Svarbiausia pitakas, toliau tai tik sportinis interesas (Karolis Šakinis):)

GENERALINIS RĖMĖJAS: Dalius Makackas

Pirma klausimų grupė (4 balai):

1. Dinaminis programavimas (15 sk. 358 psl.). Algoritmų sudarymo metodika (15.3 sk. 379-389 psl ir šios metodikos taikymas sprendžiant Konvejerio (Surinkimo linijos planavimo) (15.1 sk. Antras knygos leidimas) arba Bendro ilgiausio posekio radimo (15.4 sk. 390-395 psl.) uždavinį (rekursinės lygtys, optimalaus sprendinio reikšmės ir struktūros radimas, algoritmo sudėtingumo radimas…).

Dinaminis programavimas – programavimo metodas, paremtas uždavinio skaidymu į mažesnes susijusias problemas, bei tų problemų sprendimų įsiminimu. Taigi laiko sąnaudos pakeičiamos atminties sanaudomis.

1 pavyzdys: X = 'V**IL**NIUS', Y = 'HE**L**SINKIS', σ = 32 (lietuviška abėcėlė), m = 7, n = 9 → S = 'INIS' arba 'LNIS', p = 4

Pavyzdyje negalime rasti ilgesnio negu keturių raidžių bendro X ir Y sekų posekio, kadangi trumpesnės sekos X simbolių 'V' ir 'U' sekoje Y niekur nėra, ir iš ženklų poros 'IL' galime išrinkti tik vieną raidę, nes abiejų tų simbolių nerandame toje pačioje tvarkoje sekoje Y. Kad gautumėme vieną iš bip posekių, reikia naikinti simbolius 'V', 'U' ir 'L' arba pirmąją 'I' iš sekos 'VILNIUS' ir simbolius 'H', 'E', pirmąją 'S', 'K' ir 'L' arba pirmąją 'I' iš sekos 'HELSINKIS'.

Naudojant Vagnerio ir Fischerio tiesmukišką, vadinamąjį brutalios jėgos, algoritmą, kuris remiasi dinaminiu programavimu, galime išspręsti problemą palyginant pasirinktą sekos X simbolį su visais sekos Y simboliais.

Rekursyvi taisyklė skaičiuoti įvestų sekų prefiksų bip ilgį:

Tiesmukiškas būdas išspręsti viršutinę rekursiją yra taikyti dinaminį programavimą ir užpildyti dvimatę lentelę R[0..m][0..n].

		0	1	2	3	4	5	6	7	8	9
	X \ Y	ø	H	E	L	S	I	N	K	I	S
0	ø	0	0	0	0	0	0	0	0	0	0
1	V	0	0	0	0	0	0	0	0	0	0
2	I	0	0	0	0	0	$\boxed{1}$	1	1	1	1
3	L	0	0	0	$\boxed{1}$	1	1	1	1	1	1
4	N	0	0	0	1	1	1	$\boxed{2}$	2	2	2
5	I	0	0	0	1	1	$\boxed{2}$	2	2	$\boxed{3}$	3
6	U	0	0	0	1	1	2	2	2	3	3
7	S	0	0	0	1	$\boxed{2}$	2	2	2	3	$\boxed{4}$

Sužinojus algoritmo skaičiavimo taisykles yra lengva suprasti, kad ieškant sekų X ir Y bip ilgio realybėje mums nereikėtų registruoti kitų matricos M vietų negu tas, kur yra minimalūs sutapimai. Jeigu žinome bip ilgio žemesnę ribą iš anksto, galėtume ignoruoti net dalį iš minimalių sutapimų . Taigi, Vagnerio ir Fischerio algoritmas daro kiekvieną kartą O(mn) žingsnius rašant r(i,j) skaičius visoms prefiksų poroms – nepaisant sekų savybių. Nors beveik visų bip algoritmų sudėtingumas yra blogiausiai lygio O(mn), dažniausiai daug iš jų dirba praktiškai labai greičiau. Svarbiausia savybė, kalbant apie efektyvumą yra, kad algoritmas rinktų ir saugotų tik problemai esminę informaciją. Reikia išrinkti gerai tinkamą duomenų struktūrą, bet yra svarbu rūpintis taip pat tuo, kad struktūros pastatymas netaptų per brangus.

2. Dinaminis programavimas (15 sk. 358 psl.). Algoritmų sudarymo metodika (15.3 sk. 379-389 psl.) ir šios metodikos taikymas sprendžiant strypų pjaustymo uždavinį (15.1 sk. 379-389 psl.) (rekursinės lygtys, optimalaus sprendinio reikšmės ir struktūros radimas, algoritmo sudėtingumo radimas…).

Uždavinį galime skaidyti taip: atpjauti strypą, paimti atpjauto strypo kainą ir likusiam strypui apskaičiuoti optimalią kainą. Iš to gauname rekursinę formulę:

$$r_n = \max_{1 \le i \le n} (p_i + r_{n-i})$$

Apskaičiuoti optimalią reikšmę galime taip:

- Pradedame nuo vienetinio strypo ilgio, jo rastą kainą išsaugome
- Ieškome sekančio ilgio strypo optimalios kainos įvairiais būdais dalindami strypą ir buvusio ilgio strypo kainą imdami iš išsaugotų kainų masyvo. Kartojame kol pasieksime duoto strypo ilgį.

Sudėtingumas: kadangi skaičiuojame kiekvieno ilgio strypui kainą, padalinant visais įmanomais būdais kiekvieną strypą, bus naudojami du ciklai ir algoritmų sudėtingumas bus $\Theta(n^2)$.

$$r_n = \max(p_n, r_1 + r_{n-1}, r_2 + r_{n-2}, \dots, r_{n-1} + r_1)$$

arba

$$r_n = \max_{1 \le i \le n} (\underbrace{p_i}_{\text{Atpjauto gabalo kaina}} + \underbrace{r_{n-i}}_{\text{Likusio strypo optimali kaina}})$$

n-pradinio strypo ilgis, i-atpjauto ilgis, p-kaina atpjautam strypui, r-max kaina tam tikro ilgio strypui(kad ir supjaustytam)

Rekursinis sprendimas

```
CUT-ROD(p, n)
if n == 0
    return 0
q = -∞
for i = 1 to n
    q = max(q, p[i] + CUT-ROD(p, n - i))
return q
```

Sudėtingumas apskaičiuojamas iš formulės

$$T(n) = 1 + \sum_{j=0}^{n} T(j)$$

$$T(n) = 2^n$$

Dinaminio programavimo sprendimas

Didėjantis (iteracinis)

```
BOTTOM-UP-CUT-ROD(p, n)
let r[0..n] be a new array
r[0] = 0
for j = 1 to n          pradedame nuo vienetinio ilgio strypo ir jo ilgį palaipsniui didiname
    q = -∞
    for i = 1 to j
        q = max(q, p[i] + r[j - i])
    r[j] = q
return r[n]
```

*apskaičiuojame **j** ilgio strypo maksimalią kainą – rekursijos nereikia, nes visų trumpesnių ilgių strypų maksimalios kainos jau yra žinomos. Rastą maksimumą išsisaugome*

sudėtingumas $\Theta(n^2)$

3. Dinaminis programavimas (15 sk. 358 psl.). Algoritmų sudarymo metodika (15.3 sk. 379-389 psl.) ir šios metodikos taikymas sprendžiant Matricų sekos optimalaus dauginimo uždavinį (15.3

sk. 379-389 psl.) (rekursinės lygtys optimalaus sprendinio reikšmės ir struktūros radimas, algoritmo sudėtingumo radimas…).

