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...).
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...).
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. 414 psl.). 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.).

Antra klausimų grupė (2 balai):

1. Paieškos į plotį algoritmas (22.2 sk. 594-597 psl) ir sudėtingumo įvertinimas.
2. Paieškos į gylį algoritmas (22.3 sk. 603-606 psl.) ir sudėtingumo įvertinimas.
3. Kruskalo algoritmas (23.2 sk. 631-633 psl.) ir sudėtingumo įvertinimas.
4. Prima algoritmas (23.2 sk. 634-636 psl.) ir sudėtingumo įvertinimas.
5. Trumpiausi keliai iš vienos viršūnės (24 sk. 641-650 psl.). Relaksacijos metodas. Belmano – Fordo algoritmas (24.1 sk. 651 psl.). Sudėtingumo įvertinimas.
6. Trumpiausi keliai iš vienos viršūnės. Relaksacijos metodas. Deikstra algoritmas (24.3 sk. 658-659 psl.). Sudėtingumo įvertinimas (24.3 sk. 661-662 psl.).
7. Trumpiausių kelių paieška iš vienos viršūnės orientuotame acikliniame grafe ir sudėtingumo įvertinimas (24.2 sk. 665 psl.).

Trečia klausimų grupė (4 balai):

8. Daugiagijo dinaminio programavimo metodika. (27 sk. 772-791 psl.).
9. Daugiagijai matricų dauginimo algoritmai ir jų vykdymo laikų bei išlygiagretinimo koeficientų įvertinimas. (27.2 sk. 792-797 psl.).
10. Daugiagijai rikiavimo algoritmai ir jų vykdymo laikų bei lygiagretinimo koeficientų įvertinimas. (27.3 sk. 797-804 psl.).
11. Amortizacinė algoritmų analizė. (17 sk. 451-462 psl.).
12. NP sudėtingumas. Ir P ir NP klasės. Uždavinių pavyzdžiai. (34 sk. 1048-1053 psl.).

Pirma klausimų grupė (4 balai):

1. Dinaminis programavimas.

- Dinaminis programavimas yra tarytum skaldyk ir valdyk metodas – jis sprendžia problemas sujungdamas išskaidytas problemos subproblemas į vieną visumą.

- Dinaminis programavimas skiriasi nuo skaldyk ir valdyk tuo, kad skaldyk ir valdyk išskaido problemas į viena nuo kitos atskirtas subproblemas, o tuo tarpu dinaminis programavimas per-panaudoja vienos subproblemos sprendimus sprendžiant kitas sub-problemas.

- Šiame kontekste skaldyk ir valdyk algoritmas atlieka daugiau darbo negu būtina, kadangi tų pačių subproblemų subproblemos yra sprendžiamos keletą kartų, kai tuo tarpu, dinaminis programavimas tokias subsubproblemas išsprendžia tik vieną kartą ir sprendimą toliau saugo duomenų lentelėje.

- Dažniausiai dinaminis programavimas yra taikomas optimizavimo uždaviniuose. Tokie uždaviniai gali turėti daug galimų sprendimo būdų.

- Kuriant dinaminio programavimo algoritmą yra sekami šie žingsniai:
  1. Charakterizuojama optimalaus sprendimo struktūra.
  2. Sprendimo vertė yra apibūdinama rekursiškai.
  3. Apskaičiuojama optimali problemos sprendimo vertė, dažniausiai iš apačios į viršų (angl. bottom-up) būdu.
  4. Sukurti optimalų sprendimo būdą pagal apskaičiuotą informaciją.

- **1-3** žingsniai suformuoja problemos dinaminio programavimo sprendimo bazę. Jeigu yra reikalaujama tik optimalaus sprendimo vertės, o ne pačio sprendimo, tuomet galima praleisti **4** žingsnį.

Algoritmų sudarymo metodika.

- Norint taikyti dinaminį programavimo metodika, optimizavimo uždavinys turi turėti 2 pagrindines savybes: **optimalią struktūrą** ir **persidengiančias subproblemas.**
- **Norint rasti optimalia struktūrą reikia**
  1. Parodyti, kad problemos sprendimas susidaro iš tam tikro pasirinkimo, pvz. pasirenkant pradinį strypo įpjovimą ar indekso, ties kuriuo bus padalinama matrica. Atliekant ši pasirinkimą turi atsirasti 1 ar daugiau subproblemų, kurias reikės išspręsti.

  2. Numanoma, kad esamai problemai yra pateiktas pasirinkimas, kuris nuves prie optimalaus sprendimo. Po kol kas nesirūpinama kokiu būdu nustatyti tokį pasirinkimą. Numanoma, kad toks pasirinkimas tau buvo duotas.

3. Turint ši pasirinkimą yra nustatoma, kokios subproblemos seks po to ir kaip bus geriausia charakterizuoti atsiradusią subproblemų erdvę.

4. Parodoma, kad subproblemų sprendimai, naudojami optimaliame problemos sprendime, patys turi būti optimalūs pasitelkiant „cut-and-paste“ techniką. Tai parodoma darant prielaidą, kad kiekvienos subproblemos sprendimas nėra optimalus ir taip iškeliant prieštaravimą. Konkrečiai tariant, neoptimalus sprendimas yra „iškerpamas„ (angl. „cutting out“) iš kiekvienos subproblemos ir vietoje jo yra „įklijuojamas“ (angl „pasting in“) optimalus sprendimas. Taip yra parodoma, kad galima gauti geresnį sprendimą originaliai iškeltai problemai ir paprieštaraujama prielaidai, kad optimalus sprendimas jau rastas. Jeigu optimalus sprendimas iškelia daugiau nei 1 subproblemą, tada dažniausiai jie būna tokie panašūs, kad be didelių pastangų galima modifikuoti „cut-and-paste“ argumentą, tam, kad jis galiotų ir kitiems sprendimams.

- **Persidengiančios subproblemos**
- Optimizavimo problemos naujų subproblemų erdvė neturi būti didelė, tam, kad rekursinis algoritmas vis spręstų tas pačias problemas, o ne kurtų naujas.
- Kai dinaminis algoritmas pakartotinai aplanko tą pačią problemą – yra sakoma, kad problema turi **persidengiančias subproblemas.**

Šios metodikos taikymas sprendžiant Konvejerio (Surinkimo linijos planavimo) arba Bendro ilgiausio posekio radimo uždavinį.

1. **Bendro ilgiausio posekio (BIP) charakterizavimas**
   - Žymėjimas: jei yra seka $\mathrm{X} = \{x_1, x_2, \dots, x_m\}$ tai $X_i = \{x_1, x_2, \dots, x_i\}$
     Pvz. jeigu $\mathrm{X} = \{\mathrm{A}, \mathrm{B}, \mathrm{C}, \mathrm{D}, \mathrm{D}, \mathrm{A}, \mathrm{B}\}$ tai $X_4 = \{\mathrm{A}, \mathrm{B}, \mathrm{C}, \mathrm{D}\}$

   - Turime sekas $\mathrm{X} = \{x_1, x_2, \dots, x_m\}$ ir $\mathrm{Y} = \{y_1, y_2, \dots, y_n\}$ ir bet kurį šių sekų BIP $\mathrm{Z} = \{z_1, z_2, \dots, z_k\}$

     1. Jei $x_m = y_n$, tada $z_k = x_m = y_n$ ir Z_{k-1} yra sekų X_{m-1} ir Y_{n-1} BIP.
     2. Jei $x_m \neq y_n$, tada $z_k \neq x_m$ leidžia manyti, kad Z yra sekų X_{m-1} ir Y BIP.
     3. Jei $x_m \neq y_n$, tada $z_k \neq y_n$ leidžia manyti, kad Z yra sekų X ir Y_{n-1} BIP.

2. **Rekursinis sprendimas**
   - Pagal 1-3 charakterizuotą punktą galime susidaryti rekursinę problemos sprendimo lygtį, kurioje $\mathrm{c}[\mathrm{i}, \mathrm{j}]$ bus sekų X_i *ir* Y_j bendro ilgiausio posekio ilgis.
   - $$c[i,j] = \begin{cases} 0, & jei\ i = 0\ arba\ j = 0, \\ c[i-1, j-1] + 1, & jei\ i,j > 0\ ir\ x_i = y_j, \\ \max(c[i, j-1], c[i-1, j]), & jei\ i,j > 0\ ir\ x_i \neq y_j. \end{cases}$$

3. Suskaičiuojamas bendro ilgiausio posekio ilgis

- Remiantis 2 punkte sudaryta rekursine yra sudaromas dinaminiu programavimu paremtas sprendimas.

```
LCS-LENGTH(X, Y)
 1  m = X.length
 2  n = Y.length
 3  let b[1..m, 1..n] and c[0..m, 0..n] be new tables
 4  for i = 1 to m
 5      c[i, 0] = 0
 6  for j = 0 to n
 7      c[0, j] = 0
 8  for i = 1 to m
 9      for j = 1 to n
10          if x_i == y_j
11              c[i, j] = c[i − 1, j − 1] + 1
12              b[i, j] = "↖"
13          elseif c[i − 1, j] ≥ c[i, j − 1]
14              c[i, j] = c[i − 1, j]
15              b[i, j] = "↑"
16          else c[i, j] = c[i, j − 1]
17              b[i, j] = "←"
18  return c and b
```

- Funkcija LCS-LENGTH paima 2 sekas X ir Y kaip parametrus.
- Funkcija saugo c[i, j] vertes lentelėje c[0..m, 0..n].
- Vertės šioje matricoje yra užpildomos iš viršaus į apačią ir iš kairės į dešinę.
- Funkcija taipogi pildo lentelę b[0..m, 0..n], tam, kad galėtume surasti optimalų sprendimą.
- b[0..m, 0..n] rodyklė rodo į atitinkamą lentelėje esantį optimalų subproblemos sprendimą, kuris buvo rastas skaičiuojant c[i, j].
- c[i, j] saugomas X ir Y sekų BIP.
- Procedūra užtrunka Θ(nm), kadangi kiekvieno lentelės įrašo suradimas užtrunka Θ(1), o galimų kombinacijų yra nm.

4. Sukonstruojamas bendras ilgiausias posekis

	j	0	1	2	3	4	5	6
i		y_j	B	D	C	A	B	A
0	x_i	0	0	0	0	0	0	0
1	A	0	↑ 0	↑ 0	↑ 0	↖ 1	← 1	↖ 1
2	B	0	↖ 1	← 1	← 1	↑ 1	↖ 2	← 2
3	C	0	↑ 1	↑ 1	↖ 2	← 2	↑ 2	↑ 2
4	B	0	↖ 1	↑ 1	↑ 2	↑ 2	↖ 3	← 3
5	D	0	↑ 1	↖ 2	↑ 2	↑ 2	↑ 3	↑ 3
6	A	0	↑ 1	↑ 2	↑ 2	↖ 3	↑ 3	↖ 4
7	B	0	↖ 1	↑ 2	↑ 2	↑ 3	↖ 4	↑ 4

- LCS-Length funkcijos gražinama b lentelė leidžia greitai surasti sekų X ir Y BIP.

Kvadratėlyje esantis skaičius rodo c[i, j] vertę, b[i, j] – rodyklę.

