1. Dinaminis programavimas. Algoritmų sudarymo metodika ir šios metodikos taikymas sprendžiant Konvejerio (Surinkimo linijos planavimo) uždavinį (rekursinės lygtys, optimalaus sprendinio reikšmės ir struktūros radimas, algoritmo sudėtingumo radimas...).

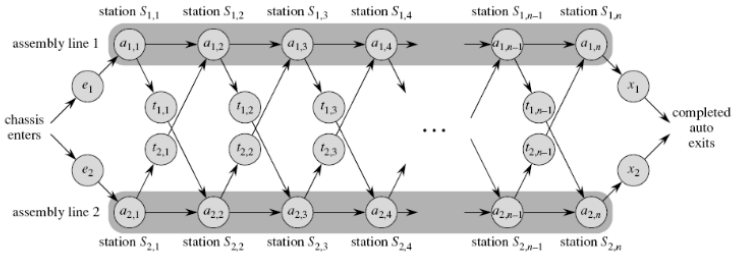
Dinaminis programavimas – rekursinių algoritmų taikymo metodika išsaugant dalinius rezultatus. Taip išvengiama pakartotinio to paties uždavinio sprendimo. Jei tas pats mažesnis uždavinys sprendžiamas daug kartų, mažesnio uždavinio sprendimas pakeičiamas jau išspręsto uždavinio rezultato paieška lentelėje. Taikant dinaminį programavimą paprastai reikia daugiau atminties rezultatų išsaugojimui, tačiau galutinis rezultatas gaunamas žymiai greičiau nei taikant įprastą rekursiją. Taikant dinaminį programavimą uždavinys paprastai sumažinamas pašalinant vieną elementą, išsprendžiamas mažesnis uždavinys ir jo sprendinys panaudojamas didesnio uždavinio sprendiniui gauti.

Uždavinio sprendimas dinaminiu programavimu susideda iš 4 žingsnių

1. Nusakyti optimalią uždavinio sprendinio struktūrą
2. Rekursiškai apibrėžti optimalų uždavinio sprendinį
3. Apskaičiuoti optimalaus sprendinio reikšmę
4. Rasti galutinį sprendinį

Konvejeris

Uždavinio formulavimas



Yra 2 gamybos linijos, kiekviena iš linijų turi n stočių. Pažymime j = 1, 2, ... n. Reikia rasti greičiausią kelią.

$$f_1[1] = e_1 + a_{1,1},$$
$$f_2[1] = e_2 + a_{2,1}.$$

$$f_1[j] = \min(f_1[j-1] + a_{1,j},\ f_2[j-1] + t_{2,j-1} + a_{1,j})$$

$$f_2[j] = \min(f_2[j-1] + a_{2,j},\ f_1[j-1] + t_{1,j-1} + a_{2,j})$$

kai

$$j = 2, 3, \ldots, n.$$

Apibendrinus

$$f_1[j] = \begin{cases} e_1 + a_{1,1} & \text{if } j = 1 \\ \min(f_1[j-1] + a_{1,j},\ f_2[j-1] + t_{2,j-1} + a_{1,j}) & \text{if } j \geq 2 \end{cases}$$

$$f_2[j] = \begin{cases} e_2 + a_{2,1} & \text{if } j = 1 \\ \min(f_2[j-1] + a_{2,j},\ f_1[j-1] + t_{1,j-1} + a_{2,j}) & \text{if } j \geq 2 \end{cases}$$

Sprendinys:

$$f^* = \min(f_1[n] + x_1,\ f_2[n] + x_2)$$

Eksponentinis !

Pažymėkime $r_i(j)$ kreipinių skaičių į funkciją $f_i(j)$.

$$r_1(n) = r_2(n) = 1$$

$$r_1(j) = r_2(j) = r_1(j+1) + r_2(j+1)$$

$$r_i(j) = 2^{n-j}$$

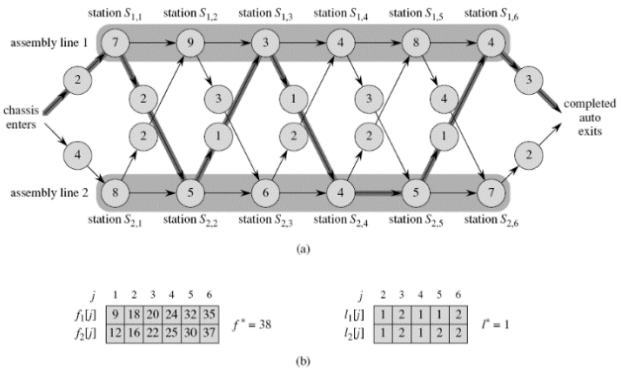
$f_1[1]$ bus kreiptasi 2^{n-1} kartų.