4. Godūs algoritmai (16 sk. 414psl.). Maksimalios procesų aibės radimo uždavinys (16.1 sk. 415-418 psl.) ir jo sprendimas (rekursinės lygtys, optimalaus sprendinio reikšmės ir struktūros radimas, algoritmo sudėtingumo radimas…) (16.2 sk. 423-427 psl.).

Godūs algoritmai dažniausiai naudojami optimizavimo uždavinių sprendimui

Tipinio optimizavimo uždavinio sprendime, algoritmas turi daug žingsnių, kuriuose **reikia priimti vienokį ar kitokį sprendimą**

godūs algoritmai priima sprendimą, kuris tuo metu yra / atrodo „**geriausias**"

- gaunamas lokaliai geriausias sprendinys, tikintis, kad jis ves prie globaliai optimalaus sprendinio
- bendriniu atveju, godūs algoritmai **negarantuoja** optimalaus sprendinio, nors kai kuriems uždaviniams galima rasti ir optimalų sprendinį
- algoritmų sudarymo procedūra panaši į dinaminio programavimo algoritmų sudarymo eigą

Godaus algoritmo elementai
Principai tokie patys kaip ir dinaminio programavimo algoritmų.
1. Randama uždavinio optimali struktūra.
2. Sudaromas rekursinis sprendimas.
3. Parodoma, kad esant godžiai strategijai lieka tik vienas pagalbinis uždavinys.
4. Parodoma (įrodoma), kad godus pasirinkimas yra saugus.
5. Sudaromas rekursinis algoritmas, realizuojantis godžią strategiją.
6. Rekursinis algoritmas transformuojamas į iteracinį.

Procesų pasirinkimo uždavinys

Turim procesų aibę $S = \{a_1, a_2, \ldots, a_n\}$.

Šie procesai naudoja tą patį resursą laike $0 \le s_i < f_i < \infty$.

Uždavinys: reikia rasti maksimalų poaibį nepersidengiančių aibės S procesų, t. y.
$[s_i, f_i) \cap [s_j, f_j) = \varnothing$, kai $i \neq j$.

Pavyzdys

i	1	2	3	4	5	6	7	8	9	10	11
s_i	1	3	0	5	3	5	6	8	8	2	12
f_i	4	5	6	7	8	9	10	11	12	13	14

Atsakymas:

$\{a_1, a_4, a_8, a_{11}\}$,
arba
$\{a_2, a_4, a_9, a_{11}\}$.

Rekursine lygtis:

$$c[i,j] = \begin{cases} 0 & \text{if } S_{ij} = \emptyset, \\ \max\limits_{\substack{i<k<j \\ a_k \in S_{ij}}} \{c[i,k] + c[k,j] + 1\} & \text{if } S_{ij} \neq \emptyset. \end{cases}$$

Užduoties paaiškinimas: Duota procesų aibė S su procesų pradžios laikais(s masyvas) ir pabaigos laikais(f masyvas). Rasti maksimalią nepersidengiančių procesų aibę.

Sprendimas:

Susidarome matricą kurioje saugosime procesus; į matricą įdedame pirmą procesą(pradžios ir pabaigos laikus); nuo antro proceso ieškome pirmo sutikto proceso, kurio pradžios laikas yra didesnis arba lygus matricoje esančio(pirmojo) proceso pabaigos laikui; rastą procesą įrašome į matricą; toliau ieškome nuo paskutinio rasto proceso vietos; kartojame kol praeisime visus procesus.

Algoritme naudojame tik vieną ciklą, kuris eis per matricą vieną kartą, todėl sudėtingumas yra ϴ(n), t.y. procesų kiekis masyvuose s ir f.

Iteracinis godaus algoritmo realizavimas

```
GREEDY-ACTIVITY-SELECTOR(s, f)
n ← length[s]
A ← {a_1}
i ← 1
for m ← 2 to n
    do if s_m ≥ f_i
        then A ← A ∪ {a_m}
            i ← m
return A
```

Sudėtingumas $\Theta(n)$

Antra klausimų grupė (2 balai):

1. Paieškos į plotį algoritmas (22.2 sk. 594-597 psl.) ir sudėtingumo įvertinimas.

Paieška į plotį (*breadth-first search*, *BFS*) – paieškos grafe arba grafo apėjimo būdas, kai pasirinkus pradinę viršūnę pirmiausia yra aplankomos visos jos kaimynės (t. y., viršūnės, sujungtos grafo briaunomis su pradine viršūne), po to kaimynių kaimynės ir t. t., kol randamas ieškomas tikslas arba kol yra apeinamos visos grafo viršūnės ir briaunos.

```
BFS(G, s)
for each vertex u ∈ G.V − {s}
    u.color = WHITE
    u.d = ∞
    u.π = NIL
s.color = GRAY
s.d = 0
s.π = NIL
Q = ∅
ENQUEUE(Q, s)
while Q ≠ ∅
    u = DEQUEUE(Q)
    for each v ∈ G.Adj[u]
        if v.color == WHITE
            v.color = GRAY
            v.d = u.d + 1
            v.π = u
            ENQUEUE(Q, v)
    u.color = BLACK
```

Korektiška sakyti, jog sudėtingumas priklauso nuo viršūnių ir briaunų, todėl sudėtingumas yra **O(V + B)**. (Kiekvienai aplankytai viršūnei reikia suskaičiuoti gretimas viršūnes, o tai galima padaryti tik einant per briaunas)

2. Paieškos į gylį algoritmas (22.3 sk. 603-606 psl.) ir sudėtingumo įvertinimas.

Paieška į gylį (*depth-first search* ar *DFS*) – paieškos grafe arba grafo apėjimo būdas, kai pasirinkus pradinę viršūnę einama grafo briaunomis kiek įmanoma giliau, renkantis vis naują viršūnę; kai paskutinė aplankyta viršūnė naujos (dar neaplankytos) kaimynės neturi, tada grįžtama iki artimiausios neaplankytos briaunos ir vėl ieškoma kuo giliau tol, kol bus rastas ieškomas tikslas arba kol bus aplankytos visos grafo viršūnės ir briaunos.

Taikymų pavyzdžiai: patikrinti, ar grafas jungus, identifikuoti neorientuoto grafo jungumo komponentes, patikrinti, ar egzistuoja kelias grafe iš viršūnės u iki viršūnės v.

```
DFS(G)
for each vertex u ∈ G.V
    u.color = WHITE
    u.π = NIL
time = 0
for each vertex u ∈ G.V
    if u.color == WHITE
        DFS-VISIT(G, u)

DFS-VISIT(G, u)
time = time + 1                  // white vertex u has just been discovered
u.d = time
u.color = GRAY
for each v ∈ G.Adj[u]            // explore edge (u, v)
    if v.color == WHITE
        v.π = u
        DFS-VISIT(G, v)
u.color = BLACK                  // blacken u; it is finished
time = time + 1
u.f = time
```

norint nustatyti ar grafas jungus, jis bus jungus tuomet jei Lankyk(viršūnė) išoriniame cikle bus iškviestas tik vieną kartą, galima Lankyk iškviesti su bet kuria viršūne ir vėliau patikrinti, ar aplankytos visos viršūnės.