- Langelyję c[6, 7] esanti vertė rodo BIP sekos ilgį.
- Pradedame nuo b[n,m] ir keliaujame lentele sekdami rodykles. Kai b[i, j] randama „↖” rodyklė, tai reiškia, kad $x_i = y_j$ yra BIP esantis elementas.

```
PRINT-LCS(b, X, i, j)
1  if i == 0 or j == 0
2      return
3  if b[i, j] == "↖"
4      PRINT-LCS(b, X, i − 1, j − 1)
5      print x_i
6  elseif b[i, j] == "↑"
7      PRINT-LCS(b, X, i − 1, j)
8  else PRINT-LCS(b, X, i, j − 1)
```

- Norint rekonstruoti BIP seką tereikia sekti rodyklėmis iš apačios į viršų ir iš kairės į dešinę. Tai atlieka funkcija PRINT-LCS, pradinis jos iškvietimas PRINT-LCS(b, X, X.length, Y,length).
- Ši procedūra užtrunka O(m + n), kadangi su lyg kiekvienų rekursiniu žingsniu yra bent kartą sumažinamos i ir j reikšmės.

2. Dinaminis programavimas. atsakyta 1 klausime.

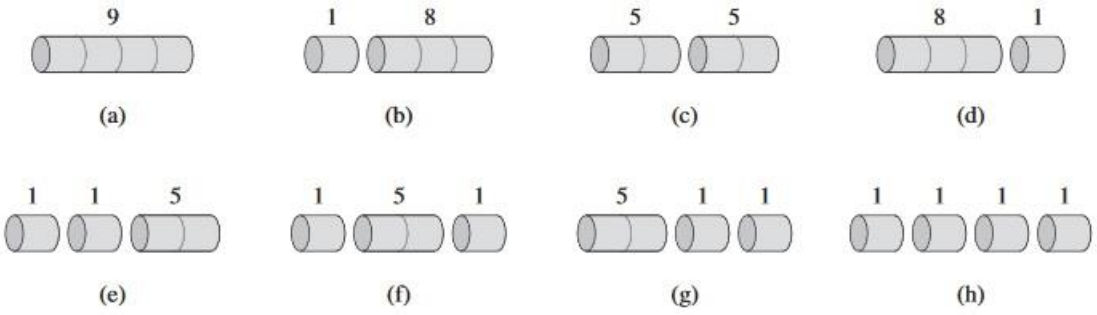
Algoritmų sudarymo metodika. atsakyta 1 klausime.

Šios metodikos taikymas sprendžiant strypų pjaustymo uždavinį.

1. Sprendimo charakterizavimas

- Strypų pjaustymo uždavinio esmė – esant n ilgumo strypui ir kainų lentelėj p_i, apskaičiuoti maksimalų pelną r_n supjaustant tą strypą ir parduodant jo dalis. Reikia turėti omenyje, kad kartais norint gauti maksimalų pelną gali neprireikti nei vieno pjūvio.
- Strypą, kurio ilgis yra n galime supjaustyti 2^{n-1} skirtingais būdais.

length i	1	2	3	4	5	6	7	8	9	10
price p_i	1	5	8	9	10	17	17	20	24	30



- Toks sprendimas gali būti išreiškiamas lygtimi $r_n = \max(p_n, r_1 + r_{n-1}, r_2 + r_{n-2}, \dots, r_{n-1} + r_1)$. Pirmasis argumentas - p_n reiškia tai, kad gali neprireikti pjaustyti strypo.
- Norint rasti r_n mums tenka išspręsti identiškas problemas, tik mažesnio dydžio, pvz. $r_1 + r_{n-1}$
- Sprendžiant uždavinį galima pasitelkti šiek tiek paprastesnę formuluotę, kada strypas dalinamas į dvi dalis ir kairioji jo pusė vėliau nebėra pjaustoma, pjaustome tik dešinę pusę
- Gaunama paprastesnė lygtis $r_n = \max_{1 \le i \le n}(p_i + r_{n-1})$

2. Rekursinis iš viršaus į apačią problemos sprendimas

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

- Argumentas p yra strypo kainų masyvas, o n yra strypo ilgis.
- Sprendimas labai lėtas, kadangi metodas pastoviai kviečia save su tokias pačiais parametrais.
- Laiko kaštai auga eksponentiškai priklausomai nuo strypo ilgio.
- Algoritmo sparta yra $T(n) = 1 + \sum_{j=0}^{n-1} T(j)$, todėl gauname, kad $T(n) = 2^n$

3. Dinaminio programavimo pasitelkimas problemai išspręsti

- Pasitelkiame dinaminio programavimo savybę naudoti papildomą atmintį norint laimėti skaičiavimo laiko.
- Dinaminio programavimo sprendimo laikas polinomiškai priklauso nuo strypo ilgio, kadangi kiekviena subproblema yra sprendžiama tik kartą, o subproblemų kiekis polinomiškai priklauso nuo strypo ilgio.
- Dinaminį algoritmą galime panaudoti dviem būdais – iš viršaus į apačią bei iš apačios į viršų.
- Iš viršaus į apačią sprendimui mes pasitelkiame tą patį rekursinį CUT-ROD algoritmą išskyrus tai, kad visus rastų subproblemų sprendinius įsimename.
- Iš apačios į viršų metode mes išrikiuojame problemas pagal dydį ir sprendžiame jas nuo mažiausios iki didžiausios.
- Abu algoritmai užtrunka vienodą laiką, tačiau iš apačios į viršų metodas turi geresnes laiko konstantas kadangi jam reikia atlikti mažiau procedūrų.

- CUT-ROAD procedūros iš viršaus į apačią dinaminio algoritmo pseudokodas.

```
MEMOIZED-CUT-ROD(p, n)
1  let r[0..n] be a new array
2  for i = 0 to n
3      r[i] = -∞
4  return MEMOIZED-CUT-ROD-AUX(p, n, r)
```

```
MEMOIZED-CUT-ROD-AUX(p, n, r)
1  if r[n] ≥ 0
2      return r[n]
3  if n == 0
4      q = 0
5  else q = -∞
6      for i = 1 to n
7          q = max(q, p[i] + MEMOIZED-CUT-ROD-AUX(p, n - i, r))
8  r[n] = q
9  return q
```

- Iš apačios į viršų algoritmas yra kur kas paprastesnis.

```
BOTTOM-UP-CUT-ROD(p, n)
1  let r[0..n] be a new array
2  r[0] = 0
3  for j = 1 to n
4      q = -∞
5      for i = 1 to j
6          q = max(q, p[i] + r[j - i])
7      r[j] = q
8  return r[n]
```

- Asimptotinis šių algoritmų laikas yra $\Theta(n^2)$, kadangi „iš apačios į viršų“ algoritme yra ciklas cikle, o „iš viršaus į apačią“, kadangi rekursinis kvietinys šiame metode kiekvieną problemą sprendžia tik vieną kartą.

4. Problemos sprendimo rekonstravimas

- Viršuje paminėti algoritmai gražina optimalaus sprendimo vertę, tačiau negražina pačio sprendimo – supjaustyto strypo dalių.
- Galima papildyti šiuos algoritmus, kad jie saugotų ne tik vertę, bet ir strypų ilgius, kurie privedė prie tokio rezultato.

```
EXTENDED-BOTTOM-UP-CUT-ROD(p, n)
1  let r[0..n] and s[0..n] be new arrays
2  r[0] = 0
3  for j = 1 to n
4      q = -∞
5      for i = 1 to j
6          if q < p[i] + r[j - i]
7              q = p[i] + r[j - i]
8              s[j] = i
9      r[j] = q
10 return r and s
```

- Šis metodas papildomai saugo s masyvą, kuriame saugomas optimalus pirmosios nukirptos strypo dalies ilgis – i.

- Žemiau esantis metodas kaip argumentus paima kainų lentelę – p ir strypo ilgį – n ir išspausdina pilną supjaustytų strypų sąrašą.

```
PRINT-CUT-ROD-SOLUTION(p, n)
1  (r, s) = EXTENDED-BOTTOM-UP-CUT-ROD(p, n)
2  while n > 0
3      print s[n]
4      n = n − s[n]
```

In our rod-cutting example, the call EXTENDED-BOTTOM-UP-CUT-ROD(p, 10) would return the following arrays:

i	0	1	2	3	4	5	6	7	8	9	10
$r[i]$	0	1	5	8	10	13	17	18	22	25	30
$s[i]$	0	1	2	3	2	2	6	1	2	3	10

- Kreipimasis PRINT-CUT-ROD-SOLUTION(p, 10) išspausdintų tik 10, tačiau kreipimasis, kuriame n = 7 išspausdintų strypo ilgius - 1 ir 6, kadangi
  - s[7] = 1, toliau n = 7 – 1 = 6
  - s[6] = 6, toliau n = 6 – 6 = 0

3. Dinaminis programavimas. atsakyta 1 klausime.

Algoritmų sudarymo metodika. atsakyta 1 klausime.

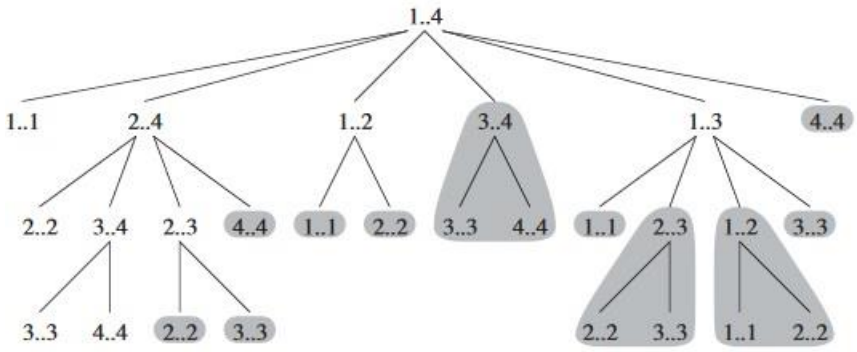
Šios metodikos taikymas sprendžiant Matricų sekos optimalaus dauginimo uždavinį.

1. **Problemos charakterizavimas**
   - Dvi matricas tarpusavyje galime sudauginti tik tuomet, kai vienos matricos stulpelių kiekis sutampa su kitos matricos eilučių kiekiu.