Norint sumažinti sudėtingumą, realizuokime naudodamiesi tarpinių rezultatų lentele/masyvu. Masyvas l bus naudojamas optimalaus kelio atspausdinimui.

---->

```
FASTEST-WAY(a, t, e, x, n)
 1  f1[1] ← e1 + a1,1
 2  f2[1] ← e2 + a2,1
 3  for j ← 2 to n
 4      do if f1[j − 1] + a1,j ≤ f2[j − 1] + t2,j−1 + a1,j
 5            then f1[j] ← f1[j − 1] + a1,j
 6                 l1[j] ← 1
 7            else f1[j] ← f2[j − 1] + t2,j−1 + a1,j
 8                 l1[j] ← 2
 9         if f2[j − 1] + a2,j ≤ f1[j − 1] + t1,j−1 + a2,j
10            then f2[j] ← f2[j − 1] + a2,j
11                 l2[j] ← 2
12            else f2[j] ← f1[j − 1] + t1,j−1 + a2,j
13                 l2[j] ← 1
14  if f1[n] + x1 ≤ f2[n] + x2
15     then f* = f1[n] + x1
16          l* = 1
17     else f* = f2[n] + x2
18          l* = 2
```

Sudėtingumas T(n)=Θ(n)



j	1	2	3	4	5	6	
$f_1[j]$	9	18	20	24	32	35	$f^* = 38$
$f_2[j]$	12	16	22	25	30	37	

j	2	3	4	5	6	
$l_1[j]$	1	2	1	1	2	$l^* = 1$
$l_2[j]$	1	2	1	2	2	

(b)

```
PRINT-STATIONS(l, n)
1  i ← l*
2  print "line " i ", station " n
3  for j ← n downto 2
4      do i ← l_i[j]
5         print "line " i ", station " j − 1
```

line 1, station 6
line 2, station 5
line 2, station 4
line 1, station 3
line 2, station 2
line 1, station 1

---->

2. Dinaminis programavimas. Algoritmų sudarymo metodika ir šios metodikos taikymas sprendžiant strypų pjaustymo uždavinį (rekursinės lygtys, optimalaus sprendinio reikšmės ir struktūros radimas, algoritmo sudėtingumo radimas…).

-..-

Turima n ilgio strypas ir kainų lentelė, kurioje nurodoma, *i* ilgio strypas kainuoja p_i, *i* = 1,2… Reikia sudaryti algoritmą maksimaliam pelnui gauti.

Tarkime, kad optimalus supjaustymas yra iš k gabalų. j-tojo gabalo ilgis - i_j , tada $n = i_1 + i_2 + \dots + i_k$. Šis padalinimas duoda maksimalų pelną, kurio dydis lygus $r_n = p_{i1} + p_{i2} + \dots + p_{ik}$. Tokių skirtingų sumų galima sudaryti 2^{n-1} variantų, nes galima nepriklausomai rinktis: atpjauti i ilgio gabalą ar nepjauti.

Pagrindinis faktas leidžiantis sudaryti uždavinio sprendimo algoritmą: atpjovus tam tikro ilgio gabalą, jo kaina ir likusiojo gabalo maksimalaus pelno suma turi būti maksimali. Taigi:

$$r_n = \max(p_n, r_1 + r_{n-1}, r_2 + r_{n-2}, \dots, r_{n-1} + r\_1\ )$$

Šią lygtį galima užrašyti paprasčiau laikant, kad atpjauname jau nedalinamą gabalą

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

---->

```
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
```

Sudėtingumas randams iš formulės

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

Pritaikius matematin3 indukciją nesunku parodyti, kad algoritmo sudėtingumas $T(n) = 2^n$.

Dinaminio programavimo taikymas:

Pirmo varianto taikymas

```
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
```

---->

antro varianto taikymas:

```
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]
```

Abiem atvejais sudėtingumas $T(n) = \Theta(n^2)$.