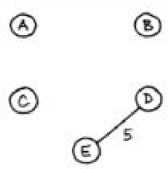
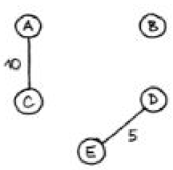
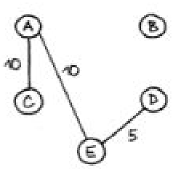
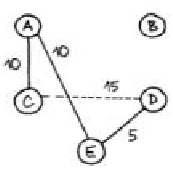
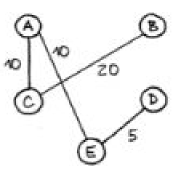
norint nustatyti trumpiausią kelią ne svoriniame grafe nuo duotosios viršūnės t iki visų grafo viršūnių. Tuomet Lankyk(I) procedūrą reikėtų šiek tiek praplėst

Korektiška sakyti, jog sudėtingumas priklauso nuo viršūnių ir briaunų, todėl sudėtingumas yra **O(V + B)**. (Kiekvienai aplankytai viršūnei reikia suskaičiuoti gretimas viršūnes, o tai galima padaryti tik einant per briaunas)

3. Kruskalo algoritmas (23.2 sk. 631-633 psl.) ir sudėtingumo įvertinimas.

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

Kruskalo algoritmo veikimo iliustracija

	Randama pigiausia briauna (jos kaina – 5) ir įtraukiama į MJM
	Pasirenkama kita pigiausia briauna (yra dvi tokios briaunos AC ir AE, imama bet kuri) ir įtraukiama į MJM
	Kita pigiausia briauną yra AE; ji įtraukiama į MJM
	Tolesnė pigiausia briauna yra CD (jos kaina 15), tačiau jos įtraukti į MJM negalima, nes susidarytų ciklas, tad ši briauna praleidžiama
	Prijungiama ketvirtoji pigiausia briauna (BC, jos kaina 20) ir gaunamas MJM; jo] kaina – 45

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

```
MST-KRUSKAL(G, w)
1  A ← Ø
2  for each vertex v ∈ V[G]
3      do MAKE-SET(v)
4  sort the edges of E into nondecreasing order by weight w
5  for each edge (u, v) ∈ E, taken in nondecreasing order by weight
6      do if FIND-SET(u) ≠ FIND-SET(v)
7            then A ← A ∪ {(u, v)}
8                 UNION(u, v)
9  return A
```

Pradžioje turime tuščią aibę. Turime išsirikuoti briaunas pagal svorį didėjimo tvarka.Tada, kol įmanoma, atliekama tokia operacija: iš visų briaunų, kurias įjungus prie jau parinktų, nesusidaro ciklas, išrenkama mažiausio svorio briauna. Kai tokių briaunų nebelieka, algoritmas baigia darbą. Tada pografis, kurį sudaro visos jo viršūnės ir surastos briaunos, sudaro grafo karkasinį medį.

Sudėtingumas

$O(E\log_2 V)$, nes

5–8 eil. vykdomo $O(E)$ paieškos ir jungimo operacijų aibių medyje nesikertančių aibių.

Kartu su $|V|$ aibių sukūrimu (2-3) trunka $O((E+V)\alpha(V))$, čia α labai lėtai auganti funkvija.

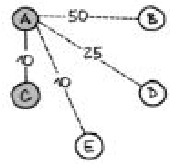
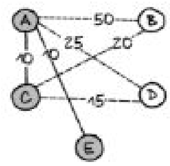
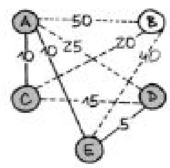
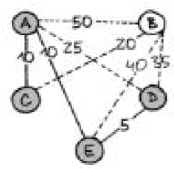
Kadangi grafas jungus $|E| \geq |V|-1$ todėl $O(E\alpha(V))$, $\alpha(V) = \log_2 V = \log_2 E$, tai sudėtingumas $O(E\log_2 E)$ (su 3 eil.).

Be to $|E| < |V|^2$, todėl $O(E\log_2 V)$.

4. Prima algoritmas (23.2 sk. 634-636 psl.) ir sudėtingumo įvertinimas

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

Primo algoritmo veikimo iliustracija

	Pasirenkame pradinę viršūnę (pavyzdžiui, A); matome, kad pigiausiai prie jos galime prijungti viršūnes C arba E; pasirenkame bet kurią - C
	Prie sudarinėjamo MJM, kuris kol kas turi dvi viršūnes A, C ir briauną tarp jų, pigiausiai galime prijungti viršūnę E (briaunos AE svoris 10)
	Toliau pigiausiai galima prijungti viršūnę D (briaunos svoris 5)
	Liko viena neprijungta viršūnė; ją pigiausiai galima prijungti briauna CB, jos svoris – 20; gauname 7 pav. pavaizduotą MJM

```
MST-PRIM(G, w, r)
    for each u ∈ G.V
        u.key = ∞
        u.π = NIL
    r.key = 0
    Q = G.V
    while Q ≠ ∅
        u = EXTRACT-MIN(Q)
        for each v ∈ G.Adj[u]
            if v ∈ Q and w(u, v) < v.key
                v.π = u
                v.key = w(u, v)
```

Efektyvumas priklauso nuo prioritetinės eilės realizacijos. Naudodami *binary heap* gausime O(|E| log |V|), naudodami *Fibonacci heap* gausime O(|E| + |V| log |V|).

5. Trumpiausi keliai iš vienos viršūnės (24 sk. 641-650 psl.). Relaksacijos metodas. Belmano –Fordo algoritmas (24.1 sk. 651 psl.). Sudėtingumo įvertinimas.

Trumpiausio kelio problema – grafų teorijos problema, bendru atveju formuluojama kaip radimas tokio kelio tarp dviejų svorinio grafo (arba daugiau) viršūnių, kad briaunų svorių suma būtų mažiausia.
Trumpiausiam kelyje ciklai negalimi. Jei yra ciklas teigiamas trumpiausiam kelyje, jį pašalinus rasime dar trumpesnį kelią. Nulinio svorio ciklus galime ignoruoti.

Relaksacijos metodas

Kiekvienai grafo $G = (V, E)$ viršūnei v priskirsime atributą $d[v]$, kuris nusakys viršutinę svorio ribą trumpiausio kelio iš pradinės viršūnės s iki v.

```
BELLMAN-FORD(G, w, s)
1  INITIALIZE-SINGLE-SOURCE(G, s)
2  for i = 1 to |G.V| - 1
3      for each edge (u, v) ∈ G.E
4          RELAX(u, v, w)
5  for each edge (u, v) ∈ G.E
6      if v.d > u.d + w(u, v)
7          return FALSE
8  return TRUE
```

Algoritmas rasti trumpiausią kelią iš viršūnės s iki visų kitų viršūnių, kai briaunų svoriai yra skirtingi.
Algoritmas gali atpažinti neigiamo svorio ciklo egzistavimą.

