ĮVADAS

Algoritmų analizės objektas yra algoritmai. Nors algoritmo sąvoka yra laikoma pirmine matematikos sąvoka, nereikalaujančia apibrėžimo, dažnai algoritmą apibūdina kaip baigtinę seką tikslių komandų (instrukcijų), nurodančių kaip rasti nagrinėjamo uždavinio sprendinį.

Beveik visus algoritmus galima suskirstyti į dvi dideles klases: *kombinatorinius algoritmus* ir *skaitinius algoritmus*. Kombinatoriniai algoritmai operuoja su diskrečiais (= kombinatoriniais) objektais: sveikaisiais skaičiais, baigtinėmis aibėmis, grafais, matricomis ir pan. Skaitiniai algoritmai paprastai yra skaičiavimo metodų realizacijos, t.y., algoritmai sprendžiantys įvairaus pavidalo matematines lygtis su realiais koeficientais arba optimizuojantys realaus argumento funkcijas. Nagrinėdami algoritmus, mes pagrindinį dėmesį skirsime kombinatoriniams algoritmams. Taigi, algoritmų analizės kursą galima laikyti skaičiavimo metodų kurso analogu diskrečioje matematikoje: skaičiavimo metodai taiko matematinę analizę (pvz., diferencialines lygtis) tolydiems uždaviniams spręsti, o kombinatoriniai algoritmai taiko diskrečią matematiką diskretiems uždaviniams spręsti. Išvardinsime tik keletą kombinatorinių uždavinių pavyzdžių:

1. Reikia sudaryti optimalų paskaitų tvarkaraštį Vilniaus Universiteto MIF fakultete.
2. Reikia rasti trumpiausią maršrutą, praeinantį po 1 kartą per kiekvieną iš n miestų, kai duoti atstumai tarp tų miestų (taip vadinamas *keliaujančio pirklio* uždavinys; trumpiau: KPU).
3. Reikia parašyti programą, gerai žaidžiančią šachmatais.

Diskretūs uždaviniai dažniausiai yra susiję su variantų perrinkimo problema. Kadangi galimų duoto uždavinio sprendinių skaičius dažniausiai būna baigtinis, tai išnagrinėję visus galimus variantus ir juos įvertinę (t.y., priskirę kiekvienam variantui jo vertę), mes galėtume išsirinkti geriausią sprendinį. Taip elgiasi taip vadinami brutalios jėgos (angl. *brute force*) algoritmai. Deja, praktikoje mes dažnai susiduriame su taip vadinama *kombinatorinio sprogimo* problema: jei uždavinys yra pakankamai didelis (toks, kurio mes be kompiuterio pagalbos nebegalime išspręsti), tai galimų variantų skaičius labai greitai auga ir pasiekia tokį kiekį, kurio negalima perrinkti ir su geriausiu pasaulyje kompiuteriu. Pavyzdžiui, ieškant optimalaus keliaujančio pirklio maršruto, norint apkeliauti n miestų, gali tekti nagrinėti $(n-1)!$ skirtingų maršrutų (plg. $20! = 2\,432\,902\,008\,176\,640\,000$); norint išnagrinėti visas galimas šachmatų partijos pozicijas n ėjimų į priekį, gali tekti perrinkti apie $20^{2n} = 400^n$ variantų, nes kiekviename ėjime tiek

baltieji, tiek juodieji gali turėti po 20 skirtingų galimybių (plg. $400^{10} = 1048576 \times 10^{20}$). Todėl kombinatoriniai uždaviniai natūraliai suskyla į dvi grupes:

- uždaviniai, kuriems yra žinomas polinominio sudėtingumo algoritmas, arba *paprasti* (angl. *tractable*) uždaviniai, pavyzdžiui, trumpiausio kelio tarp dviejų miestų radimo problema, ir
- uždaviniai, kuriems nėra žinoma jokio polinominio sudėtingumo algoritmo, arba *sunkūs* (angl. *intractable*) uždaviniai, pavyzdžiui, aukščiau minėtas KPU.