Papildoma duomenų struktūra

Abi procedūros randa maksimalų pelną. Norint rasti ir patį padalinimo variantą, algoritmus reikia papildyti struktūra, kurioje bus saugoma reikšmė, kokio ilgio gabalas buvo atpjautas.
BOTTOM-UP-CUT-ROD procedūros 6 eilutėje reikia įsiminti kurio ilgio strypas duoda maksimumą. Mūsų atveju bus masyvas s.

```
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
```

Optimalaus sprendinio radimas

Remiantis s masyvu randamas optimalus strypo padalinimas

```
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]
```

---->

3. Dinaminis programavimas. Algoritmų sudarymo metodika ir šios metodikos taikymas sprendžiant Matricų sekos optimalaus dauginimo uždavinį (rekursinės lygtys optimalaus sprendinio reikšmės ir struktūros radimas, algoritmo sudėtingumo radimas…).

Duota matricų seka. Jas reikia sudauginti taip, kad gautųsi kuo mažiau sandaugų.

Matricų daugybos tvarka

$A_1 A_2 \cdots A_n$.

Dauginimo variantai

$(A_1(A_2(A_3A_4)))$,
$(A_1((A_2A_3)A_4))$,
$((A_1A_2)(A_3A_4))$,
$((A_1(A_2A_3))A_4)$,
$(((A_1A_2)A_3)A_4)$.

$$P(n) = \begin{cases} 1 & \text{if } n = 1 , \\ \sum_{k=1}^{n-1} P(k)P(n-k) & \text{if } n \geq 2 . \end{cases}$$

Sprendinys

Kiek yra skirtingų variantų? ----> $\Omega(2^n)$.

```
MATRIX-MULTIPLY(A, B)
1  if columns[A] ≠ rows[B]
2     then error "incompatible dimensions"
3     else for i ← 1 to rows[A]
4               do for j ← 1 to columns[B]
5                     do C[i, j] ← 0
6                        for k ← 1 to columns[A]
7                            do C[i, j] ← C[i, j] + A[i, k] · B[k, j]
8          return C
```

$$m[i, j] = m[i, k] + m[k + 1, j] + p_{i-1} p_k p_j$$

$$m[i, j] = \begin{cases} 0 & \text{if } i = j , \\ \min_{i \leq k < i} \{m[i, k] + m[k + 1, j] + p_{i-1} p_k p_j\} & \text{if } i < j . \end{cases}$$

---->

```
MATRIX-CHAIN-ORDER(p)
1  n ← length[p] - 1
2  for i ← 1 to n
3      do m[i, i] ← 0
4  for l ← 2 to n          ▷ l is the chain length.
5      do for i ← 1 to n - l + 1
6             do j ← i + l - 1
7                m[i, j] ← ∞
8                for k ← i to j - 1
9                    do q ← m[i, k] + m[k + 1, j] + p_{i-1} p_k p_j
10                      if q < m[i, j]
11                        then m[i, j] ← q
12                             s[i, j] ← k
13 return m and s
```

trys for ciklai, todėl sudėtingumas $\Theta(n^3)$

Optimalus sprendinys

```
PRINT-OPTIMAL-PARENS(s, i, j)
1  if i = j
2     then print "A"_i
3     else print "("
4          PRINT-OPTIMAL-PARENS(s, i, s[i, j])
5          PRINT-OPTIMAL-PARENS(s, s[i, j] + 1, j)
6          print ")"
```

---->

Strypo padalinimo atveju $\Theta(n)$ pagalbinių uždavinių ir nedaugiau kaip n variantų, todėl sudėtingumas $\Theta(n^2)$.

Konvejerio atveju pagalbinių uždavinių skaičius priklauso nuo darbo vietų skaičiaus, t. y. $\Theta(n)$, o pasirinkimo variantai tik 2, todėl sudėtingumas $\Theta(n)$.

Matricų daugybos atveju sprendžiamų pagalbinių uždavinių yra $\Theta(n^2)$, o pasirinkimo variantų nedaugiau $n-1$, todėl sudėtingumas $\Theta(n^3)$.

Pagalbiniai uždaviniai yra nepriklausomi.

4. Godūs algoritmai. Maksimalios procesų aibės radimo uždavinys ir jo sprendimas (rekursinės lygtys, optimalaus sprendinio reikšmės ir struktūros radimas, algoritmo sudėtingumo radimas…).