Efektyvumas -- O(V*E), nes O(V) - inicializacija in line 1, o O(E), nes for ciklas line 5-7

6. Trumpiausi keliai iš vienos viršūnės. Relaksacijos metodas. Deikstra algoritmas (24.3 sk. 658-659 psl.). Sudėtingumo įvertinimas (24.3 sk. 661-662 psl.).

Trumpiausio kelio problema – grafų teorijos problema, bendru atveju formuluojama kaip radimas tokio kelio tarp dviejų svorinio grafo (arba daugiau) viršūnių, kad briaunų svorių suma būtų mažiausia.

Trumpiausiam kelyje ciklai negalimi. Jei yra ciklas teigiamas trumpiausiam kelyje, jį pašalinus rasime dar trumpesnį kelią. Nulinio svorio ciklus galima ignoruoti.

Relaksacijos metodas

Kiekvienai grafo $G=(V,E)$ viršūnei v priskirsime atributą $d[v]$, kuris nusakys viršutinę svorio ribą trumpiausio kelio iš pradinės viršūnės s iki v.

Deikstra algoritmas

```
DIJKSTRA(G, w, s)
INITIALIZE-SINGLE-SOURCE(G, s)
S ← Ø
Q ← V[G]
while Q ≠ Ø
    do u ← EXTRACT-MIN(Q)
       S ← S ∪ {u}
       for each vertex v ∈ Adj[u]
           do RELAX(u, v, w)
```

Sudėtingumas $O(V^2+E)=O(V^2)$.

Grafas gali būti orientuotas arba neorientuotas, negali būti neigiamu viršūnių grafas turi būti jungus.
Inicializuojama pradzios virsune
Sukuriama struktura žymėti aplankytom virsunem, S virsune pažymima kaip aplankyta.
Kol yra neaplankytu virsuniu vykdoma:
Čia naudojama prioritetinė eilė. Operacija extract_min randa elementą u, kurio svoris (distance[u]) yra mažiausias.
Ir kiekviena virsunej atliekamas relaksacijos metodas
Ir viršūnė pažymima kaip aplankyta
Gaunamas sudetingumas O(V^2).

7. Trumpiausių kelių paieška iš vienos viršūnės orientuotame acikliniame grafe ir sudėtingumo įvertinimas. (24.2 sk. 665 psl.).

Algoritmas:

```
DAG-SHORTEST-PATHS(G, w, s)
1  topologically sort the vertices of G
2  INITIALIZE-SINGLE-SOURCE(G, s)
3  for each vertex u, taken in topologically sorted order
4      for each vertex v ∈ G.Adj[u]
5          RELAX(u, v, w)
```

Algoritmas rasti trumpiausią kelią iš viršūnės s iki visų kitų viršūnių, kai briaunų svoriai yra skirtingi.
Grafas turi būti beciklis (DAG'as).
1 eilute visa grafiką paverčia topologiniu grafu (O(V+E)).2 eilute užtrunka O(v). 3-5 iteruoja kiekvieną viršūnę. Susumavus 4-5 eilutės atpalaiduoja kiekvieną kraštinę viena karta. Kiekvienas vidinis for ciklas užtrunka Ө(1)laiko, tad sudetingumas O(V+E).

```
RELAX(u, v, w)
if d[v] > d[u] + w(u, v)
    then d[v] ← d[u] + w(u, v)
         π[v] ← u
```

Trečia klausimų grupė (4 balai):
8. Daugiagijo dinaminio programavimo metodika. (27 sk. 772-791 psl.)

Daugiagijis (lygiagretus) programavimas

- Strategijos
1. Kiekvienam procesoriui/branduolui atskira atmintis (distributed memory)
2. Visiems procesoriui/branduoliams bendra atmintis (shared memory)

Paskirstyta atmintis - kiekvieno procesoriaus atmintis yra privati ir turi būti siunčiama atvira žinutė tarp procesorių kad būtų galima pasiekti kito procesoriaus atmintį.

Bendrai naudojama atmintis - kiekvienas procesorius gali tiesiogiai prieiti bet kurią atminties vietą.

Daugiagijo programavimo metodai/principai

- Statinėmis gijomis – visos programos gyvavimo metu išlieka gijos, t. y. mažai kinta gijų skaičius.
- Dinaminėmis gijomis – leidžia programuotojui nurodyti lygiagretinimo lygį labai nekeičiant programos kodo ir nesirūpinant apkrovos balansavimu, komunikavimu…

Daugiagijis dinaminis programavimas leidžia programeriams apibrėžti lygiagretumą aplikacijose per daug negalvojant apie komunikacijų protokolus, apkrovos balansavimą ir apie kitas statinių gijų programavimo užgaidas.

- Parallel – ciklo iteracijų vykdymas vienu metu.
- spawn – naujos gijos/proceso paleidimas.
- sync – gijų/procesų sinchronizavimas.

Pagrindiniai terminai naudojami DDP - pavyzdys - Fibonačio skaičiaus skaičiavimas.
Gijos - virtualių procesorių programų abstrakcija
- Gijos dalinasi bendra atmintimi
- Prižiūri savo stack'ą ir programų skaičių
- Vykdo kodą nepriklausomai nuo kitų gijų
- OS užkrauna giją į procesorių, įvykdo jį ir pakeičia į kitą giją kai to reikia.

Lygiagrečios platformos:
- Nested lygiagretumas - leidžia įjungti paprogrames
- Planuoja - turi schedulerį (automatinis load-balance skaičiavimas)
- Turi lygiagrečius ciklus - ciklų iteracijos gali būti vykdomos lygiagrečiai

9. Daugiagijai matricų dauginimo algoritmai ir jų vykdymo laikų bei lygiagretinimo koeficientų įvertinimas. (27.2 sk. 792-797 psl.)

```
SQUARE-MATRIX-MULTIPLY(A, B)
n = A.rows
let C be a new n × n matrix
for i = 1 to n
    for j = 1 to n
        c_ij = 0
        for k = 1 to n
            c_ij = c_ij + a_ik · b_kj
return C
```

$$c_{ij} = \sum_{k=1}^{n} a_{ik} \cdot b_{kj}$$

Sudėtingumas: $\Theta(n^3)$

Tekstinis variantas, kuris padėjo gauti 10:

Pasileidžiame du ciklus nuo 1 iki n (i ir j kintamieji) tada c matricos elemnta i,j verciamę i nulį, darome trečia ciklą K nuo 1 iki n. Tada c i,j elemntas lygus c i,j sudečiai su a i,k ir b k,j sandaugai. Sudėtinguma matot.