Pagrindinės algoritmų analizės nagrinėjamos problemos yra šios:

(1) *algoritmo sustojimo* problema: reikia nustatyti, ar konkretus algoritmas, pritaikytas konkretiems pradiniams duomenims, baigs darbą ar dirbs be galo;

(2) *algoritmo korektiškumo* (teisingumo) problema: reikia nustatyti, ar konkretus algoritmas išduos atsakymą, sutampantį su tikruoju nagrinėjamo uždavinio sprendiniu;

(3) *algoritmo sudėtingumo* problema: reikia nustatyti, kiek žingsnių daugiausia atliks konkretus algoritmas iki sustojimo, ar jis užbaigs darbą per mums priimtiną laiką, ir ar šiam algoritmui užteks turimų atminties resursų;

(4) *algoritmo efektyvumo* problema: nustačius algoritmo sudėtingumą, reikia įvertinti, kiek jis yra efektyvus, t.y., ar tai yra pats geriausias galimas algoritmas nagrinėjamam uždaviniui spręsti, ar galima rasti geresnį algoritmą.

Pavyzdys 1. Duotas sveikasis teigiamas skaičius n. Reikia rasti jo faktorialą $n!$. Panagrinėkime tokį algoritmą:

function $\text{faktorialas} = \text{fact1}(n)$
$i := 1;$
$\text{faktorialas} := 1;$
while $i < n$ **do**
  $i := i + 1;$
  $\text{faktorialas} := \text{faktorialas} \cdot i;$

1. Pirmiausia įrodysime, kad algoritmas fact1 baigia darbą (sustojimo problema). Pakanka įrodyti, kad ciklas **while** nėra begalinis. Ciklo pradžioje kintamasis $i = 1$. Kadangi pirmoji vidinė ciklo komanda prie i prideda 1, o antroji komanda kintamojo i nekeičia, tai po $n-1$ ciklo iteracijos i tampa lygus n, ir ciklo vykdymas yra nutraukiamas.

2. Dabar įrodysime algoritmo korektiškumą, t.y., kad atsakymas tikrai bus $\text{faktorialas} = n!$. Programų verifikacijos principus pasiūlė anglų informatikas Hoare[1]. Pažymėkime raide p teiginį

[1] **C. Anthony R. Hoare (g. 1934).** Hoare įnešė svarbų indėlį į programavimo kalbų teoriją ir programavimo metodologiją. Jis pirmasis apibrėžė programavimo kalbą, leidžiančią įrodinėti programų korektiškumą jų specifikacijų atžvilgiu. Hoare pasiūlė algoritmą *Quicksort*, kuris dabar yra vienas iš plačiausiai naudojamų ir ištyrinėtų rūšiavimo algoritmų.

"faktorialas $= i!$ ir $i \leqslant n$". Pirmiausia įrodysime, kad šis teiginys yra invariantiškas ciklo atžvilgiu.

Programoje $\mathrm{fact1}$ naudojamą ciklą schematiškai galime pažymėti "**while** *sąlyga* **do** S". Teiginį p vadina invariantišku tokio ciklo atžvilgiu, jei teiginys "$(p \,\&$ *sąlyga*$)\{S\}p$" yra teisingas. Užrašas $p\{S\}q$ reiškia, kad programos segmentas S yra *dalinai teisingas pradinės prielaidos* p *ir galutinės prielaidos* q *atžvilgiu*, t.y., jei p yra teisingas prieš pradedant vykdyti S, tai ir q bus teisingas 1 kartą įvykdžius S. Tarkime, kad prieš prasidedant ciklui teisinga $p \,\&$ *sąlyga*, t.y., teisinga $\mathrm{faktorialas} = i!$ ir $i < n$. Naujos kintamųjų i ir faktorialas reikšmės bus $i_{\mathrm{new}} = i + 1$ ir $\mathrm{faktorialas}_{\mathrm{new}} = \mathrm{faktorialas} \cdot (i + 1) = (i + 1)! = i_{\mathrm{new}}!$. Kadangi $i < n$, tai $i_{\mathrm{new}} = i + 1 \leqslant n$. Taigi, teiginys p yra teisingas ciklo gale; vadinasi, p yra invariantiškas šio ciklo atžvilgiu.

Norėdami baigti įrodyti programos $\mathrm{fact1}$ korektiškumą pasinaudosime programų verifikacijoje ciklui **while** naudojama *išvedimo taisykle* $(p \,\&\, \textit{sąlyga})\{S\}p \vdash p\{\textbf{while}\ \textit{sąlyga}\ \textbf{do}\ S\}(\neg \textit{sąlyga}\ \&\, p)$, kuri sako, kad jei teiginys p yra teisingas prieš ciklo vykdymą, tai teiginys $\neg$*sąlyga* $\&\, p$ yra teisingas užbaigus ciklą.