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, tik negarantuoja optimalaus sprendinio radimo.

$S = \{a_1, a_2, \dots, a_n\}$ – procesų (užduočių) aibė

s_i ir f_i – i-tojo proceso pradžios ir pabaigos laikas, $0 \le s_i < f_i < \infty$

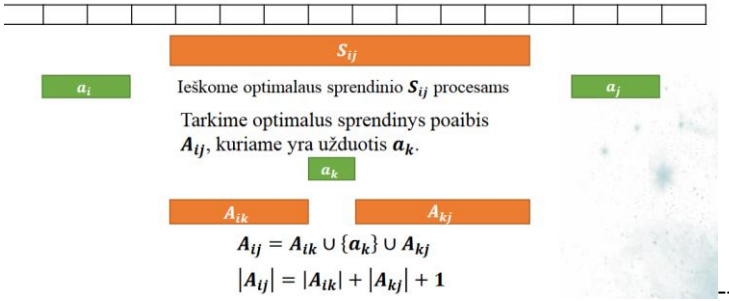
Procesai nesikerta jei intervalai $[s_i, f_i)$ ir $(s_j, f_j]$ nepersidengia.

Procesai pateikti surikiuoti didėjimo tvarka pagal pabaigos laiką: $f_1 \le f_2 \le \dots \le f_n$

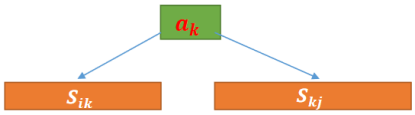
	i	1	2	3	4	5	6	7	8	9	10	11
Duomenų pvz.:	s_i	1	3	0	5	3	5	6	8	8	2	12
	f_i	4	5	6	7	9	9	10	11	12	14	16

Optimali struktūra

Tarkime S_{ij} - procesų poaibis, kuris prasideda baigusis procesui a_i ir baigiasi dar neprasidėjus procesui a_j



---->



rekursinis / dinaminio programavimo sprendimas

Iš optimalios sprendinio struktūros seka:

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

Tačiau k nežinome ir reikia patikrinti visus galimus variantus

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

Čia sudėtingumas eksponentinis. Norint padaryti godaus algoritmo realizaciją, reikia priimti optimalų sprendimą nesprendžiant mažesnių uždavinių. Pasirenkamas procesas, kuris trunka mažiausiai, kad liktų kuo daugiau laiko kitiems procesams. Turint godų sprendimą, mums reikia spręsti tik vieną uždavinį – rasti procesą, kuris prasideda, kai baigiasi ankstesnis.

Rekursinė godaus algoritmo realizacija

```
RECURSIVE-ACTIVITY-SELECTOR(s, f, i, n)
1  m ← i + 1
2  while m ≤ n and s_m < f_i        ▷ Find the first activity in S_{i,n+1}.
3      do m ← m + 1
4  if m ≤ n
5     then return {a_m} ∪ RECURSIVE-ACTIVITY-SELECTOR(s, f, m, n)
6     else return ∅
```

s – užduočių pradžios laikai
f – užduočių pabaigos laikai
n – viso užduočių
i – paskutinė įtraukta užduotis

Sudėtingumas Θ(n), o jei nesurikiuoti įėjimo duomenys pagal laiką, kur pirmas trumpiausias, tai papildomai Θ(nlog2n). Iteracinio godaus algoritmo realizavime Θ(n).

5. Duomenų struktūros aprašančios grafą ir jų privalumai bei trūkumai. Paieškos į plotį algoritmas ir sudėtingumo įvertinimas.

Grafą G = (V, E) sudaro dvi aibės: V viršūnių aibė ir E lankų (Briaunų) aibė. Algoritmuose naudojamos dvi struktūros:

1. Grafas išreiškiamas viršūnių kaimynų sąrašais (arba tiesiog sąrašais). Šį būdą tikslinga naudoti, kai grafas yra retas, t. y. $|V| << |E^2|$. Kiekvienai viršūnei priskiriamas viršūnių sąrašas, kuris apima visas briaunas. Orientuoto grafo G atveju visų sąrašų ilgis lygus |E|, neorientuoto – 2|E|. Abiem atvejais reikia Θ(V+E) atminties. Pagrindinis trūkumas – nebuvimas greito būdo nustatyti ar egzistuoja grafe (u,v) briauna.
2. Grafo viršūnių kaimynai sužymimi matricoje. Šis būdas pasiteisina kai |V| yra artimas $|E^2|$. Jei egzistuoja briauna, rašomas 1, jei ne – 0. Kai neorientuotas grafas pateikiamas A matrica, galioja lygybė $A=A^T$. Šiuo atveju pakanka saugoti reikšmes esančias pagrindinėje įstrižainėje ir virš jos, sumažinant beveik dvigubai naudojamos atminties.