2. **Rekursinis problemos sprendimas**
   - Matricų dauginimo algoritmas (kuris bbz kur nx yra toje knygoje) ieško subproblemų sprendimo žemesnėse eilutėse, kada yra sprendžiamos aukščiau esančių eilučių subproblemos.
   - Pvz., jis kreipiasi į m[3, 4] keturis kartus, kada yra skaičiuojama m[2, 4], m[1, 4], m[3, 5] ir m[3, 6]
   - Rekursinis metodas, vietoje to, kad perpanaudotų subproblemų sprendimus, juos skaičiuoja kiekvieną kartą iš naujo kas drastiškai išaugina laiko kaštus.
   - Žemiau pateiktas rekursinis algoritmas ieškantis matricų sekos sandaugos atlieka tiek veiksmų: $A_{i..j} = A_i A_{i+1} \dots A_j$

```
RECURSIVE-MATRIX-CHAIN(p, i, j)
1  if i == j
2      return 0
3  m[i, j] = ∞
4  for k = i to j − 1
5      q = RECURSIVE-MATRIX-CHAIN(p, i, k)
           + RECURSIVE-MATRIX-CHAIN(p, k + 1, j)
           + p_{i−1} p_k p_j
6      if q < m[i, j]
7          m[i, j] = q
8  return m[i, j]
```

- Šis metodas su kreipiniu RECURSIVE-MATRIX-CHAIN(p, 1, 4) sugeneruoja štai tokį rekursijos medį (papilkinti laukai iliustruoja dinaminio programavimo sugeneruotą medį)



- Kiekvienas mazgas apibūdinamas parametrais i ir j. Kai kurios poros pasikartoja daugybe kartų.
- Galima įrodyti, kad toks algoritmas eksponentiškai priklauso nuo matricų skaičiaus.

$$T(1) \geq 1,$$
$$T(n) \geq 1 + \sum_{k=1}^{n-1}(T(k) + T(n-k) + 1) \qquad \text{for } n > 1.$$

Noting that for $i = 1, 2, \ldots, n-1$, each term $T(i)$ appears once as $T(k)$ and once as $T(n-k)$, and collecting the $n-1$ 1s in the summation together with the 1 out front, we can rewrite the recurrence as

$$T(n) \geq 2\sum_{i=1}^{n-1} T(i) + n. \tag{15.8}$$

We shall prove that $T(n) = \Omega(2^n)$ using the substitution method. Specifically, we shall show that $T(n) \geq 2^{n-1}$ for all $n \geq 1$. The basis is easy, since $T(1) \geq 1 = 2^0$. Inductively, for $n \geq 2$ we have

$$\begin{aligned} T(n) &\geq 2\sum_{i=1}^{n-1} 2^{i-1} + n \\ &= 2\sum_{i=0}^{n-2} 2^i + n \\ &= 2(2^{n-1} - 1) + n \quad \text{(by equation (A.5))} \\ &= 2^n - 2 + n \\ &\geq 2^{n-1}, \end{aligned}$$

which completes the proof. Thus, the total amount of work performed by the call RECURSIVE-MATRIX-CHAIN$(p, 1, n)$ is at least exponential in n.

- Naudojant dinaminį programavimą ir taikant iš viršaus į apačią sprendimą, galime pasinaudoti persidengiančiomis problemomis.
- Matricų sekos daugyba gali turėti ne daugiau $\Theta(n^2)$ skirtingų subproblemų, o dinaminis programavimas kiekvieną problemą sprendžia vieną kartą.

3-4. Optimalaus sprendimo rekonstravimas

- Kiekvieną apskaičiuotą subproblemą saugome į duomenų lentelę, kad mums daugiau nebereikėtų jos perskaičiuoti.
- Tada, kiekvieną kartą, kai pakartotinai prireiks subproblemos sprendimo, jį ištraukti galėsime per O(1) laiką.
- Žemiau pateiktas algoritmas yra RECURSIVE-MATRIX-CHAIN metodo versija su atmintimi.

```
MEMOIZED-MATRIX-CHAIN(p)
1  n = p.length − 1
2  let m[1..n, 1..n] be a new table
3  for i = 1 to n
4      for j = i to n
5          m[i, j] = ∞
6  return LOOKUP-CHAIN(m, p, 1, n)
```

```
LOOKUP-CHAIN(m, p, i, j)
1  if m[i, j] < ∞
2      return m[i, j]
3  if i == j
4      m[i, j] = 0
5  else for k = i to j − 1
6          q = LOOKUP-CHAIN(m, p, i, k)
               + LOOKUP-CHAIN(m, p, k + 1, j) + p_{i−1} p_k p_j
7          if q < m[i, j]
8              m[i, j] = q
9  return m[i, j]
```

- MEMOIZED-MATRIX-CHAIN procedūra saugo lentelę m[1..n, 1..n] su apskaičiuotomis vertėmis iš m[i, j], viršuje esančiame mazgų medyje papilkinti mazgai iliustruoja kaip šis metodas sutaupo laiko lyginant su rekursine jo versija.
- Kiekviena pradinė m reikšmė yra ∞, tam, kad parodytu, jog vertė dar neužpildyta
- Kviečiant LOOKUP-CHAIN(m, p, i, j), jei 1 eilutėja nustatoma, kad m[i, j] < ∞, tuomet gražinama prieš tai suskaičiuota m[i, j] vertė.
- Priešingu atveju yra vertė surandama analogiškai rekursiniam metodui ir išsaugoma m[i, j] vietoje, todėl šis metodas kiekvienos subproblemos sprendimo ieško tik kartą.
- MEMOIZED-MATRIX-CHAIN užtrunka $O(n^3)$ laiko. Šio metodo 5-oji eilutė vykdoma $\Theta(n^2)$ kartų.
- Galime kategorizuoti metodo LOOKUP-CHAIN kvietimus į 2 tipus:
    1. Kai m[i, j] = ∞, tada vykdomos 3-9 eilutės.
    2. Kai [m, i] < ∞, tada metodas iškart gražina 2-oje eilutėje.
- Iš viso 1-ojo tipo kreipinių yra $\Theta(n^2)$. Visi 2-ojo tipo kreipiniai yra atliekami kaip 1-ojo tipo rekursiniai kreipiniai. Kai 2-ojo tipo kreipinys atlieka rekursinį metodo kvietimą, jis tai daro O(n) kartų. Todėl iš viso yra $O(n^3)$ 2-ojo tipo kreipinių.
- Kiekvienas 1-ojo tipo kreipinys užtrunka O(n) + rekursiniai kreipiniai.
- Kiekvienas 2-ojo tipo kreipinys užtrunka O(1).
- Taigi, bendras gaunamas laikas yra $O(n^3)$. Subproblemų sprendimas $\Omega(2^n)$ algoritmą padaro $O(n^3)$.

4. Godūs algoritmai:

Optimizavimo problemoms spręsti skirti algoritmai paprastai pereina žingsnių seką, kuri kiekviename žingsnyje turi pasirinkimų aibę. Daugeliui optimizavimo problemų naudoti dinaminį programavimą, kad būtų rastas geriausias pasirinkimas, yra overkillas (nežinau kaip išversti padoriai). Godus algoritmas visad priima sprendimą, kuris tuo momentu atrodo geriausias. Jis priimant lokaliai optimalų sprendimą tikisi, kad tai ves link globaliai optimalaus sprendimo. Bendru atveju, godūs algoritmai negarantuoja optimalaus sprendimo, nors kai kuriems uždaviniams galima rasti ir optimalų sprendinį.

Maksimalios procesų aibės radimo uždavinys:

Sąlyga: Reikia paskirstyti resurso prieigą keliems konkuruojantiems procesams (vienu metu resursu gali naudotis tik vienas procesas). Kuriems procesams priskirti resurso prieigą, kad atliktų procesų (užduočių) skaičius būtų maksimalus?

Duomenys:

- $\boldsymbol{S} = \{\boldsymbol{a1}, \boldsymbol{a2}, \dots, \boldsymbol{an}\}$ – procesų (užduočių) aibė
- $\boldsymbol{si}$ ir $\boldsymbol{fi}$ – $\boldsymbol{i}$-tojo proceso pradžios ir pabaigos laikas, $0 \leq \boldsymbol{si} < \boldsymbol{fi} < \infty$
- Procesai nesikerta jei intervalai $[\boldsymbol{si}, \boldsymbol{fi})$ir $(\boldsymbol{sj}, \boldsymbol{fj}]$ nepersidengia.
- Procesai pateikti surikiuoti didėjimo tvarka pagal pabaigos laiką: $\boldsymbol{f1} \leq \boldsymbol{f2} \leq \cdots \leq \boldsymbol{f}_n$

Duomenų pvz.:

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	9	9	10	11	12	14	16

Šiam pavyzdžiui poaibiai {a3, a9, a11} susideda iš abipusiai suderinamų veiklų. Tai nėra maksimalus poaibis, tačiau poaibis (a1, a4, a8, a11} yra didesnis ir taip pat didžiausias poaibis iš abipusiai suderinamų veiklų; kitas didžiausias poaibis yra {a2, a4, a9, a11}.

a_i imti iš šitos vietos:

i	1	2	3	4	5	6	7	8	9	10	11

Ir žiūrėkit ar intervalai po apačia nesikerta, pvz: {a3, a9, a11}

3	9	11
0	8	12
6	12	16

Nesikerta, nes aibės [0,6); [8,12); [12;16)

Problema sprendžiama keliais žingsniais. Pradedame galvodami apie dinaminio programavimo būdą, kuriame mes apsvarstome kelis pasirinkimus kai bandome išsiaiškinti, katrą subproblemą naudoti optimaliame sprendime. Turime suprasti, kad mums reikia tik vieno pasirinkimo – godaus ir kai jį pasirinksime, liks tik viena subproblema.

Jo sprendimas:

Activity-selection problem.

Rekurentinė formulė:

This way of characterizing optimal substructure suggests that we might solve the activity-selection problem by dynamic programming. If we denote the size of an optimal solution for the set S_{ij} by $c[i, j]$, then we would have the recurrence

$$c[i, j] = c[i, k] + c[k, j] + 1 .$$

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

Žmonių kalba, tai turi n veiklų su jų pradžios ir pabaigos laikais. Reikia pasirinkti maksimalų kiekį veiklų, kurios gali būti atliktos vieno asmens, turint omeny, kad asmuo vienu metu gali atlikti tik vieną užduotį. Godus algoritmas visada pasirinks sekančią veiklą, kurios pabaigos laikas yra mažiausias tarp likusių veiklų ir kurios pradžios laikas yra didesnis arba lygus praeitos veiklos pabaigos laikui.