Kadangi prieš pradedant ciklą turime $i = 1 \leqslant n$ ir "$\mathrm{faktorialas} = 1 = i!$, tai prieš ciklo vykdymą teiginys p yra teisingas. Pagal aukščiau pateiktą taisyklę užbaigus ciklą bus teisingas teiginys ($\neg$*sąlyga* $\&\, p$), t.y., bus teisinga konjunkcija $i \geqslant n \,\&\, \mathrm{faktorialas} = i! \,\&\, i \leqslant n$. Gauname, kad $i = n$ ir $\mathrm{faktorialas} = n!$. Taigi, programa $\mathrm{fact1}$ yra korektiška.

3. Apskaičiuosime algoritmo $\mathrm{fact1}$ sudėtingumą. Laikydami vienoje eilutėje užrašytos komandos vykdymą 1 žingsniu, gausime, kad algoritmas $\mathrm{fact1}$ sustoja po $L_1(n) = 3n$ žingsnių. Pavyzdžiui, kai $n = 2$, bus įvykdytos tokios komandos:

$i := 1;$
$\mathrm{faktorialas} := 1;$
$1 < 2?$
$i := 1 + 1 = 2;$
$\mathrm{faktorialas} := 1 \cdot 2 = 2;$
$2 < 2?$

4. Panagrinėkime, ar galima rasti efektyvesnį algorimą skaičiaus faktorialui skaičiuoti. Pabandykime tą pačią programą užrašyti su ciklu **for**:

function $\mathrm{faktorialas} = \mathrm{fact2}(n)$
$\mathrm{faktorialas} := 1;$
if $n > 1$ **then for** $i := 2$ **to** n **do** $\mathrm{faktorialas} := \mathrm{faktorialas} \cdot i;$

Kadangi realizuojant ciklą **for** kiekvienoje ciklo iteracijoje prie ciklo kintamojo i bus pridedama po 1 ir kiekvieną kartą bus tikrinama ciklo pabaigos sąlyga $i = n?$, tai ir šio algoritmo sudėtingumas bus $L_2(n) = 3n$, kai $n > 1$, ir $L_2(1) = 2$, kai $n = 1$.

Panašų sudėtingumą gausime ir realizuodami faktorialo skaičiavimą rekursijos pagalba:

function faktorialas $=$ fact3(n)
if $n = 1$ **then** faktorialas $:= 1$ **else** faktorialas $:=$ fact3$(n-1) \cdot n$;

Kreipimąsi į funkciją fact3 laikant atskiru žingsniu, algoritmo fact3 sudėtingumas gaunasi $L_3(n) = 3n - 1$. Pavyzdžiui, kai $n = 2$, bus įvykdytos tokios komandos:

$2 = 1$?
call fact3(1)
$1 = 1$?
faktorialas $:= 1$;
faktorialas $:= 1 \cdot 2 = 2$;

Kadangi funkcijų iškvietimas praktiškai reikalauja daugiau laiko, negu paprastos priskyrimo komandos, reikėtų tikėtis, kad algoritmas fact3 praktiškai bus lėtesnis už algoritmus fact1 ir fact2. Pastarųjų vykdymo laikas priklausys nuo pasirinkto kompiuterio ir kompiliatoriaus.

Visų trijų aukščiau pateiktų algoritmų sudėtingumas tiesiškai priklauso nuo skaičiaus n, todėl šių algoritmų vykdymo laikas skirsis labai nežymiai. Pridėkime dar vieną "kvailą" algoritmą, tinkantį kompiuteriui, kuris moka tik sudėti sveikus skaičius, bet nemoka jų dauginti:

function faktorialas $=$ fact4(n)
faktorialas $:= 1$;
if $n > 1$ **then for** $i := 2$ **to** n **do**
  sumfakt $:=$ faktorialas;
  for $j := 2$ **to** i **do** sumfakt $:=$ sumfakt $+$ faktorialas;
  faktorialas $:=$ sumfakt

Kadangi šis algoritmas turi dvigubą ciklą, tai jo sudėtingumas bus $L_4(n) = \mathrm{O}(n^2)$.[2]

Visi 4 algoritmai buvo realizuoti su Matlab programa 700 MHz kompiuteriu. Papildomai $n!$ dar buvo skaičiuojamas su vidine Matlab funkcija $\mathrm{prod}(1:n)$, kuri randa masyvo $[1, 2, 3, \ldots, n]$ elementų sandaugą ir su Matlab funkcija gamma$(n+1)$, kur $\mathrm{gamma}(x) = \int_0^\infty t^{x-1}\mathrm{e}^{-t}\,\mathrm{d}t$ ir yra žinoma, kad $\mathrm{gamma}(n+1) = n!$. Lentelėje pateikiame visų 6 algoritmų išnaudotą CPU laiką mikrosekundėmis (1 mikrosekundė $= 10^{-6}$ sek.). Kadangi Matlab laiką matuoja tik šimtųjų sekundės dalių tikslumu, tai kiekvienas algoritmas buvo kartojamas cikle 100000 kartų.