Paieška į plotį: duotas G = (V,E) grafas ir pradinė viršūnė s Є V. Tikslas ateiti į visas viršūnės pasiekiamas iš s. Be to reikia rasti minimalų atstumą nuo s iki kitų pasiekiamų viršūnių. Pirma nagrinėjamos viršūnės nutolusios k atstumu, tik vėliau k+1 atstumu.

Kiekvienai viršūnei u priskirsime spalvos atributą $color[u]$, kuris gali įgyti tris reikšmes:

- balta – nenagrinėta,
- pilka – pažymėta, kaip kaimyninė viršūnė nagrinėtos viršūnės, bet gali turėti kaimynystėje ir baltų viršūnių,
- juoda – išnagrinėta viršūnė.

Paieška į plotį algoritmas sudaro paieškos į plotį medį, kuris pradiniu laiko momentu turi tik šakninę viršūnę s, be to kiekvienai viršūnei u priskiriamas atributas $\pi[u]$, nurodantis kokia viršūnė eina prieš ją ir atributas $d[u]$ nurodantis kokiu atstumu yra nutolusi viršūnė nuo pradinės s.

Algoritmas naudoja FIFO eilę Q pilkoms viršūnėms saugoti.

Paieška į plotį (breadth - first search)

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

Kodėl algoritmo sudėtingumas yra $O(V+E)$?

6. Duomenų struktūros aprašančios grafą ir jų privalumai bei trūkumai. Paieškos į gylį algoritmas ir sudėtingumo įvertinimas.

-..-

Paieška į gylį

Kiekvienai viršūnei u priskirsime spalvos atributą $color[u]$, kuris gali įgyti tris reikšmes:

- balta – nenagrinėta,
- pilka – pažymėta, kaip pradėta nagrinėti,
- juoda – išnagrinėta viršūnė.

Kiekvienai viršūnei u priskiriamas atributas $\pi[u]$, nurodantis kokia viršūnė eina prieš ją ir atributas $d[u]$ nurodantis kokiu laiko momentu pradėta nagrinėti, $f[u]$ – kada baigta.

$d[u] < f[u]$, iki $d[u]$ – viršūnė balta, nuo $d[u]$ iki $f[u]$ – viršūnė pilka, po $f[u]$ – juoda.

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

Paieška į gylį

```
DFS-VISIT(u)
1  color[u] ← GRAY        ▷ White vertex u has just been discovered.
2  time ← time+1
3  d[u] ← time
4  for each v ∈ Adj[u]    ▷ Explore edge (u, v).
5       do if color[v] = WHITE
6             then π[v] ← u
7                  DFS-VISIT(v)
8  color[u] ← BLACK       ▷ Blacken u; it is finished.
9  f[u] ← time ← time+1
```

Bendras laikas Θ(V+E)

7. Topologinis rikiavimas bei stipriai susietų komponentų paieškos algoritmas bei jų sudėtingumo įvertinimas.

Taikomas orientuotam grafui be ciklų. Pvz. briaunos išreiškia, koks dalykas prieš kokį turi būti apsirengtas (pvz. kojinės prieš batus).

```
TOPOLOGICAL-SORT(G)
1  call DFS(G) to compute finishing times v.f for each vertex v
2  as each vertex is finished, insert it onto the front of a linked list
3  return the linked list of vertices
```

Topologinio rikiavimo sudėtingumas Θ(V+E), kai paieška į gylį yra Θ(V+E) ir įterpti kiekvieną viršūnę į sąrašo pradžią užima O(1).