Lygiagretaus programavimo:

**** lygiagretus***

```
P-SQUARE-MATRIX-MULTIPLY(A, B)
n = A.rows
let C be a new n × n matrix
parallel for i = 1 to n
    parallel for j = 1 to n
        c_ij = 0
        for k = 1 to n
            c_ij = c_ij + a_ik · b_kj
return C
```

$$T_\infty(n) = \Theta(n)$$

Viskas tas pats kaip ir paprastam tik parareliškai paleidžiam ciklus i ir j, taip sumažindami iki teta(n) sudetingumo, jis krenta, nes realiai tų ciklų sudėtingumas virsta 1.

Išlygiagretinimo Rezultatas: $\Theta(n^3)/\Theta(n) = \Theta(n^2)$.

Skaldyk ir valdyk:

Kvadratinių matricų daugyba – skaldyk ir valdyk

```
SQUARE-MATRIX-MULTIPLY-RECURSIVE(A, B)
n = A.rows
let C be a new n × n matrix
if n == 1
    c11 = a11 · b11
else partition A, B, and C as in equations (4.9)
    C11 = SQUARE-MATRIX-MULTIPLY-RECURSIVE(A11, B11)
          + SQUARE-MATRIX-MULTIPLY-RECURSIVE(A12, B21)
    C12 = SQUARE-MATRIX-MULTIPLY-RECURSIVE(A11, B12)
          + SQUARE-MATRIX-MULTIPLY-RECURSIVE(A12, B22)
    C21 = SQUARE-MATRIX-MULTIPLY-RECURSIVE(A21, B11)
          + SQUARE-MATRIX-MULTIPLY-RECURSIVE(A22, B21)
    C22 = SQUARE-MATRIX-MULTIPLY-RECURSIVE(A21, B12)
          + SQUARE-MATRIX-MULTIPLY-RECURSIVE(A22, B22)
return C
```

Informatikos fakultetas

Prisikiriame n a eilučių kiekį. Tada susikuriame naują matricą C, kur bus n x n dydžio. Pasirašome rekursijos stabdį, kai n= 1, tuo atveju c elemntas lygus a ir b elemento sandaugai, kitu atveju dalinamę a,b,c matricas į n/2*n/2 dydžio matricas, tad realiai gauname iš vienos matricos 4. Ir tada kviečiame reukrsine sudėti su skirtingomis matricu dalių sudėtimis. Pasibaigus visam rekursijos procesui turesime apjungti C dalys į viena vientisą n x n dydžio matricą. Taip gausime pilnai sudaugintą matricą.

$$T(1) = \Theta(1)$$

$$\begin{aligned} T(n) &= \Theta(1) + 8T(n/2) + \Theta(n^2) \\ &= 8T(n/2) + \Theta(n^2) . \end{aligned}$$

$\boxed{\text{Sudėtingumas: } \Theta(n^3)}$

Informatikos fakultetas

Lygiagretus:

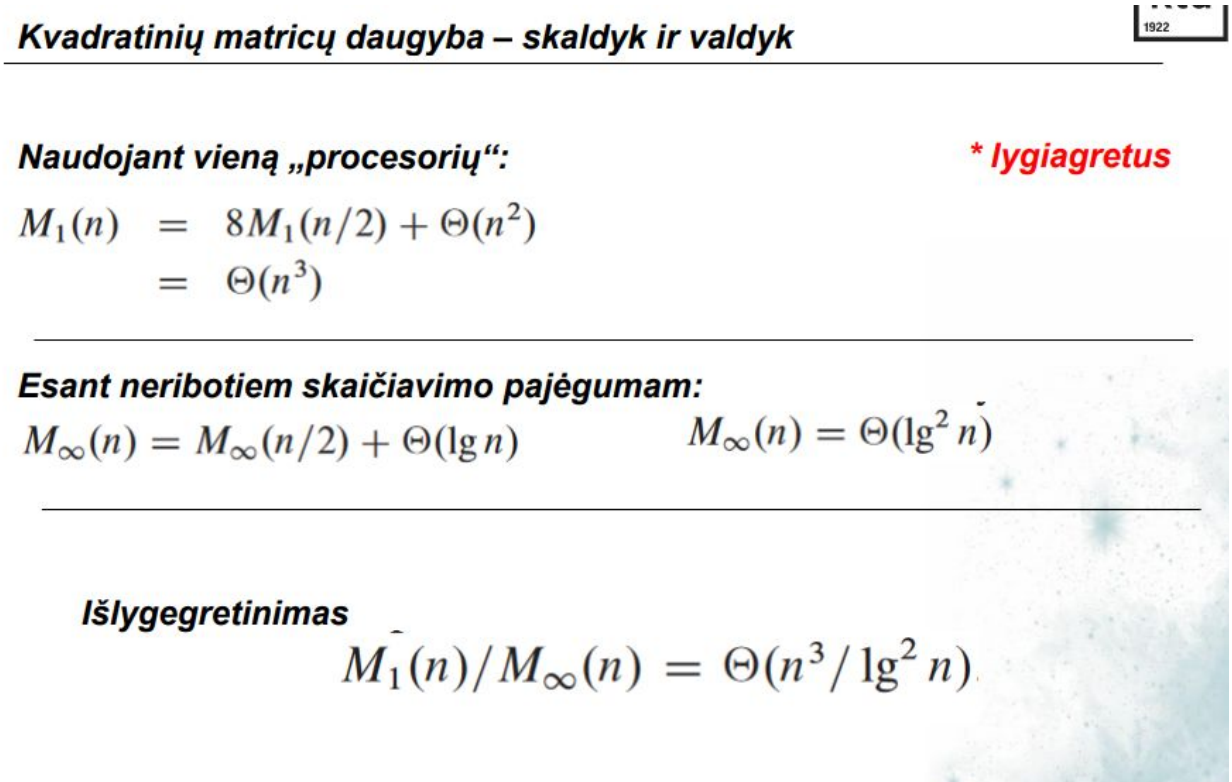
Kvadratinių matricų daugyba – skaldyk ir valdyk

*** lygiagretus**

```
P-MATRIX-MULTIPLY-RECURSIVE(C, A, B)
n = A.rows
if n == 1
    c11 = a11 b11
else let T be a new n × n matrix
    partition A, B, C, and T into n/2 × n/2 submatrices
        A11, A12, A21, A22; B11, B12, B21, B22; C11, C12, C21, C22;
        and T11, T12, T21, T22; respectively
    spawn P-MATRIX-MULTIPLY-RECURSIVE(C11, A11, B11)
    spawn P-MATRIX-MULTIPLY-RECURSIVE(C12, A11, B12)
    spawn P-MATRIX-MULTIPLY-RECURSIVE(C21, A21, B11)
    spawn P-MATRIX-MULTIPLY-RECURSIVE(C22, A21, B12)
    spawn P-MATRIX-MULTIPLY-RECURSIVE(T11, A12, B21)
    spawn P-MATRIX-MULTIPLY-RECURSIVE(T12, A12, B22)
    spawn P-MATRIX-MULTIPLY-RECURSIVE(T21, A22, B21)
    P-MATRIX-MULTIPLY-RECURSIVE(T22, A22, B22)
    sync
    parallel for i = 1 to n
        parallel for j = 1 to n
            cij = cij + tij
```

Informatikos fakultetas

Prisikiriame n reikšmei a eilučių kiekį. Tada susikuriame naują matricą T, kuri bus n x n dydžio. (C nebekuriame, nes vis persisiunčiame rekursiskai į metodą, tad jį turime susikurti pirma karta main metode).Pasirašome rekursijos stabdį, kai n= 1, tuo atveju c elemntas lygus a ir b elemento sandaugai, kitu atveju dalinamę a,b,c,t matricas į n/2*n/2 dydžio matricas. Susikuriame 8 metodus, kurie rekursiskai daugins matricas, pirmuose 4 siusime A ir B submatricas kartu su C submatricom, o kiti 4 metodai skaičiuos T submatricos reikšmes su A ir B submatricom, (lygegriačiai skaičiuos T ir C elementus). Po visų lygegriačių rekursijų gausime dvi matricas C ir T, kurios turės dalį rezultatų, tad juos turime apjungti į C matricą per dvigubą ciklą, kad apeitumę visą matricos dydį n x n (T matricą papildomai atsirantas, nes pvz C ims A11 B11, o T ims A12,B12, o C nebedarys ka darė T, taip paskirstis darbą procesoriui).