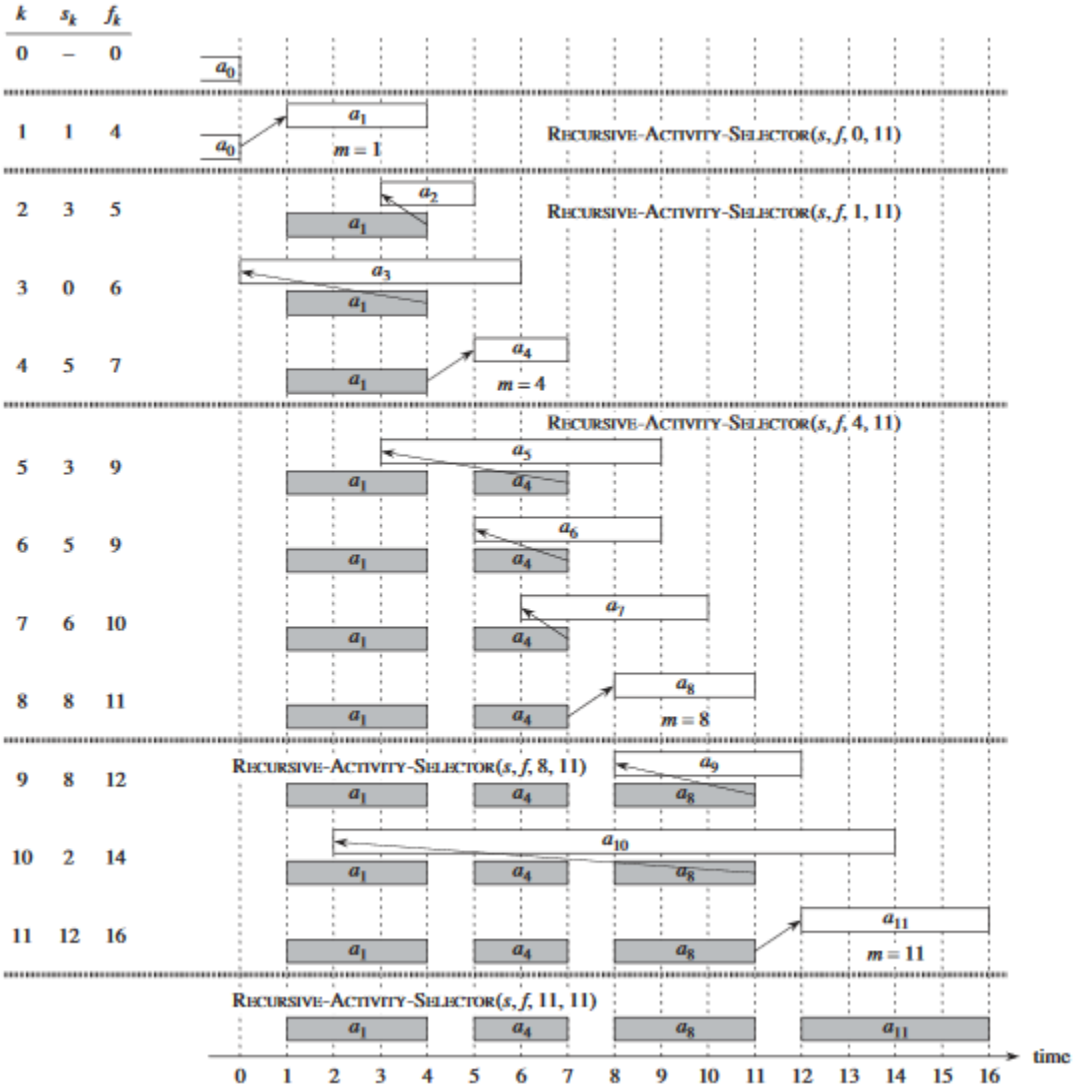
Galima išrikiuoti veiklas pagal jų pabaigos laiką, pasirinkti pirmąją veiklą iš masyvo ir taip eiti paeiliui. Išrikiuotam masyve sudėtingumas **O(n)**, neišrikiuotam – **O(nlogn)**

Recursive activity selector. Turime masyvus s ir f, indeksą k, kuris apibrėžią subproblemą S_k ir problemos dydį n. Ji grąžina maksimalaus dydžio abipusiškai suderinamų veiklų aibę S_k. Laikysime, kad n veiklų yra išrikiuotos pagal pabaigos laiką didėjimo tvarka.

```
RECURSIVE-ACTIVITY-SELECTOR(s, f, k, n)
1  m = k + 1
2  while m ≤ n and s[m] < f[k]        // find the first activity in S_k to finish
3      m = m + 1
4  if m ≤ n
5      return {a_m} ∪ RECURSIVE-ACTIVITY-SELECTOR(s, f, m, n)
6  else return ∅
```

Arba

```
GREEDY-ACTIVITY-SELECTOR(s, f)
1  n = s.length
2  A = {a_1}
3  k = 1
4  for m = 2 to n
5      if s[m] ≥ f[k]
6          A = A ∪ {a_m}
7          k = m
8  return A
```



Atsakymas: {a1, a4, a8, a11}

Taigi žingsniai tokie:

- Nustatyti optimalią substruktūrą problemai
- Sukurti rekursinį sprendimą
- Įrodyti, kad jei pasirenkam godų pasirinkimą, lieka tik viena subproblema
- Įrodome, kad visada saugu pasirinkti godų būdą
- Sukuriam rekursinį algoritmą, kuris implementuoja godžią strategiją
- Konvertuojam rekursinį algoritmą į įteracinį

Antra klausimų grupė (2 balai):

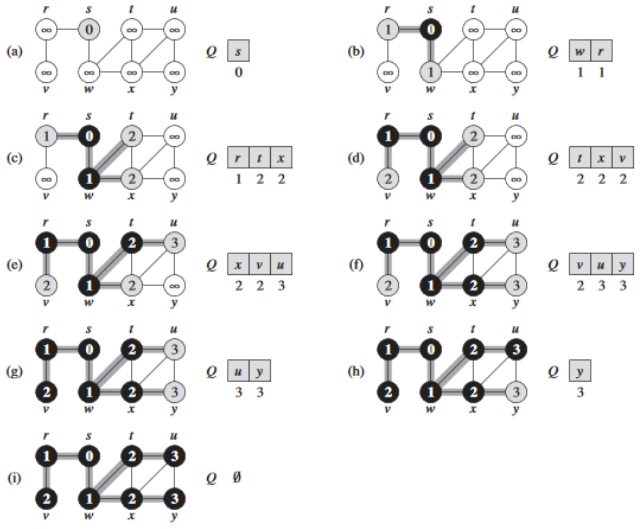
1. Paieškos į plotį algoritmas ir sudėtingumo įvertinimas:

https://www.youtube.com/watch?v=zaBhtODEL0w

Kad algoritmas veiktų, jam reikia grafo ir pradinės viršūnės s. Algoritmas atranda kiekvieną viršūnę, pasiekiamą iš viršūnės s, taip pat apskaičiuoja atstumą iki visų viršūnių. Kiekviena viršūnė nudažoma baltai (jei dar yra neatrasta), pilkai (jei yra atrasta ir įdėta į eilę) arba juodai (jei yra jau patikrinta). Kiekvieną kartą iš eilės imama viršūnė ir tikrinama, ar ji turi baltų gretimų viršūnių. Jei taip – viršūnė nudažoma pilkai ir įdedama į eilės galą.

Formulė: G = (V, E), kur V – vertexes (viršūnės) ir E – edges (briaunos).

```
BFS(G, s)
1  for each vertex u ∈ G.V − {s}
2      u.color = WHITE
3      u.d = ∞
4      u.π = NIL
5  s.color = GRAY
6  s.d = 0
7  s.π = NIL
8  Q = ∅
9  ENQUEUE(Q, s)
10 while Q ≠ ∅
11     u = DEQUEUE(Q)
12     for each v ∈ G.Adj[u]
13         if v.color == WHITE
14             v.color = GRAY
15             v.d = u.d + 1
16             v.π = u
17             ENQUEUE(Q, v)
18     u.color = BLACK
```



Paaiškinimas:

1-4 eilutės nudažo kiekvieną viršūnę, išskyrus s (pradinę), baltai, atstumą iki jų nustato begalybę, o tėvinę viršūnę – NIL. 5-7 eilutės nudažo pradinę viršūnę s pilkai, nustato atstumą iki jos 0 ir tėvinę viršūnę NIL. 8-9 eilutės sukuria eilę ir įdeda į ją pradinę viršūnę s. While ciklas 10-18 eilutėse iteruoja tol, kol yra pilkų viršūnių ir kiekvieną kartą išima iš eilės viršūnę u, patikrina visas jai gretimas viršūnes ir, jei randa baltą, įdeda ją į eilę, nudažo pilkai, priskiria atstumą u.d + 1 bei tėvinę viršūnę u. Kai viršūnės u visos gretimos viršūnės yra patikrintos, ji nudažoma juodai.

Sudėtingumo įvertinimas:

Viršūnės įtraukimas į eilę arba išėmimas iš jos užtrunka O(1), tai vyksta daugiausiai kartą vienai viršūnei, todėl visas darbo su eile laikas yra O(V). Kadangi funkcija tikrina visas gretimas viršūnes tik tada, kai tikrinama viršūnė išimama iš eilės, reiškia, kad tokį tikrinimą programa atlieka tik kartą. Taigi, iš to išeina, kad bendras grafo gretimų viršūnių tikrinimo laikas yra O(E). Pridėtinės laiko išlaidos inicializacijai yra O(V), todėl paieškos į plotį algoritmas užtrunka **O(V + E)**.

2. Paieškos į gylį algoritmas ir sudėtingumo įvertinimas:

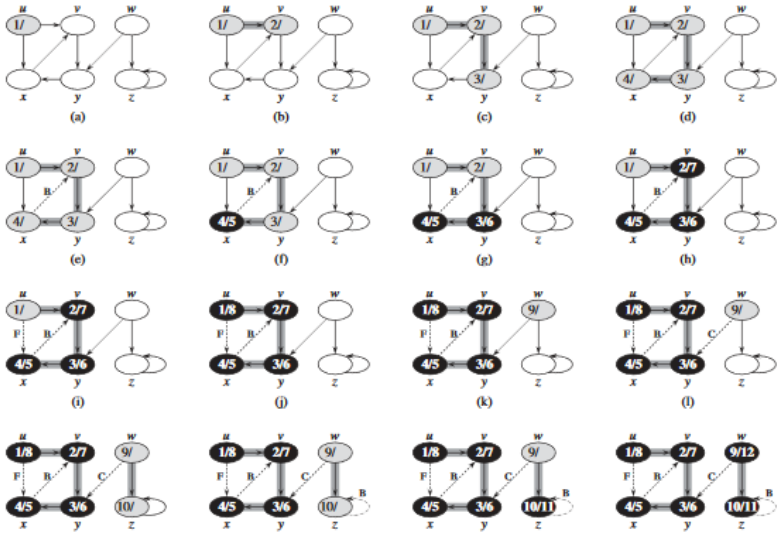
https://www.youtube.com/watch?v=zaBhtODEL0w

Paieška į gylį eina nuo vėliausiai atrastos viršūnės, tikrindama, ar ta viršūnė dar turi neatrastų viršūnių. Jei randama neatrasta viršūnė – ji įrašoma į eilę ir tampa tikrinamąja viršūne. Taip einama gylyn tol, kol neberandama gretimų neatrastų viršūnių. Tokiu atveju einame atgal į tą viršūnę, iš kurios atėjome ir tikriname toliau. Kaip ir paieškoje į plotį, čia naudojame spalvas: balta (neatrastas viršūnė), pilka (atrasta viršūnė), juoda (viršūnė, kurios gretimos viršūnės visos atrastos). Kiekviena viršūnė turi parametrus: v.d – parodo, kada viršūnė buvo atrasta (nudažyta pilkai), v.f – parodo, kada viršūnė buvo išsisėmusi (nudažyta juodai), kur visada v.d < v.f.