Orientuotas grafas yra stipriai susietas, jei bet kurios dvi viršūnės pasiekiamos viena iš kitos. Du kartus vykdoma paieška į gylį, vieną kartą su G, kitą kartą su G^T.

```
STRONGLY-CONNECTED-COMPONENTS(G)
1  call DFS(G) to compute finishing times u.f for each vertex u
2  compute G^T
3  call DFS(G^T), but in the main loop of DFS, consider the vertices
       in order of decreasing u.f (as computed in line 1)
4  output the vertices of each tree in the depth-first forest formed in line 3 as a
       separate strongly connected component
```

Sudėtingumas Θ(V+E).

8. Minimalūs jungūs (dengiantys) medžiai. Jų radimo algoritmų sudarymo metodika (Bendras algoritmas). Teiginys apie lengviausią briauną.

Minimalūs jungūs grafai

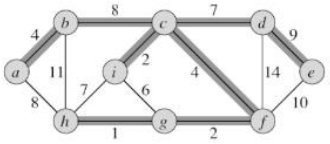
Duotas grafas $G=(V,E)$ ir svorių funkciją:

$$w: E \to R,$$

čia V – viršūnių aibė, E – lankų (briaunų) aibė.

Tikslas: rasti neciklinį poaibį $T \subseteq E$, kuris jungtų visas viršūnes ir turėtų minimalų bendrą svorį:

$$w(T) = \sum_{(u,v)\in T} w(u,v).$$



Bendras algoritmas

```
GENERIC-MST(G, w)
1  A ← ∅
2  while A does not form a spanning tree
3       do find an edge (u, v) that is safe for A
4          A ← A ∪ {(u, v)}
5  return A
```

Teorema 23.1. Turim jungų neorientuotą grafą $G=(V,E)$, su svorių funkcija $w: E \to R$. Tegul poaibis $A \subseteq E$, priklauso kokiam nors minimaliam jungiam grafui, o $(S, V-S)$ – pjūvis G suderintas su A, o (u,v) – lengva (lengviausia) briauna kertanti pjūvį. Tada (u,v) yra saugi briauna A.

9. Kruskalo algoritmas. Sudėtingumo įrodymas. Algoritme naudojamų aibių struktūra ir veiksmų su jomis sudėtingumo įvertimas.

Kruskalo algoritme imama iš eilės po vieną trumpiausią briauną ir ji įtraukiama, jei nesusidaro ciklas. Taip gaunamas minimalus dengiantis medis.

Kruskalo algoritmas

```
MST-KRUSKAL(G, w)
1  A ← ∅
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
```

$O(E \log_2 V)$

Pagalbinės funkcijos:

```
MAKE-SET(x)
1  p[x] ← x
2  rank[x] ← 0

UNION(x, y)
1  LINK(FIND-SET(x), FIND-SET(y))
```

```
LINK(x, y)
1  if rank[x] > rank[y]
2     then p[y] ← x
3     else p[x] ← y
4          if rank[x] = rank[y]
5            then rank[y] ← rank[y] + 1
```

```
FIND-SET(x)
1  if x ≠ p[x]
2     then p[x] ← FIND-SET(p[x])
3  return p[x]
```

$O(E \log_2 V)$, nes

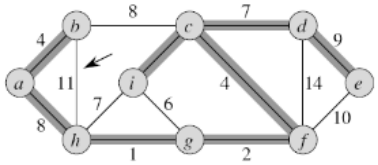
5–8 eil. vykdomo $O(E)$ paieškos ir jungimo operacijų aibių medyje nesikertančių aibių.

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

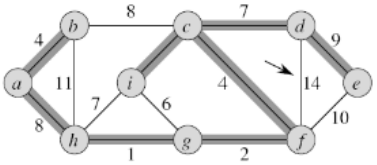
Kadangi grafas jungus $|E| \geq |V| - 1$ todėl $O(E\alpha(V))$, $\alpha(V) = \log_2 V = \log_2 E$, tai sudėtingumas $O(E \log_2 E)$ (su 3 eil.).

