35
47	40	53	26	59	36	19	8
24	3	50	45	52	27	62	29
39	48	5	22	37	60	7	20
4	23	38	49	6	21	28	61

a) *b*)

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

Naują žirgo padėtį $U = P+R$ generuojame naudodami koordinačių pokyčių vektorių

$$R[0] = (2,1),\ R[1] = (1,2),\ R[2] = (-1,2),\ R[3] = (-2,1),$$
$$R[4] = (-2,-1),\ R[5] = (-1,-2),\ R[6] = (1,-2),\ R[7] = (2,-1).$$

Taškas $U = (u, v)$ turi priklausyti šachmatų lentai, todėl dar patikriname sąlygą:

$$1 \leqslant u \leqslant n, \quad 1 \leqslant v \leqslant n\,.$$

Informaciją apie jau aplankytus langelius saugome matricoje H, kurios visi elementai pradžioje yra lygūs nuliui. Jeigu (p, q) langelį žirgas aplankė k ėjimo metu, tai $H(p, q) = k$. Taigi nauja pozicija yra leistina, jei parinkome langelį $U = (u, v)$, kuris dar neaplankytas, t.y. $H(u, v) = 0$.

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.

Algoritmo sudėtingumas

Pateiktojo variantų perrinkimo algoritmo sudėtingumas labai priklauso nuo variantų išrinkimo tvarkos. Svarbi rekursinio algoritmo savybė yra yra ta, kad pakartotinai netikrinami jau peržiūrėti variantai. Bet blogiausiu atveju tenka nagrinėti visus galimus paieškos variantus, kurių skaičius greitai didėja, kai didiname uždavinio charakteringą parametrą n.

Kaip pavyzdį imkime žirgo maršruto šachmatų lentoje uždavinį. $n \times n$ dydžio lentoje egzistuoja $\mathcal{O}(n^{16})$ skirtingų maršrutų. Tarkime, kad per vieną sekundę galime patikrinti vieną miliardą variantų. Tada 6^{16} variantus tikrintume 47 minutes, 7^{16} variantus – $9,5$ valandas, o 8^{16} variantus – 78 valandas. Pateiksime rezultatus, kuriuos gavome ieškodami žirgo maršrutų 6×6 šachmatų lentoje. Pasirinkę pradinę žirgo padėtį langeliuose $(1,1)$, $(1,6)$ arba $(2,2)$, maršrutą randame greičiau nei per $0,1$ sekundės dalį. Tačiau, pradėję paiešką iš langelio $(5,5)$, kuris yra simetrinis langeliui $(2,2)$, uždavinį sprendėme 54 sekundes. Jei šachmatų lentos dydis yra 8×8, tai maršrutą, prasidedantį $(1,1)$ langelyje, randame per vieną sekundę. Pradėję paiešką iš langelio $(2,2)$ ir po dešimties valandų dar neradome tokio maršruto.

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 renkamės 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 renkamės 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 $H(p, q)$ priskirisme 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.