Formulė: G = (V, E), kur V – vertexes (viršūnės) ir E – edges (briaunos).

```
DFS(G)
1  for each vertex u ∈ G.V
2      u.color = WHITE
3      u.π = NIL
4  time = 0
5  for each vertex u ∈ G.V
6      if u.color == WHITE
7          DFS-VISIT(G, u)

DFS-VISIT(G, u)
1  time = time + 1
2  u.d = time
3  u.color = GRAY
4  for each v ∈ G.Adj[u]
5      if v.color == WHITE
6          v.π = u
7          DFS-VISIT(G, v)
8  u.color = BLACK
9  time = time + 1
10 u.f = time
```



Paaiškinimas:

1-3 eilutėse visos viršūnės nudažomos baltai ir jų tėvinės viršūnės nustatomos į NIL. 4 eilutė nustato, kad pradinis laikas lygus 0. 5-7 eilutės tikrina kiekvieną viršūnę ir, jei randa baltą, eina į ją per DFS-VISIT funkciją, kad galėtų tikrinti toliau. Šios funkcijos 1-3 eilutės padidina laiką vienetu, priskiria atradimo (u.d) laiką ir nudažo viršūnę u pilkai. 4-7 eilutės tikrina gretimas viršūnes viršūnei u ir, jei randa baltą, nustato atrastos viršūnės tėvinę viršūnę į u bei eina į ją su funkcija DFS-VISIT. 8-10 eilutės vyksta, kai viršūnė išsisemia, todėl nudažo ją juodai, padidina laiką vienetu ir priskiria jį viršūnės pabaigos laikui (u.f).

Sudėtingumo įvertinimas:

DFS funkcijoje ciklai eilutėse 1-3 ir 5-7 užima Θ(V), atmetant laiką, kurio reikia iškviesti DFS-VISIT. Funkcija DFS-VISIT kviečiama vieną kartą kiekvienai viršūnei v. DFS-VISIT funkcijoje ciklas eilutėse 4-7 veikia |Adj[v]| (Adj[v] – viršūnės v gretimų viršūnių masyvas) kartų. Kadangi $\sum_{v \in V} |Adj[v]| = \Theta(E)$, tai to ciklo laikas yra Θ(E). Galiausiai, pilnas paieškos į gylį laikas yra **Θ(V + E)**.

3. Kruskalo algoritmas ir sudėtingumo įvertinimas:

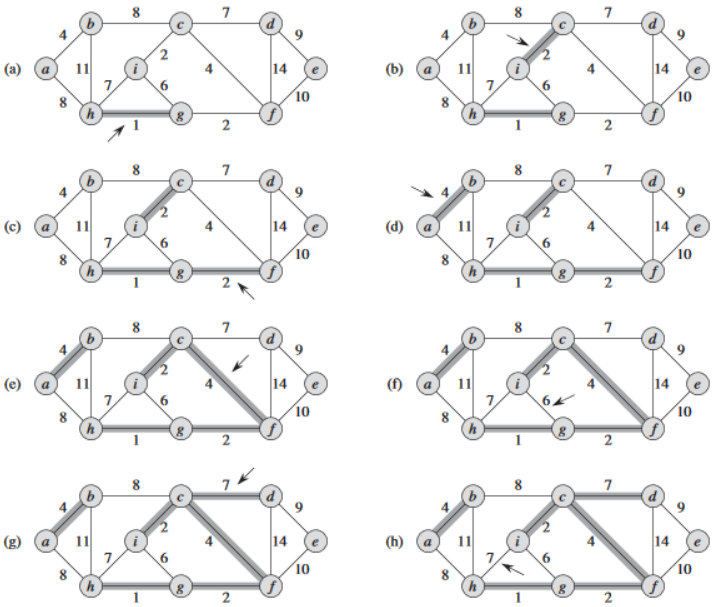
https://www.youtube.com/watch?v=71UQH7Pr9kU

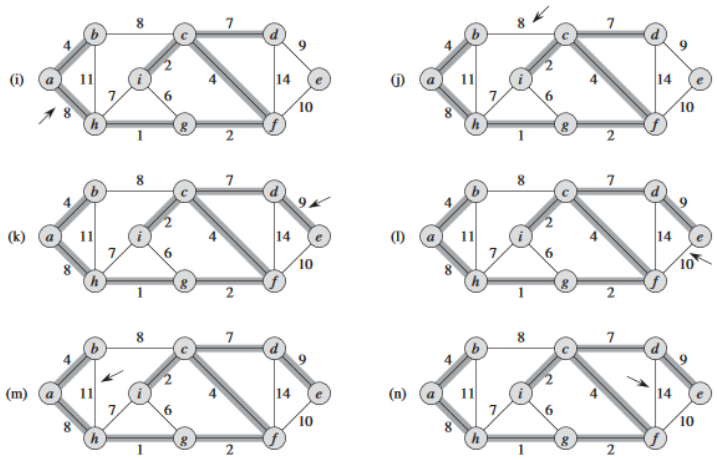
Tai yra trumpiausio dengiančio medžio paieškos algoritmas, kuris yra godus, nes kiekviename žingsnyje prideda trumpiausią įmanomą briauną. Grafo formulė: G = (V, E), kur V – viršūnės, o E – briaunos.

Iš pradžių susikuriame tiek aibių, kiek yra viršūnių, kad kiekvienoje aibėje būtų viena viršūnė, taip pat apsirašome rezultatų aibę A, kuri kol kas yra tuščia. Tada išrikiuojame briaunas pagal svorį. Einame pro kiekvieną briauną ir žiūrime, ar abu briaunos galai nepriklauso tai pačiai aibei. Jei nepriklauso – pridedame briauną į rezultatų aibę A ir sujungiame tikrinamas viršūnes.

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

Paaiškinimas: Eilutės 1-3 inicializuoja aibę A ir sukuria tiek atskirų medžių, kiek yra viršūnių. Kiekvienas medis kol kas turi po vieną viršūnę. Eilutėse 5-8 esantis ciklas eina pro briaunas pagal jų svorį didėjančia tvarka. Ciklas patikrina, ar u ir v viršūnės priklauso tam pačiam medžiui. Jeigu nepriklauso tam pačiam medžiui – 7 eilutė prideda briauną (u, v) į aibę A, o 8 eilutė sujungia viršūnes abiejuose medžiuose.





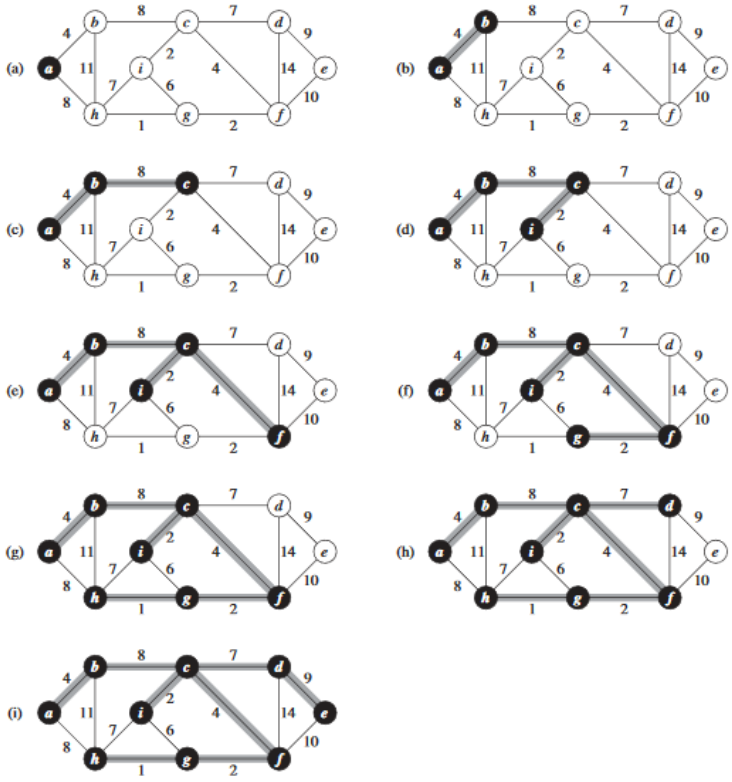
Sudėtingumo įvertinimas:

Sukurti tuščią aibę A užtrunka O(1). Surikiuoti briaunas pagal svorį užtrunka O(E*lg E), neskaičiuojant V kartų vykdomomis MAKE-SET operacijomis. For ciklas eilutėse 5-8 atlieka O(E) FIND-SET ir UNION operacijų. Kartu su V kartų vykdytomis MAKE-SET operacijomis, viskas užtrunka O((V + E) α(V)), kur α yra labai lėtai auganti funkcija. Kadangi manome, jog grafas yra jungusis, turime, kad |E| >= |V| - 1 (briaunų yra daugiau/lygu nei viršūnių – 1), taigi atskirimo operacijos užtrunka O(Eα(V)). Kadangi α(|V|) = O(lg V) = O(lg E), tai Kruskalo algoritmo laikas yra O(E*lg E). Žinodami, kad $|E| < |V|^2$, turime lg|E| = O(lg V), tai galime pasakyti, kad Kruskalo algoritmo užtrunkamas laikas yra **O(E lg V)**.

4. Prima algoritmas ir sudėtingumo įvertinimas:

https://www.youtube.com/watch?v=cplfcGZmX7I

Prima algoritmas yra naudojamas rasti trumpiausią dengiantį medį. Jis veikia labai panašiai į Dijkstros algoritmą. Šis algoritmas prasideda nuo vienos viršūnės-šaknies r ir auga tol, kol padengia visas likusias viršūnes. Tai yra godus algoritmas, nes kiekviename žingsnyje prie medžio prideda briauną, kuri yra mažiausia įmanoma. Pasirenkame višūnę-šaknį ir žiūrėkime, kokią viršūnę mūsų medis gali pasiekti per mažiausią įmanomą briaunos svorį. Kai randame viršūnę, iki kurios kelias yra mažiausias, pridedame ją prie medžio ir toliau ieškome mažiausios viršūnės tol, kol padengsime visas.




```
MST-PRIM(G, w, r)
 1  for each u ∈ G.V
 2      u.key = ∞
 3      u.π = NIL
 4  r.key = 0
 5  Q = G.V
 6  while Q ≠ ∅
 7      u = EXTRACT-MIN(Q)
 8      for each v ∈ G.Adj[u]
 9          if v ∈ Q and w(u, v) < v.key
10              v.π = u
11              v.key = w(u, v)
```

Paaiškinimas:

1-5 eilutės nustato kiekvienos viršūnės u.key į begalybę ir tėvinę viršūnę į NIL. Viršūnės-šaknies r.key yra lygus 0. Į eilę Q yra įdedamos visos esamos viršūnės. 7 eilutėje išrenkama mažiausią key atributą turinti viršūnė ir briauna tarp jos ir tėvinės viršūnės yra prijungiama prie medžio. 8-11 eilutės ciklas atnaujina visų gretimų viršūnių, kurios nėra mūsų medyje, key ir tėvinės viršūnės atributus, jei jų atstumas iki medžio yra mažesnis nei nustatytas key.

Ciklo invariantas:

- $A = \{(v, v.\pi) : v \in V - \{r\} - Q\}$.
- Viršūnės, kurios jau yra įdėtos į mažiausią dengiantį medį yra V – Q.
- Visoms viršūnėms v esančioms eilėje Q: jei tėvinė viršūnė tv nelygi NIL, tada key atributas bus mažiau nei begalybė ir jis bus svoris briaunos (v, tv).

Sudėtingumo įvertinimas:

Pradinių reikšmių nustatymas trunka O(V). Kiekviena EXTRACT-MIN operacija trunka O(lg V), taigi visų kvietimų į EXTRACT-MIN laikas yra O(V * lg V). For ciklas 8-11 eilutėje trunka O(E) kartų. 11 eilutė yra įgyvendinama per O(lg V). Taigi, bendra suma Prima algoritmo yra O(V * lg V + E * lg V) = **O(E lg V)**. Tai yra tas pats, kas Kruskalo algoritme.