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

(m)

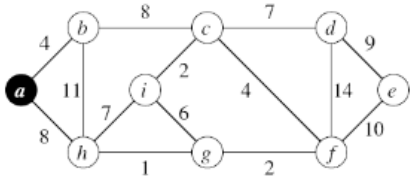


(n)

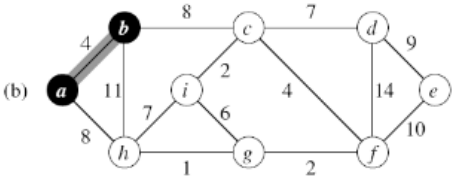


10. Prima algoritmas. Sudėtingumo įrodymas. Algoritme naudojamų struktūros ir veiksmų su jomis sudėtingumo įvertimas.

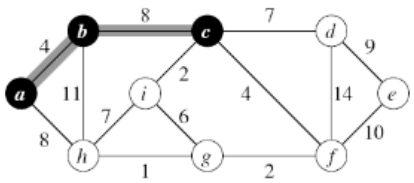
a)



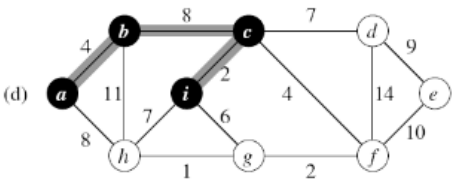
(b)



c)



(d)



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

Sudėtingumas

$O(E \log_2 V)$, nes

Išėmimas iš eilės vykdomas $O(\log_2 V)$ laiko.

9 eil galima įvykdyti per $O(1)$. Kaip?

11. Trumpiausi keliai iš vienos viršūnės. Relaksacijos metodas. Belmano –Fordo algoritmas. Sudėtingumo įvertinimas.

Trumpiausias kelias iš vienos viršūnės

Duotas grafas $G=(V,E)$ ir svorių funkciją:

$$w: E \to R,$$

čia V – viršūnių aibė, E – lankų (briaunų) aibė.

Kelias $p=\langle v_0, v_1, \ldots, v_k \rangle$, o jo svoris $w(p)=\sum_{i=1}^{k} w(v_{i-1}, v_i)$.

Tikslas: rasti trupiausią kelią iš viršūnės u į viršūnę v, kuris turi mažiausią svorį

$$\delta(u,v)=\begin{cases}\min\{w(p): u \xrightarrow{p} v\}, & \text{jei egzistuoja}\\ \infty, & \text{jei neegzistuoja.}\end{cases}$$

Šis uždavinys leidžia spręsti:

- Ieškoti trumpiausio kelio iki nurodytos viršūnės;
- Ieškoti trumpiausio kelio tarp poros viršūnių;
- Ieškoti trumpiausių kelių tarp visų grafo viršūnių.

Jei kelyje yra neigiamo svorio ciklas, tai bendras kelio svoris tarp dviejų viršūnių minus begalybė.

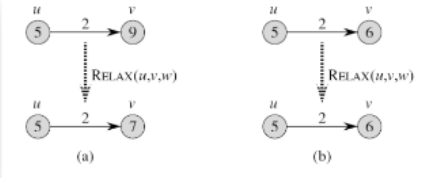
Relaksacijos metodas

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

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

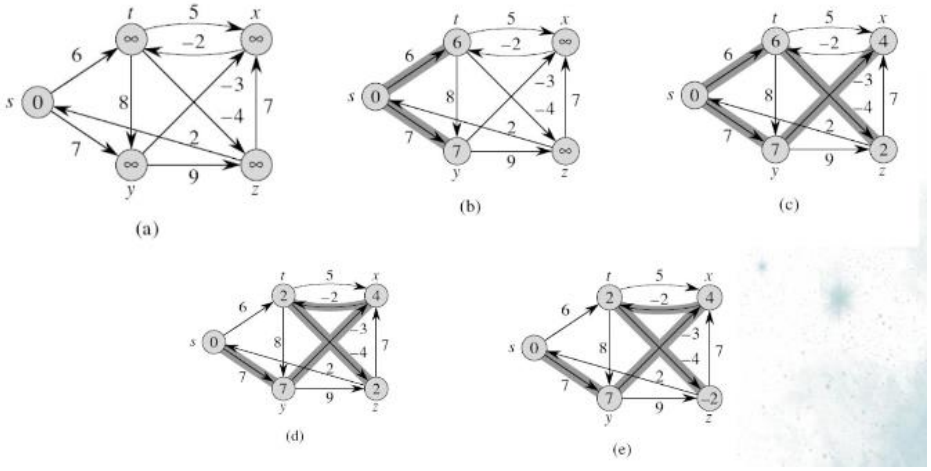
Darbo laikas $\Theta(V)$.