Čia mintinai išmokit, nes ką žinau kaip čia palengvinti.
Strassen metodas:

Kvadratinių matricų daugyba – Strassen algoritmas

1. Pradinių A, B ir rezultatų matricų C išskaldymas į $n/2 \times n/2$ dydžio matricas. $\boldsymbol{\Theta(1)}$.

2. Sukuriame 10 matricų $S_1, S_2, \dots, S_{10}$, kurių kiekviena yra $n/2 \times n/2$ dydžio. $\boldsymbol{\Theta(n^2)}$.

3. Panaudojant 2 žingsnyje sukurtas 1 ir 10 matricas, rekursyviai sudaromos 7 tarpinės matricos $P_1, P_2, \dots, P_7$. Kurių kiekviena yra $n/2 \times n/2$ dydžio.

4. Apskaičiuojamos $C_{11}, C_{12}, C_{21}, C_{22}$ sudedant skirtingas matricų P_i kombinacijas. $\boldsymbol{\Theta(n^2)}$.

Realiai parašiau kas viršuje nuotraukos ir apačioje esančias lygtis su paaiškinimu.

- vietoje 8 rekursinių kreipinių perduodant– atlieka 7 kreipinius (matricos išlieka $n/2 \times n/2$ dydžio)

- vienos daugybos panaikinimo kaina, kelios naujos $n/2 \times n/2$ matricų sudėtys

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1\,, \\ 7T(n/2) + \Theta(n^2) & \text{if } n > 1\,. \end{cases}$$

$$T(n) = \Theta(n^{\lg 7})$$

2. Sukuriame 10 matricų $S_1, S_2, \dots, S_{10}$, kurių kiekviena yra $\mathrm{n}/2 \times n/2$ dydžio. $\boldsymbol{\Theta(n^2)}$.

$$
\begin{aligned}
S_1 &= B_{12} - B_{22}\,, \\
S_2 &= A_{11} + A_{12}\,, \\
S_3 &= A_{21} + A_{22}\,, \\
S_4 &= B_{21} - B_{11}\,, \\
S_5 &= A_{11} + A_{22}\,, \\
S_6 &= B_{11} + B_{22}\,, \\
S_7 &= A_{12} - A_{22}\,, \\
S_8 &= B_{21} + B_{22}\,, \\
S_9 &= A_{11} - A_{21}\,, \\
S_{10} &= B_{11} + B_{12}\,.
\end{aligned}
$$

Neverta išmokti apačioje esančios ir viršuje esančios foto, nes čia rodo kaip veikia.

Kvadratinių matricų daugyba – Strassen algoritmas

3. Panaudojant 2 žingsnyje sukurtas 1 ir 10 matricas, rekursyviai sudaromos 7 tarpinės matricos $P_1, P_2, \dots, P_7$. Kurių kiekviena yra $\mathrm{n}/2 \times n/2$ dydžio.

$$
\begin{aligned}
P_1 &= A_{11} \cdot S_1 &&= A_{11} \cdot B_{12} - A_{11} \cdot B_{22}\,, \\
P_2 &= S_2 \cdot B_{22} &&= A_{11} \cdot B_{22} + A_{12} \cdot B_{22}\,, \\
P_3 &= S_3 \cdot B_{11} &&= A_{21} \cdot B_{11} + A_{22} \cdot B_{11}\,, \\
P_4 &= A_{22} \cdot S_4 &&= A_{22} \cdot B_{21} - A_{22} \cdot B_{11}\,, \\
P_5 &= S_5 \cdot S_6 &&= A_{11} \cdot B_{11} + A_{11} \cdot B_{22} + A_{22} \cdot B_{11} + A_{22} \cdot B_{22}\,, \\
P_6 &= S_7 \cdot S_8 &&= A_{12} \cdot B_{21} + A_{12} \cdot B_{22} - A_{22} \cdot B_{21} - A_{22} \cdot B_{22}\,, \\
P_7 &= S_9 \cdot S_{10} &&= A_{11} \cdot B_{11} + A_{11} \cdot B_{12} - A_{21} \cdot B_{11} - A_{21} \cdot B_{12}\,.
\end{aligned}
$$

4. Apskaičiuojamos $C_{11}, C_{12}, C_{21}, C_{22}$ sudedant skirtingas matricų P_i kombinacijas.

$$C_{11} = P_5 + P_4 - P_2 + P_6 \qquad C_{12} = P_1 + P_2$$

$$C_{21} = P_3 + P_4 \qquad C_{22} = P_5 + P_1 - P_3 - P_7$$

Same here, neverta mokytis.(Jei reiks gyvenime yra puslapis www.google.com, labai zjbs šaltinis, tačiau abidna, kad nepadėjo man išmokti rašyti be klaidų :( )

Lygiagretus:

1. Pradinių A, B ir rezultatų matricų C išskaldymas į $n/2 \times n/2$ dydžio matricas. $\boldsymbol{\Theta(1)}$. **** lygiagretus***

2. Sukuriame 10 matricų $S_1, S_2, \dots, S_{10}$, kurių kiekviena yra $n/2 \times n/2$ dydžio. $\boldsymbol{\Theta(n^2)}$. $\boldsymbol{\Theta(\ln n)}$ - panaudojant ***parralel for***.

3. Panaudojant 2 žingsnyje sukurtas 1 ir 10 matricas, rekursyviai sudaromos 7 tarpinės matricos $P_1, P_2, \dots, P_7$. Kurių kiekviena yra $n/2 \times n/2$ dydžio.

4. Apskaičiuojamos $C_{11}, C_{12}, C_{21}, C_{22}$ sudedant skirtingas matricų P_i kombinacijas. $\boldsymbol{\Theta(n^2)}$. $\boldsymbol{\Theta(\ln n)}$ - panaudojant ***parralel for***.

$$\boldsymbol{\Theta(n^{\ln 7}/\lg^2 n)}.$$

Informatikos fakultetas

Nu Čia irgi iškalt mintinai, tik apačioje parašytas teisingas sudėtingumas.
Parallel for duoda lg(n), o ne ln(n). (Nesugeba net nurašyti nuo knygos gerai :( )

Jei daskaitėt iki tiek, tai zjbs, jei ne tai bbz. Klauskit ko nesupratot iš šito uždavinio, nes geriausiai mokėjau iš jo.(Rašo žmogus, kuris įkėlė šį konspektą į FB)

10. Daugiagijai rikiavimo algoritmai ir jų vykdymo laikų bei lygiagretinimo koeficientų įvertinimas. . (27.3 sk. 797-804 psl.)