5. Trumpiausi keliai iš vienos viršūnės:

Trumpiausio kelio radimo principas yra paprastas. Jei turimi du taškai ir reikia rasti trumpiausią kelią tarp jų, tai tiesiog galima sužymėti visus esamus kelius tarp jų, išmatuoti atstumus/svorius ir pasirinkti trumpiausią/lengviausią. Tačiau nesunku pastebėti, kad net ir išimant kelius, kurie turi ciklus, pasirinkimų yra be galo daug, taigi toks būdas yra neefektyvus.

Trumpiausio kelio algoritmai paprastai laikosi savybės, kad trumpiausias kelias tarp dviejų viršūnių turi dar trumpesnį subkelią tarp jų (subpaths of shortest paths are shortest paths).

Joks kelias nebus trumpiausias, jei savyje jis turi neigiamą ar teigiamą ciklą. Jei kelias nuo s iki v savyje turi neigiamą ciklą, tai jo svoris bus minus begalybė. Jei nuo viršūnės s negali tiesiogiai pasiekti viršūnes v (nėra jungiančių atkarpų), tai tokios kelio svoris bus plius begalybė, net jei ir kita kelio dalis savyje turės neigiamą ciklą.
Jei kelias nuo s iki v turi nulinio svorio ciklą, galima tą ciklą panaikinti, taip sukuriant kelią, kurio svoris yra toks pat. Kol kelias turi nulinio svorio ciklus, mes galime juos panaikinti tol, kol liks tik trumpiausias kelias, kuris neturi jokio ciklo.

Relaksacijos metodas:

Kiekvienai viršūnei v, kuri priklauso V turime atributą v.d, kuris yra trumpiausio kelio nuo s iki v aukštesnioji riba (lygus arba didesnis skaičius nei turima aibėje). v.d vadinsime trumpiausio kelio įverčiu. Inicializuojame trumpiausio kelio įverčius ir pirmtakus šia procedūra:

```
INITIALIZE-SINGLE-SOURCE(G, s)
1  for each vertex v ∈ G.V
2      v.d = ∞
3      v.π = NIL
4  s.d = 0
```

Sudėtingumas – Θ(V)
Briaunos relaksacijos procesas (u,v) susidaro iš testavimo, ar galima pagerinti trumpiausią kelią rastą iki v, einant per u, ir jei taip, tai atnaujinti v.d ir v.π. Rekalsacijos žingsnis gali sumažinti trumpiausio kelio įverčio v.d vertę, tačiau atnaujinti v pirmtako atributą v.π. Šitas kodas atlieka relaksacijos žingsnį ant (u,v) briaunos per O(1) laiką:

```
RELAX(u, v, w)
1  if v.d > u.d + w(u, v)
2      v.d = u.d + w(u, v)
3      v.π = u
```

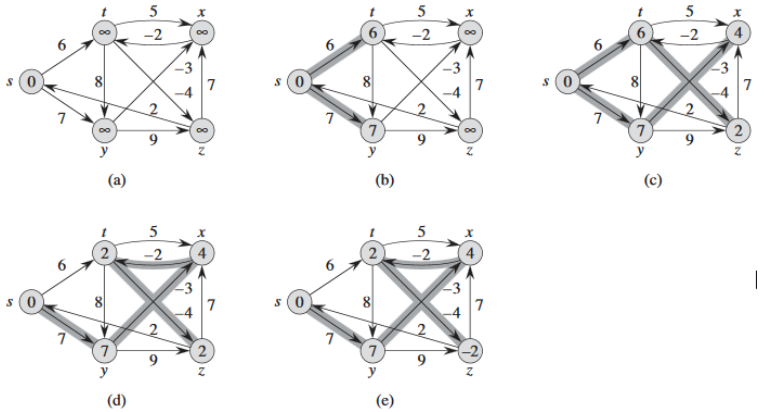
Relaksacija yra vienintelis būdas keisti trumpiausio kelio įverčius ir pirmtakus. Tarp algoritmų skiriasi tik tai, kiek kartų relaksacija yra panaudojama, pvz. Deikstros algoritmas ir trumpiausio kelio algoritmai tiesiniams acikliniams grafams relaksuoja kiekvieną briauną po vieną kartą. Bellman-Ford algoritmas relaksuoja kiekvieną briauną |V| -1 kartą.

Belmano – Fordo algoritmas:

https://www.youtube.com/watch?v=obWXjtg0L64

Belmano – Fordo algoritmas sprendžia single-source trumpiausio kelio problemą bendru atveju, kada briaunų svoriai gali būti neigiami. Duodant pasvertą grafą su kryptimi $G = (V, E)$, su šaltiniu s ir svorio funkcija $w : E \rightarrow \mathbb{R}$, Belmano-Fordo algoritmas gražina Boolean reikšmę, kuri indikuoja, ar yra neigiamo svorio ciklas, kuris yra pasiekiamas iš šaltinio s. Jei yra, algoritmas indikuoja, kad neegzistuoja joks sprendimas. Jei tokios reikšmės nėra (t.y nėra neigiamo ciklo), algoritmas gamina trumpiausius kelius ir jo svorius. Algoritmas relaksuoja briaunas palaipsniui mažinant įvertį v.d , esantį trumpiausiame kelyje nuo šaltinio s iki kiekvienos viršūnės, kol pagaliau pasiekiamas tikrasis trumpiausio kelio svoris.

```
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 grąžina TRUE tik tuo atveju, kai grafas neturi neigiamo svorio ciklų, kurie yra pasiekiami iš šaltinio s.

Sudėtingumo įvertinimas:

Belmano-Fordo algortimas veikia **O(VE)** sudėtingumu, kadangi inicializacija pirmoje eilutėje trunka Θ(V) laiko, kiekvienam |V|-1 praėjimui pro briaunas (2-4 eilutės) reikia Θ(E) laiko ir for ciklas 5-7 eilutėse reikalauja O(E) laiko.

6. Trumpiausi keliai iš vienos viršūnės: atsakyta 5 klausime.

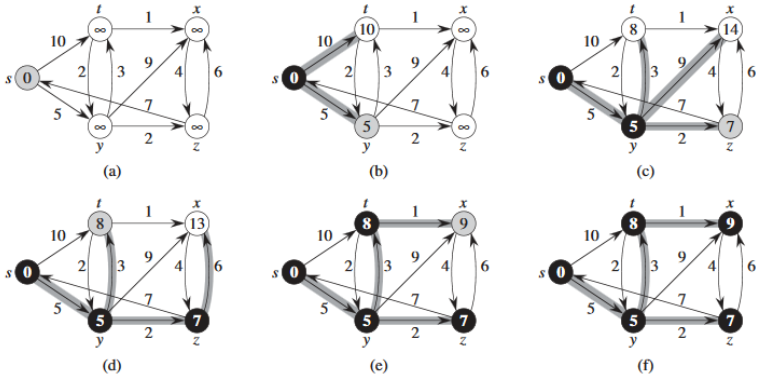
Relaksacijos metodas: atsakyta 5 klausime.

Deikstra algoritmas:

https://www.youtube.com/watch?v=_lHSawdgXpI

Deikstra algoritmas sprendžia single-source trumpiausio kelio problemą atvejui, kada briaunų svoriai yra neneigiami. Gera Deikstros algoritmo implementacija leidžia jai veikti greičiau nei už Belmano-Fordo algoritmą.

```
DIJKSTRA(G, w, s)
1  INITIALIZE-SINGLE-SOURCE(G, s)
2  S = ∅
3  Q = G.V
4  while Q ≠ ∅
5      u = EXTRACT-MIN(Q)
6      S = S ∪ {u}
7      for each vertex v ∈ G.Adj[u]
8          RELAX(u, v, w)
```



Deikstros algoritmas turi viršūnių aibę S, kurios galutiniai trumpiausių kelių svoriai yra nustatyti. Algoritmas pakartotinai atrenka viršūnes su mažiausiu trumpiausio kelio įverčiu, prie S prideda u ir relaksuoja visas briaunas, išeinančias iš u.

Duotas grafas ir pradinė viršūnė. Iš pradžių žinome atstumą nuo pradinės viršūnės iki jos pačios (nulis, per ją pačią). Atstumas iki kitų viršūnių nežinomas (todėl reikšmės yra begalybės, po to mažinamos). Kiekviename žingsnyje randame viršūnę, atstumas iki kurios jau nusistovėjo, įtraukiame tą viršūnė į aibę viršūnių, iki kurių atstumas jau nusistovėjo ir atnaujiname atstumo iki kitų viršūnių įverčius.

Sudėtingumo įvertinimas:
Blogiausiu atveju algortimas veiks **$O(V^2)$** (V- viršūnių kiekis, E – briaunų kiekis). Kiekviena INSERT ir DECREASE KEY operacija trunka O(1) ir kiekviena EXTRACT-MIN operacija trunka O(V), taigi bendra trukmė $O(V^2 + E) = \mathbf{O(V^2)}$.

Geriausiu atveju, kiekviena EXTRACT-MIN operacija trunka O(lgV) ir tokių operacijų yra |V|. Laikas skirtas sukurti binarine piramide (binary heap) yra O(V) ir tokių operacijų yra |E|. Kiekviena DECREASE-KEY operacija trunka O(lgV). Taigi sudėtingumas yra O((V+E) lgV) = O(E lg V)
Naudojant gretimumą, algoritmas gali veikti O(E log V), pasitelkiant binarinės piramidės pagalba.

7. Trumpiausių kelių paieška iš vienos viršūnės orientuotame acikliniame grafe ir sudėtingumo įvertinimas:

Trumpiausi keliai orientuotame cikliniame grafe yra puikiai apibrėžti, kadangi jei ten yra neigiamo svorio briaunų, negali egzistuoti neigiamo svorio ciklai. Algoritmas prasideda topologiškai išrikiuojant orientuotą ciklinį grafą (DAG) primetant linijinį išsidėstymą viršūnėse. Jei orientuotas ciklinis grafas turi kelią nuo viršūnės u iki v, tada u topologiniame rikiavime eina anksčiau už v. Kol dirbama su kiekviena viršūne, kiekviena briauna, kuri išeina iš tos viršūnės būna relaksuojama