Algoritmas	$n = 5$	$n = 10$	$n = 20$	$n = 50$
fact1	38.464	80.488	162.127	404.050
fact2	24.577	42.690	76.018	175.392
fact3	87.946	179.047	359.362	906.222
fact4	85.803	258.583	844.149	4438.480
prod	10.796	11.105	12.424	13.540
gamma	437.719	581.236	298.690	298.150

[2]Primename, kad žymėjimas $f(n) = \mathrm{O}(g(n))$ reiškia, kad $\exists N \in \mathbb{N}$ ir $\exists c > 0$: $f(n) \leqslant cg(n)\ \forall n \geq N$.

Gauti rezultatai rodo, kad vidinė Matlab funkcija prod neilgus masyvus daugina praktiškai per pastovų laiką, nepriklausomai nuo masyvo ilgio. Ši funkcija sprendžia nagrinėjamą uždavinį efektyviausiai. Iš šių paskaitų autoriaus pateiktų algoritmų "nugalėjo" algoritmas $\mathrm{fact2}$, naudojantis ciklą **for**. Rekursyvus algoritmas, kaip ir reikėjo tikėtis, veikia ilgiau už iteracinius. Keistoką funkcijos $\mathrm{gamma}(x)$ laiko kitimą nesunku paaiškinti: kai $x < 12$, ši funkcija yra skaičiuojama iteratyviai, o kai $x > 12$, yra naudojamos apytikslės formulės. Todėl didesnėms argumento reikšmėms funkcijos reikšmė apskaičiuojama greičiau.

Išnagrinėtas pavyzdys rodo, kad algoritmo sustojimo ir korektiškumo nagrinėjimas yra gana nuobodus ir varginantis užsiėmimas. Gal būt galima šį užsiėmimą pavesti kompiuteriui, t.y., sukurti universalią programą, kuri pagal duotą programą P ir jos įėjimo duomenis I atsakytų, ar programa kada nors sustos. Įrodysime, kad tokios programos neegzistuoja, t.y., sustojimo problema yra algoritmiškai neišsprendžiama.

Visas galimas programas, parašytas kuria nors pasirinkta programavimo kalba, galima koduoti natūraliaisiais skaičiais, t.y., sugalvoti taisyklę, kaip kiekvienai programai P priskirti jos kodą $\langle P\rangle \in \mathbb{N}$. Tarkime, kad egzistuoja programa H su dviem įėjimais, skaičiuojanti funkciją

$$\mathrm{HALT}(\langle P\rangle, I) = \begin{cases} 1, & \text{jei } P(I) \text{ sustoja;} \\ 0, & \text{jei } P(I) \text{ dirba be galo.} \end{cases}$$

Tada galima sukonstruoti programą K, kuri programai H į abu įėjimus paduoda programos P kodą $\langle P\rangle$ ir tikrina, kam lygus atsakymas $\mathrm{HALT}(\langle P\rangle, \langle P\rangle)$. Jei atsakymas yra 1, programa K nukreipia į amžiną ciklą. Jei atsakymas yra 0, programa K sustoja ir išduoda vienetą. Taigi, programa $K(\langle P\rangle)$ sustoja tada ir tik tada, kai savo kodui pritaikyta programa $P(\langle P\rangle)$ nesustoja. Ji skaičiuoja dalinę funkciją

$$K(\langle P\rangle) = \begin{cases} 1, & \text{jei } P(\langle P\rangle) \text{ nesustoja;} \\ ?, & \text{jei } P(\langle P\rangle) \text{ sustoja,} \end{cases}$$

kur ? reiškia, kad funkcijos reikšmė yra neapibrėžta. Dabar imame programos K kodą $\langle K\rangle$ ir paleidžiame su juo programą K. Gauname prieštaravimą: $K(\langle K\rangle)$ sustoja tada ir tik tada, kai $K(\langle K\rangle)$ nesustoja. Gautas prieštaravimas įrodo, kad neegzistuoja sustojimo problemą sprendžiančios programos H.