Rikiavimas suliejimu

```
MERGE-SORT'(A, p, r)
if p < r
    q = ⌊(p + r)/2⌋
    spawn MERGE-SORT'(A, p, q)
    MERGE-SORT'(A, q + 1, r)
    sync
    MERGE(A, p, q, r)
```

$$\begin{aligned} MS'_1(n) &= 2\,MS'_1(n/2) + \Theta(n) \\ &= \Theta(n \lg n) , \end{aligned}$$

$$\begin{aligned} MS'_\infty(n) &= MS'_\infty(n/2) + \Theta(n) \\ &= \Theta(n) . \end{aligned}$$

$$MS'_1(n)/MS'_\infty(n) = \Theta(\lg n)$$

Binarinio medzio:

```
BINARY-SEARCH(x, T, p, r)
low = p
high = max(p, r + 1)
while low < high
    mid = ⌊(low + high)/2⌋
    if x ≤ T[mid]
        high = mid
    else low = mid + 1
return high
```

```
P-MERGE(T, p1, r1, p2, r2, A, p3)
1   n1 = r1 − p1 + 1
2   n2 = r2 − p2 + 1
3   if n1 < n2                    // ensure that n1 ≥ n2
4       exchange p1 with p2
5       exchange r1 with r2
6       exchange n1 with n2
7   if n1 == 0                    // both empty?
8       return
9   else q1 = ⌊(p1 + r1)/2⌋
10      q2 = BINARY-SEARCH(T[q1], T, p2, r2)
11      q3 = p3 + (q1 − p1) + (q2 − p2)
12      A[q3] = T[q1]
13      spawn P-MERGE(T, p1, q1 − 1, p2, q2 − 1, A, p3)
14      P-MERGE(T, q1 + 1, r1, q2, r2, A, q3 + 1)
15      sync
```

1-2 eilutėmis apskaičiuojame T1 ir T2 masyvų ilgį. 3-6 eilutės perša mintį, kad n1 turi būti didesnė už n2. 9-15 eilutės naudoja skaldyk ir valdyk principą. 9 suranda vidurio taska T1 masyve, o T2 randa toki tašką, kuriame T2[p2..q2-1] yra mažiau nei T[q1] (kas yra X) ir taskus T2[q2..r2], kurie yra bent didesni už T[q1]. 11 padalina gautus masyvus į A[p3..q3-1] ir A[q3+1..r3] matricas ir 12 eilute perkopijuoja T[q1] tiesiai į A[q3]. Tada naudojama rekursiine nested *daugiagijyste.13 sprendžia viena subproblemą, o 14 sekančia subproblemą. 15 eilutė sinchronizuoja rezultatus prieš išeinant iš kodo.*
Tad gauname:

$$\begin{aligned} \lfloor n_1/2 \rfloor + n_2 &\le n_1/2 + n_2/2 + n_2/2 \\ &= (n_1 + n_2)/2 + n_2/2 \\ &\le n/2 + n/4 \\ &= 3n/4 . \end{aligned}$$

Dar pridedant, kad kvieciant binary-search kainuoja Θ(lg n) gauname (išsprendus medžio būdu):

$$PM_\infty(n) = \Theta(\lg^2 n).$$

Lygiagretumas lygus:

$$PM_1(n)/PM_\infty(n) = \Theta(n/\lg^2 n).$$

Kai išanalizavom suliejimo procedūrą, dabar galime jį pritaikyti daugiagijystės suliejimo rūšiavimui. Ši versiją yra panaši į merge sort, kuria dareme, tačiau šitoje versijoje B masyvo išvedimą priima kaip argumentą, kuris palaiko surušiuotą masyvą.

```
P-MERGE-SORT(A, p, r, B, s)
n = r − p + 1
if n == 1
    B[s] = A[p]
else let T[1..n] be a new array
    q = ⌊(p + r)/2⌋
    q' = q − p + 1
    spawn P-MERGE-SORT(A, p, q, T, 1)
    P-MERGE-SORT(A, q + 1, r, T, q' + 1)
    sync
    P-MERGE(T, 1, q', q' + 1, n, B, s)
```

Normalus darbas neįvertinant gijų:

$$\begin{aligned} PMS_1(n) &= 2\,PMS_1(n/2) + PM_1(n) \\ &= 2\,PMS_1(n/2) + \Theta(n) . \end{aligned}$$

$$PMS_1(n) = \Theta(n \lg n)$$

Gijų laiko įvertinimas:
(PMSinf(n/2), nėra 2PMSinf(n/2), nes naudojama gijas ir jos vyks tuo pačiu metu)

$$\begin{aligned} PMS_\infty(n) &= PMS_\infty(n/2) + PM_\infty(n) \\ &= PMS_\infty(n/2) + \Theta(\lg^2 n) . \end{aligned}$$

$$PMS_\infty(n) = \Theta(\lg^3 n),$$

P-MERGE SORT lygiagretinimo koeficientas:

$$\begin{aligned} PMS_1(n)/PMS_\infty(n) &= \Theta(n \lg n)/\Theta(\lg^3 n) \\ &= \Theta(n/\lg^2 n) , \end{aligned}$$

11. Amortizacinė algoritmų analizė. (17 sk. 451-462 psl.)

Amortizacinė analizė (17 sk.)

Tikslas surasti viršutinį laiko įvertį, reikalingą atlikti seką veiksmų su duomenų struktūrą. Tai atliekama vidurkinant visų operacijų laiką.

Tai leidžia parodyti, kad net esant „brangioms“ operacijoms, jos nedaro didelės įtakos.

Metodai

- Grupinė analizė (aggregate analysis)
- Buhalterinė apskaita (accounting method)
- Potencialų metodas (potential method)

Grupinė analizė

Randama $T(n)$ viršutinis įvertis n operacijų. Vidurkis $\frac{T(n)}{n}$ laikomas amortizaciniais kaštais kiekvienos operacijos

Stekas

Trys operacijos: push, pop, multipop

```
MULTIPOP(S, k)
while not STACK-EMPTY(S) and k > 0
    POP(S)
    k = k - 1
```

Sudėtingumai:

$$T_{push}(s) = O(1)$$
$$T_{pop}(s) = O(1)$$
$$T_{multi}(s, k) = O(\min(s, k)) = O(k)$$

Sekos iš n minėtų operacijų viršutinis įvertis lygus $O(n^2)$, nes steke gali būti nedaugiau kaip n elementų.

Per grubus įvertinimas, nes galima išimti tik jau įdėtą elementą. Tokui atveju n operacijų laiką galima įvertinti, kaip $O(n)$. Šiuo atveju amortizacinė operacijos kaina lygi $O(1)$.