```
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)
```

Topologinis rikiavimas (1 eil) užima Θ(V+E) laiko. INITIALIZE-SINGLE-SOURCE užima Θ(V). Ciklas nuo 3 iki 5 eilutės atlieka vieną iteraciją vienai viršūnei. For ciklas 4-5 eilutėse relaksuoja kiekvieną briauną lygiai po kartą. Kadangi vidinis ciklas trunka Θ(1) laiko, bendras sudėtingumas lieka **Θ(V+E)**.

Trečia klausimų grupė (4 balai):

8. Daugiagijo dinaminio programavimo metodika:

1. **Kas yra daugiagijis programavimas?**

   Tai programavimas skaidant uždavinį į kelis procesus, yra dvi pagrindinės tokio programavimo strategijos:

   - Kiekvienam procesoriui/branduolui atskira atmintis (distributed memory)
   - Visiems procesoriui/branduoliams bendra atmintis (shared memory)

   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 (šį darbą atlieka os dispečeris).

2. **Dinaminis daugiagijis programavimas**

   Dinaminis daugiagijis programavimas yra viena iš metodikų, kaip ir statinis daugiagijis programavimas. Dinaminis daugiagijis programavimas leidžia programuotuojui kurti algoritmus lygiagrečiai nesirūpinant dėl įvairių klaidų, kurios įvyksta programuojant statiškai, nes visą lygiagretumą tvarko įrenginio (mašinos) užduočių planuoklė. Pagrindiniai du dinaminio daugiagijo programavimo bruožai yra nested parallelism (kai skirtingos užduotys padalintos skirtingoms gijoms) ir lygiagretūs ciklai.

3. **Dinaminio daugiagijo programavimo modelio pranašumai**
   - Tai leidžia lygiai padalinti darbą pagal apimtį ir laiką reikalingą jam atlikti
   - Dauguma dinaminių daugiagijų algoritmų yra paremti skaldyk ir valdyk principu

4. **Papildomi operatoriai:**
   - Parallel – ciklo iteracijų vykdymas vienu metu.
   - Spawn – naujos gijos/proceso paleidimas.
   - Sync – gijų/procesų sinchronizavimas.

5. **Algoritmų vertinimo metrikos**
   - Darbas (work)
   - Intervalas (span)

   Pirmasis nusako kiek yra vienetinių užduočių, t. y. mazgų skaičius acikliniame grafe. Antrasis nusako koks didžiausias kelias acikliniame grafe.

6. **Žymėjimai**
   - P – procesorių skaičius
   - TP – laikas reikalingas užduotį atlikti su P procesorių.

   Tvirtinimai
   - Idealus kompiuteris su P procesorių per vieną žingsnį gali atlikti P darbo vienetų (mazgų, jei žiūrėsime į vykdymo grafą), kas leidžia apibrėžti darbo taisyklę: $TP \geq T1\ P$
   - Idealus kompiuteris su P procesorių negali greičiau dirbti (įvykdyti užduotį) nei kompiuteris su neribotu procesorių kiekiu, nes pirmąjį jis gali imituoti naudodamas tik P procesorių. Taigi $TP \geq T\infty$.

7. **Dispečeris**
   - Centralizuotas – kiekvienu laiko momentu žino globalią skaičiavimo resursų būseną.
   - Godus dispečeris – duotuoju laiko momentu vykdo visas turimas užduotis visais procesoriais.

   $TP \leq T1\ P/ + T\infty$

9. Daugiagijai matricų dauginimo algoritmai ir jų vykdymo laikų bei išlygiagretinimo koeficientų įvertinimas:

Pradžiai: https://www.youtube.com/watch?v=bOmAllndo-4

Daugiagijai matricų dauginimo algoritmai yra paremti skaldyk ir valdyk principu bei padalinant į tris skirtingas dalis (works)

- **Tiesioginis algoritmas (kvadratinių matricų dauginimo algoritmas)**

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

To analyze this algorithm, observe that since the serialization of the algorithm is just SQUARE-MATRIX-MULTIPLY, the work is therefore simply $T_1(n) = \Theta(n^3)$, the same as the running time of SQUARE-MATRIX-MULTIPLY. The span is $T_\infty(n) = \Theta(n)$, because it follows a path down the tree of recursion for the **parallel for** loop starting in line 3, then down the tree of recursion for the **parallel for** loop starting in line 4, and then executes all n iterations of the ordinary **for** loop starting in line 6, resulting in a total span of $\Theta(\lg n) + \Theta(\lg n) + \Theta(n) = \Theta(n)$. Thus, the parallelism is $\Theta(n^3)/\Theta(n) = \Theta(n^2)$. Exercise 27.2-3 asks you to parallelize the inner loop to obtain a parallelism of $\Theta(n^3/\lg n)$, which you cannot do straightforwardly using **parallel for**, because you would create races.

- **Skaldyk ir valdyk algoritmas**

```
P-MATRIX-MULTIPLY-RECURSIVE(C, A, B)
1  n = A.rows
2  if n == 1
3      c_11 = a_11 b_11
4  else let T be a new n × n matrix
5      partition A, B, C, and T into n/2 × n/2 submatrices
           A_11, A_12, A_21, A_22; B_11, B_12, B_21, B_22; C_11, C_12, C_21, C_22;
           and T_11, T_12, T_21, T_22; respectively
6      spawn P-MATRIX-MULTIPLY-RECURSIVE(C_11, A_11, B_11)
7      spawn P-MATRIX-MULTIPLY-RECURSIVE(C_12, A_11, B_12)
8      spawn P-MATRIX-MULTIPLY-RECURSIVE(C_21, A_21, B_11)
9      spawn P-MATRIX-MULTIPLY-RECURSIVE(C_22, A_21, B_12)
10     spawn P-MATRIX-MULTIPLY-RECURSIVE(T_11, A_12, B_21)
11     spawn P-MATRIX-MULTIPLY-RECURSIVE(T_12, A_12, B_22)
12     spawn P-MATRIX-MULTIPLY-RECURSIVE(T_21, A_22, B_21)
13     P-MATRIX-MULTIPLY-RECURSIVE(T_22, A_22, B_22)
14     sync
15     parallel for i = 1 to n
16         parallel for j = 1 to n
17             c_ij = c_ij + t_ij
```

The pseudocode implements this divide-and-conquer strategy using nested parallelism. Unlike the SQUAREMATRIX-MULTIPLY-RECURSIVE procedure on which it

is based, P-MATRIXMULTIPLY-RECURSIVE takes the output matrix as a parameter to avoid allocating matrices unnecessarily.

Paralelizmo ir laikų įvertinimas:

We first analyze the work $M_1(n)$ of the P-MATRIX-MULTIPLY-RECURSIVE procedure, echoing the serial running-time analysis of its progenitor SQUARE-MATRIX-MULTIPLY-RECURSIVE. In the recursive case, we partition in $\Theta(1)$ time, perform eight recursive multiplications of $n/2 \times n/2$ matrices, and finish up with the $\Theta(n^2)$ work from adding two $n \times n$ matrices. Thus, the recurrence for the work $M_1(n)$ is

$$\begin{aligned} M_1(n) &= 8M_1(n/2) + \Theta(n^2) \\ &= \Theta(n^3) \end{aligned}$$

by case 1 of the master theorem. In other words, the work of our multithreaded algorithm is asymptotically the same as the running time of the procedure SQUARE-MATRIX-MULTIPLY in Section 4.2, with its triply nested loops.

To determine the span $M_\infty(n)$ of P-MATRIX-MULTIPLY-RECURSIVE, we first observe that the span for partitioning is $\Theta(1)$, which is dominated by the $\Theta(\lg n)$ span of the doubly nested **parallel for** loops in lines 15–17. Because the eight parallel recursive calls all execute on matrices of the same size, the maximum span for any recursive call is just the span of any one. Hence, the recurrence for the span $M_\infty(n)$ of P-MATRIX-MULTIPLY-RECURSIVE is

$$M_\infty(n) = M_\infty(n/2) + \Theta(\lg n) \; . \tag{27.7}$$

This recurrence does not fall under any of the cases of the master theorem, but it does meet the condition of Exercise 4.6-2. By Exercise 4.6-2, therefore, the solution to recurrence (27.7) is $M_\infty(n) = \Theta(\lg^2 n)$.

Now that we know the work and span of P-MATRIX-MULTIPLY-RECURSIVE, we can compute its parallelism as $M_1(n)/M_\infty(n) = \Theta(n^3/\lg^2 n)$, which is very high.

- **Strassen's algorithm (jei gerai atsimenu tai čia Štrauzo algoritmas būtų)**

1. Divide the input matrices A and B and output matrix C into $n/2 \times n/2$ submatrices, as in equation (27.6). This step takes $\Theta(1)$ work and span by index calculation.
2. Create 10 matrices $S_1, S_2, \ldots, S_{10}$, each of which is $n/2 \times n/2$ and is the sum or difference of two matrices created in step 1. We can create all 10 matrices with $\Theta(n^2)$ work and $\Theta(\lg n)$ span by using doubly nested **parallel for** loops.
3. Using the submatrices created in step 1 and the 10 matrices created in step 2, recursively spawn the computation of seven $n/2 \times n/2$ matrix products $P_1, P_2, \ldots, P_7$.
4. Compute the desired submatrices $C_{11}, C_{12}, C_{21}, C_{22}$ of the result matrix C by adding and subtracting various combinations of the P_i matrices, once again using doubly nested **parallel for** loops. We can compute all four submatrices with $\Theta(n^2)$ work and $\Theta(\lg n)$ span.

Paralelizmo ir laikų įvertinimas:

To analyze this algorithm, we first observe that since the serialization is the same as the original serial algorithm, the work is just the running time of the serialization, namely, $\Theta(n^{\lg 7})$. As for P-MATRIX-MULTIPLY-RECURSIVE, we can devise a recurrence for the span. In this case, seven recursive calls execute in parallel, but since they all operate on matrices of the same size, we obtain the same recurrence (27.7) as we did for P-MATRIX-MULTIPLY-RECURSIVE, which has solution $\Theta(\lg^2 n)$. Thus, the parallelism of multithreaded Strassen's method is $\Theta(n^{\lg 7}/\lg^2 n)$, which is high, though slightly less than the parallelism of P-MATRIX-MULTIPLY-RECURSIVE.

10. Daugiagijai rikiavimo algoritmai ir jų vykdymo laikų bei lygiagretinimo koeficientų įvertinimas **(čia kinda bs, nes tiek knygoje, tiek skaidrėse tik apie merge sort rašo, bet here we go):**

Geriausias konspektas:
https://www.youtube.com/watch?v=_XOZ2IiP2nw
https://www.youtube.com/watch?v=GvtgV2NkdVg
https://www.youtube.com/watch?v=RCFxlXkP2fs

- Pseudocode:

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