Analogiškai yra įrodoma, kad ir algoritmo korektiškumo problema yra algoritmiškai neišsprendžiama, t.y., neegzistuoja programos, kuri patikrintų ar pritaikius programą P įėjimui I, jos išėjimas bus O. Kitaip sakant, funkcija

$$\mathrm{CORRECT}(\langle P\rangle, I, O) = \begin{cases} 1, & \text{jei } P(I) = O; \\ 0, & \text{priešingu atveju} \end{cases}$$

nėra algoritminė.

Kadangi algoritmų korektiškumas yra nagrinėjamas programų verifikacijos kurse, algoritmų analizėje pagrindinį dėmesį skirsime algoritmų sudėtingumui ir efektyvių algoritmų konstravimui.

Pademonstruosime, kad sugalvoti greitesnį algoritmą gali būti dar svarbiau, negu sugebėti iš esmės padidinti procesoriaus greitį.

Pavyzdys 2. Tarkime, kokiam nors uždaviniui yra žinomas $\mathrm{O}(n^4)$ sudėtingumo algoritmas ir nėra jokio geresnio algoritmo (tokio sudėtingumo, pavyzdžiui, yra kai kurie maksimalaus srauto tinkle radimo algoritmai). Tarkime, konstanta, įeinanti į O apibrėžimą yra lygi 5. Jei superkompiuteris sugeba atlikti 1 milijardą operacijų per sekundę, o personalinis namų kompiuteris — 1 milijoną operacijų per sekundę, tai uždavinį dydžio $n = 1000$ pirmasis kompiuteris spręs $5 \cdot 10^{12}/10^9 = 5000$ sekundžių $=$ 1 h 23 min. 20 sek., o antras kompiuteris — $5 \cdot 10^{12}/10^6 = 5000000$ sekundžių $=$ 57 paros 20 h 53 min. 20 sek. Aišku, namų sąlygomis nepavyks išspręsti tokio uždavinio, nes per tą laiką bent kartą dings elektra arba namiškiams atsibos per naktis dūzgiantis kompiuteris.

Dabar tarkime, kad per keletą metų, investavus milijardines lėšas, pasaulyje pavyko 10 kartų pagreitinti superkompiuterį (iki 10 milijardų operacijų per sekundę) ir 10 kartų pagreitinti personalinius kompiuterius (iki 10 milijonų operacijų per sekundę). Tada superkompiuteris šį uždavinį išspręs per 8 min. 20 sek., na o paprastam mirtingajam su savo personaliniu kompiuteriu atsakymo reikės laukti 5 paras 18 h 53 min. 20 sek.

Tačiau gali būti ir kitaip. Vieną rytą koks nors genialus matematikos olimpiadininkas pliaukšteli sau per kaktą ir rėkia: "Radau algoritmą sudėtingumo $\mathrm{O}(n^3)$". Tarkime, kad konstanta vėl lygi 5. Tada senuoju superkompiuteriu šį uždavinį spręsime $5 \cdot 10^9/10^9 = 5$ sekundes, o senuoju geruoju namų PC-iuku — $5 \cdot 10^9/10^6 = 5000$ sekundžių $=$ 1 h 23 min. 20 sek., t.y., tiek laiko kiek naudodamas seną algoritmą sugaišo superkompiuteris.

Išvada 1. *Geras algoritmas geriau už gerą kompiuterį!*

LITERATŪRA

[CLR] T.H. Cormen, C.E. Leiserson and R.L. Rivest, *Introduction to Algorithms*, The MIT Press/ McGraw-Hill, Cambridge, MA/New York, 1990. (Yra taip pat rusiškas vertimas: T. Kormen, Č. Leizerson, R. Rivest, *Algoritmy: Postroenije i Analiz*, MCNMO, Moskva, 2001.)

[RND] E.M. Reingold, J. Nievergelt and N. Deo, *Combinatorial Algorithms: Theory and Practice*, Prentice-Hall, Englewood Cliffs, NJ, 1977. (Yra taip pat rusiškas vertimas: Ė. Reingol'd, Ju. Nivergel't, N. Deo, *Kombinatornyje Algoritmy: Teorija i Praktika*, Mir, Moskva, 1980.)

[AHU] A.V. Aho, J.E. Hopcroft and J.D. Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, MA, 1976. (Yra taip pat rusiškas vertimas: A. Acho, Dž. Chopkroft, Dž. Ul'man, *Postroenije i Analiz Vyčislitel'nych Algoritmov*, Mir, Moskva, 1979.)

[CH] N. Christofides, *Graph Theory: An Algorithmic Approach*, Academic Press, New York, 1975. (Yra taip pat rusiškas vertimas: N. Kristofides, *Teorija Grafov: Algoritmičeskij Podchod*, Mir, Moskva, 1978.)