Binarinis skaitliukas

```
INCREMENT(A)
i = 0
while i < A.length and A[i] == 1
    A[i] = 0
    i = i + 1
if i < A.length
    A[i] = 1
```

Procedūros viršutinis įvertis yra $O(k)$, čia k– skaitliuke esančių bitų skaičius, taigi n veiksmų sudėtingumas $O(nk)$.

Šis įvertinimas būtų tikslesnis jei mes skaičiuotume kiek kartų keitėme bitus iš 0 į 1.

$$\sum_{i=0}^{k-1} \left\lfloor \frac{n}{2^i} \right\rfloor = \sum_{i=0}^{\infty} \left\lfloor \frac{n}{2^i} \right\rfloor = 2n = O(n)$$

Taigi *Increment* procedūros amortizaciniai kaštai $\frac{O(n)}{n} = O(1)$.

Buhalterinė apskaita

Agregatinis metodas tiesiogiai nustato bendrą operacijų sekos vykdymo laiką. Priešingai, apskaitos metodu siekiama rasti papildomų laiko vienetų, mokamų kiekvienai atskirai operacijai, sumokėti taip, kad mokėjimų suma būtų viršutinė riba bendrų visų išlaidų. Intuityviai galima galvoti apie banko sąskaitos tvarkymą. Pigios operacijos yra apmokestinamos šiek tiek daugiau negu jos iš tikro kainuoja, o perteklius yra pervedamas į banko sąskaitą vėlesniam naudojimui. Tada brangios operacijos gali būti apmokestinamos mažesnėmis už jų tikras išlaidas, o deficitas mokamas iš taupymo banko sąskaitoje. Tokiu būdu mes paskirstome didelių sąnaudų operacijų išlaidas per visą seka. Kiekvienos operacijos mokesčiai turi būti tokie dideli, kad banko sąskaitos balansas visada išliktų teigiamas, bet pakankamai mažas, kad nė viena operacija nebūtų apmokestinta žymiai daugiau už jo tikras išlaidas.

Mes pabrėžiame, kad papildomas operacijos laikas nereiškia, kad operacija užtrunka tiek daug laiko. Tai tik apskaitos metodas, kuris leidžia lengviau analizuoti.

Potencialų metodas

Tarkim, kad galime apibrėžti potencialų funkciją Φ duomenų struktūros būsenoje su tokiomis savybėmis:

Φ(h0) = 0, kur h0 yra pradinė duomenų struktūros būsena

Φ(h0) >= 0 visoms būsenoms ht duomenų struktūrai visoje skaičiavimo eigoje.
Intuityviai, potenciali funkcija seks iš anksto nustatytą laiką viso skaičiavimo metu. Ji nustato kiek sutaupyto laiko yra prieinama apmokėti brangioms operacijoms. Šis metodas yra analogiškas buhalterinei apskaitai. Įdomiausia tai, kad funkcija priklauso tik nuo dabartines duomenų struktūros būsenos, nepriklausomai nuo skaičiavimo istorijos kaip ji pateko į tą būseną.

Tada nustatome amortizuojama operacijos laiką kaip

$$c + \Phi(h') - \Phi(h),$$

Kur c - tikra operacijos kaina, o h ir h` yra duomenų struktūros būsenos prieš ir po operacijos.
Idealiai Φ turi būti nustatytas taip, kad kiekvienos operacijos amortizuojamas laikas būtų mažas.

Dinaminiams, neapibrėžto dydžio, masyvams su dvigubu padidinimu, galime naudoti potencialų funkciją

$$\Phi(h) = 2n - m,$$

Kur n yra dabartinis elementų skaičius, o m dabartinis masyvo ilgis. Jei pradėsim nuo masyvo ilgio 0 ir nustatysim jo ilgį 1, kai pirmas elementas yra pridėtas, o vėliau padvigubinsim masyvą reikės daugiau vietos, turim Φ(ht) = 0 ir Φ(ht) >= 0 visoms t. Paskutinė nelygybė veikia, nes elementų skaičius yra visada bent pusė masyvo dydžio.

Pridedant elementus yra naudojama amortizuojamo laiko konstanta. Yra 2 atvejai:

- Jei n < m, tada tikra kaina yra 1, n padidėja per 1, o m nesikeičia. Tada potencialas padidėja 2, o amortizuojamas laikas yra 1 + 2 = 3
- Jei n = m, tada masyvas yra padvigubintas, tada tikras laikas yra n+1. Bet potencialas sumažėjo nuo n iki 2, todėl amortizavimo laikas yra n + 1 + (n-2) = 3.

Abejais atvejais, amortizavimo laikas yra O(1).
Esminis dalykas potencialų metode yra nustatyti teisingą potencialų funkciją. Potencialų funkcija turi sutaupyti pakankamai laiko tolimesniam naudojimui, kai to reikia. Bet ji neturi sutaupyti tiek laiko kad amortizuojamas laikas dabartinei operacijai būtų per didelis.

12. NP sudėtingumas. Ir P ir NP klasės. Uždavinių pavyzdžiai. (34 sk. 1048-1053 psl.)

NP–pilnumas bei P ir NP uždavinių klasės

P uždavinių klasė – uždaviniai, kurie sprendžiami per polinominę laiko trukmę $O(n^k)$, čia k – konstanta, o n – įvedamų duomenų kiekis.

NP uždavinių klasė – uždaviniai, kurie pasiduoda *patikrinimui* per polinominę laiko trukmę. Tai yra jei, kokiu nors būdu buvo gautas sprendinio *sertifikatas*, tai jo teisingumą galima patikrinti per polonominę laiko trukmę tokio sprendinio korektiškumą.

$P \subseteq NP$, nes P – uždavinio sprendinys gaunamas per polinominį laiką, net ir neturint sprendinio sertifikato. Uždavinys yra NP pilnas, jei jis priklauso NP uždavinių klasei ir toks pat *sudėtingas* kaip ir bet kuris kitas NP uždavinys.

Taigi jei egzistuoja bent vienam NP pilnam uždaviniui polinominis sprendimo algoritmas, tai bet kuriam kitam šios klasės uždaviniui egzistuoja toks algoritmas.

Jau nagrinėjome kelis tokius uždavinius:
1. Keliaujančio pirklio uždavinys. Duotas svertinis grafas G = (V,E), reikia rasti trumpiausią pirklio maršrutą, kai jis po vieną kartą aplanko visas grafo viršūnes ir grįžta į pradinę viršūnę.
2. Hamiltono ciklas. Reikia patikrinti ar duotajame grafe G = (V,E) egzistuoja ciklas, jungiantis visas jo viršūnes.
3. Diskretusis kuprinės užpildymo uždavinys. Turime n daiktų, kurių tūriai yra v1,v2,... ,vn, o kaina p1,p2,... ,pn. Reikia rasti tokį daiktų rinkinį, kuris tilptų į V tūrio kuprinę ir krovinio vertė būtų didžiausia.
4. Dėžių užpildymo uždavinys. Turime keletą vienetinio tūrio dėžių ir n daiktų, kurių dydžiai v1, v2,..., vn, čia 0 < vj < 1. Šiuos daiktus reikia sudėti į kuo mažesnį skaičių dėžių.