```
BINARY-SEARCH(x, T, p, r)
1  low = p
2  high = max(p, r + 1)
3  while low < high
4      mid = ⌊(low + high)/2⌋
5      if x ≤ T[mid]
6          high = mid
7      else low = mid + 1
8  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
```

- **Pagrindinė idėja:**

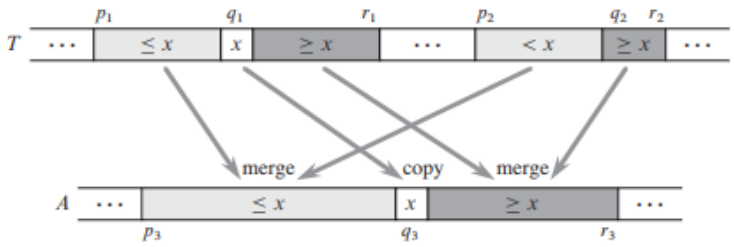


Figure 27.6 The idea behind the multithreaded merging of two sorted subarrays $T[p_1 \,..\, r_1]$ and $T[p_2 \,..\, r_2]$ into the subarray $A[p_3 \,..\, r_3]$. Letting $x = T[q_1]$ be the median of $T[p_1 \,..\, r_1]$ and q_2 be the place in $T[p_2 \,..\, r_2]$ such that x would fall between $T[q_2 - 1]$ and $T[q_2]$, every element in subarrays $T[p_1 \,..\, q_1 - 1]$ and $T[p_2 \,..\, q_2 - 1]$ (lightly shaded) is less than or equal to x, and every element in the subarrays $T[q_1 + 1 \,..\, r_1]$ and $T[q_2 + 1 \,..\, r_2]$ (heavily shaded) is at least x. To merge, we compute the index q_3 where x belongs in $A[p_3 \,..\, r_3]$, copy x into $A[q_3]$, and then recursively merge $T[p_1 \,..\, q_1 - 1]$ with $T[p_2 \,..\, q_2 - 1]$ into $A[p_3 \,..\, q_3 - 1]$ and $T[q_1 + 1 \,..\, r_1]$ with $T[q_2 \,..\, r_2]$ into $A[q_3 + 1 \,..\, r_3]$.

- **Vykdymo laikų ir lygiagretinimo koeficientų įvertinimas (kiekvienai funkcijai atskirai)**

Merge-Sort

which is the same as the serial running time of merge sort. Since the two recursive calls of MERGE-SORT′ can run in parallel, the span MS'_∞ is given by the recurrence

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

Thus, the parallelism of MERGE-SORT′ comes to $MS'_1(n)/MS'_\infty(n) = \Theta(\lg n)$, which is an unimpressive amount of parallelism. To sort 10 million elements, for example, it might achieve linear speedup on a few processors, but it would not scale up effectively to hundreds of processors.

Binary-Search

The call BINARY-SEARCH(x, T, p, r) takes $\Theta(\lg n)$ serial time in the worst case, where $n = r - p + 1$ is the size of the subarray on which it runs. (See Exercise 2.3-5.) Since BINARY-SEARCH is a serial procedure, its worst-case work and span are both $\Theta(\lg n)$.

P-Merge

We start by analyzing the work $PMS_1(n)$ of P-MERGE-SORT, which is considerably easier than analyzing the work of P-MERGE. Indeed, the work is given by the recurrence

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

This recurrence is the same as the recurrence (4.4) for ordinary MERGE-SORT from Section 2.3.1 and has solution $PMS_1(n) = \Theta(n \lg n)$ by case 2 of the master theorem.

We now derive and analyze a recurrence for the worst-case span $PMS_\infty(n)$. Because the two recursive calls to P-MERGE-SORT on lines 7 and 8 operate logically in parallel, we can ignore one of them, obtaining the recurrence

$$
\begin{aligned}
PMS_\infty(n) &= PMS_\infty(n/2) + PM_\infty(n) \\
&= PMS_\infty(n/2) + \Theta(\lg^2 n) \,. \qquad (27.10)
\end{aligned}
$$

As for recurrence (27.8), the master theorem does not apply to recurrence (27.10), but Exercise 4.6-2 does. The solution is $PMS_\infty(n) = \Theta(\lg^3 n)$, and so the span of P-MERGE-SORT is $\Theta(\lg^3 n)$.

Parallel merging gives P-MERGE-SORT a significant parallelism advantage over MERGE-SORT′. Recall that the parallelism of MERGE-SORT′, which calls the serial MERGE procedure, is only $\Theta(\lg n)$. For P-MERGE-SORT, the parallelism is

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

11. Amortizacinė algoritmų analizė:

Amortizacinėje analizėje mes daliname laiką reikalingą įvykdyti seką duomenų struktūros operacijų iš visų įvykdytų operacijų. Amortizacinė analizė garantuoja vidutinį kiekvienos operacijos našumą blogiausiu atveju.

$\sum_{i=1}^{n} \hat{c}_i \geq \sum_{i=1}^{n} c_i$ — n amortizacinių kainų suma, negali būti mažesnė negu n operacijų kainų suma

Yra 3 šios analizės tipai:

- Agregatinė
- Apskaitos
- Potencialo

Agregatinė analizė: surandama vidutinė operacijos kaina, kuri tada yra naudojama kaip kiekvienos operacijos kaina ir apskaičiuojamas sudėtingumas.

PVZ:

Tarkim turime tuščią stack`ą kuriam atliksim *n* Push, Pop ir Multipop operacijų. Blogiausiu atveju Multipop = O(*n*), nes didžiausias stack`o dydis yra *n*. Taigi blogiausias betkurios operacijos atvejis yra O(*n*) ir *n* tokių operacijų kainuos O(n^2). Šis įvertis yra teisingas, bet netikslus.

Naudojant agregatinę analizę galime surasti geresnį viršutinį rėžį. Nors ir Multipop operacija gali būti „brangi“, betkokia seka *n* Push, Pop ir Multipop operacijų su stack`e, kuris iš pradžių buvo tuščias, gali daugiausiai kainuoti O(*n*), todėl kad Pop ir Multipop operacijas galima kviesti tik tiek kartų kiek buvo kviesta Push operacija, o tai yra *n*. Padalinus iš sekos ilgio vidutinė operacijos kaina yra O(*n*)/*n* = O(1).

Apskaitos metodas: priskiriamos skirtingos kainos skirtingom operacijom (gali būt didesnės arba mažesnės nei iš tikro) – šias kainas vadiname **amortizuojančiomis kainomis.** Kai amortizuojanti kaina yra didesnė negu tikra kaina, skirtumą priskiriame „**kreditui“** – kurį vėliau galime kompensuoti operacijomis, kurių amortizuojanti kaina yra mažesnė negu tikra kaina.

Turi būti tenkinama sąlyga (t.y. kreditas turi būti teigiamas):

$$\sum_{i=1}^{n} \hat{c}_i \geq \sum_{i=1}^{n} c_i$$

PVZ:

Grįžkime prie buvusio pavyzdžio ir prisiminkime operacijų kainas:

Push	1,
Pop	1,
Multipop	min(k,s).

Priskirkime tokias amortizuotas kainas:

Push	2
Pop	0
Multipop	0

Kaip ir prieš tai, Multipop operacija neįvyks (jeigu pradinis stack`as tuščias), todėl jos kaina 0. Pop kainą pridedame prie Push kainos kaip kreditą, kadangi visi Push operacijos elementai bus išimti operacijos Pop. Panaudojus Push operaciją mes turėsime dar 1 kreditą, kurį galėsim panaudoti Pop operacijai sumokėti. Taip bus kiekvienos iteracijos metu ir tai įrodo, kad kreditas niekada nebus neigiamas. Iš to kyla išvada, kad visų *n* sekos operacijų Push, Pop ir Multipop kaina yra O(*n*).

Naudojant dvejetainį skaitliuką bito pavertimo 1 operacijos kaina būtų 2, kadangi bitas turės būti paverčiamas 0.

Potencialų metodas: vietoje kreditų naudojama **potencines energijos** arba **potencialo** sąvoka. Atliekam *n* iteracijų pradedant su pradine duomenų struktūra D_0. c_i bus tikroji *i*-osios operacijos kaina. **Potencinė funkcija** Φ priskiria kiekvieną duomenų struktūrą D_i realiam skaičiui, kuris yra tos D_i potencialas. **Amortizuota kaina** *i*-ajai operacijai:

$$\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) .$$

Amortizuota kaina lygi jos realios kainos ir potencialų skirtumo sumai. Išskleidus formulę gaunam:

$$\begin{aligned} \sum_{i=1}^{n} \hat{c}_i &= \sum_{i=1}^{n} (c_i + \Phi(D_i) - \Phi(D_{i-1})) \\ &= \sum_{i=1}^{n} c_i + \Phi(D_n) - \Phi(D_0) . \end{aligned}$$

PVZ:

Grįžkime prie uždavinio su stack. Aprašykime potencinę funkciją Φ kaip objektų skaičių stack`e. Pradinis stack`as D_0 turi 0 elementų savyje, todėl $\Phi(D_0) = 0$. Objektų skaičius stack`e niekada nebus mažesnis už 0, todėl stack`as D_i, kuris aprašomas po *i*-osios operacijos, visada turės neneigiamą potencialą.

Potencialų skirtumas Push operacijai, kai *s* – objektų skaičius stack`e:

$$\begin{aligned}\Phi(D_i) - \Phi(D_{i-1}) &= (s+1) - s \\ &= 1 .\end{aligned}$$

Įstačius šį skirtumą į formulę gauname Push operacijos amortizuotą kainą:

$$\begin{aligned}\hat{c}_i &= c_i + \Phi(D_i) - \Phi(D_{i-1}) \\ &= 1 + 1 \\ &= 2 .\end{aligned}$$

Multipop(*S,k*) operacija panaikina *k* objektų iš stack`o. Tikroji Multipop kaina yra *k`*, o potencialų skirtumas Multipop operacijai:

$$\Phi(D_i) - \Phi(D_{i-1}) = -k' .$$

Tada amortizuota Multipop kaina:

$$\begin{aligned}\hat{c}_i &= c_i + \Phi(D_i) - \Phi(D_{i-1}) \\ &= k' - k' \\ &= 0 .\end{aligned}$$

Pop funkcijos amortizuota kaina taip pat 0.

Visų trijų operacijų amortizuotos kainos yra O(*1*) ir todėl galutinė amortizuota kaina viršutiniam rėžiui *n* operacijų sekai yra O(*n*) .

12. NP Sudėtingumas. Ir P ir NP klasės. Uždavinių pavyzdžiai.

https://www.youtube.com/watch?v=YX40hbAHx3s&t

P klasės uždaviniai gali būti išspręsti per polinominį laiką. Tiksliau, visi P klasės uždaviniai gali būti išspręsti per $O(n^k)$ laiką, kur k – kokia nors konstanta, o n – įvedamų duomenų kiekis.

NP klasės uždaviniai gali būti patikrinami per polinominį laiką. Čia turima omenyje, kad jei mums kas nors duotų uždavinio atsakymą, mes galėtume jį patikrinti per polinominį laiką, tačiau nebūtinai išspręsti per jį. Pavyzdžiai:

- Gavę Hamiltono ciklo seką, mes lengvai per polinominį laiką galėtume patikrinti, ar tokia seka yra teisinga.
- Gavę sudoku atsakymą, per polinominį laiką galime patikrinti, ar jis teisingas, tačiau programai išspręsti didelį ir sudėtingą sudoku per polinominį laiką yra neįmanoma.

P uždaviniai visada priklauso NP klasės uždaviniams, bet ne atvirkščiai. $\mathbf{P} \subseteq \mathbf{NP}$.

Yra dar viena klasė uždavinių - NP-Complete (NPC). Šiai grupei priklauso uždaviniai, kurie yra NP klasėje ir yra tokie pat sunkūs kaip bet kuris NP uždavinys. Jei turime du uždavinius: A ir B (kuris yra NP uždavinys), bei A uždavinį galime suprastinti iki B uždavinio per polinominį laiką, tai reiškia, kad uždavinys B yra bent toks pat sunkus, koks yra A. Tai yra NP-Hard uždavinys. NPC grupei priklauso uždaviniai, kurie yra NP-Hard ir NP vienu metu.

NPC uždaviniai būna “decision problems”, kas reiškia, kad atsakymas bus taip arba ne (1 arba 0).

P Uždavinių pavyzdžiai:

- Rikiavimo, skaičiavimo algoritmai.

NPC Uždavinių pavyzdžiai:

- Hamiltono kelias, Keliaujančio pirklio uždavinys, grafų dažymas.