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



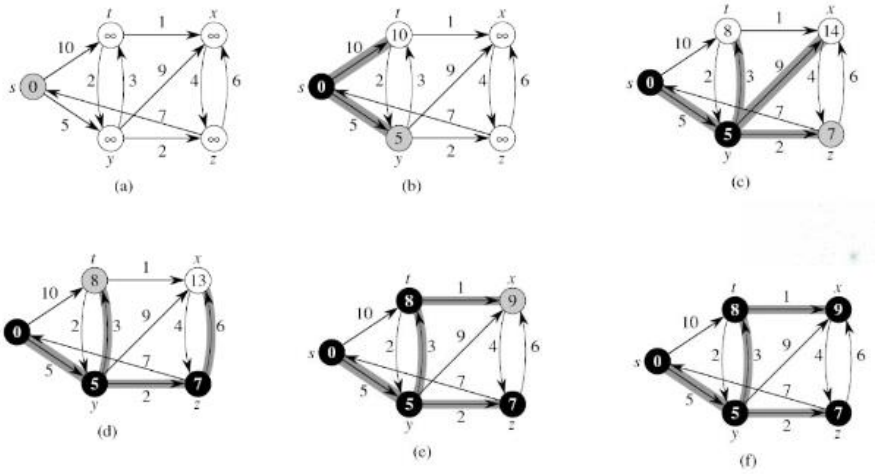

```
BELLMAN-FORD(G, w, s)
1  INITIALIZE-SINGLE-SOURCE(G, s)
2  for i ← 1 to |V[G]| − 1
3       do for each edge (u, v) ∈ E[G]
4            do RELAX(u, v, w)
5  for each edge (u, v) ∈ E[G]
6       do if d[v] > d[u] + w(u, v)
7            then return FALSE
8  return TRUE
```

Sudėtingumas $O(V\,E)$.



12. Trumpiausi keliai iš vienos viršūnės. Relaksacijos metodas. Deikstra algoritmas. Sudėtingumo įvertinimas.

-..-

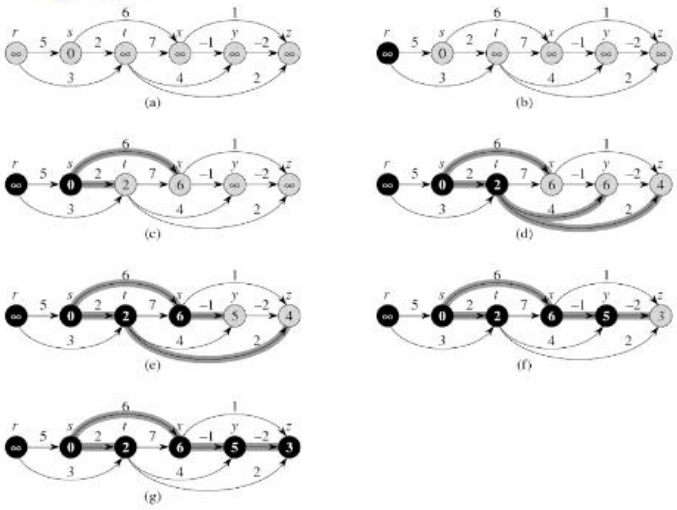


DIJKSTRA(G, w, s)
1 INITIALIZE-SINGLE-SOURCE(G, s)
2 $S \leftarrow \emptyset$
3 $Q \leftarrow V[G]$
4 **while** $Q \neq \emptyset$
5 **do** $u \leftarrow$ EXTRACT-MIN(Q)
6 $S \leftarrow S \cup \{u\}$
7 **for** each vertex $v \in Adj[u]$
8 **do** RELAX(u, v, w)

Sudėtingumas $O(V^2 + E) = O(V^2)$.

13. Trumpiausių kelių paieška iš vienos viršūnės orientuotame acikliniame grafe. Sudėtingumo įvertinimas.

Pavyzdys:



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 **do for** each vertex $v \in Adj[u]$
5 **do** RELAX(u, v, w)

Sudėtingumas $\Theta(V + E)$

14. Daugiagijo dinaminio programavimo metodika.

Lygiagretaus programavimo strategijos: 1. Kiekvienam procesoriui/branduoliui atskira atmintis 2. Visiems procesoriams/branduoliams bendra atmintis.

Daugiagijo (lygiagretaus) programavimo metodai/principai:

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

Idealus kompiuteris su P procesorių per vieną žingsnį gali atlikti P darbo vienetų.

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

```
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
```

Įprastinis SQUARE-MATRIX-MULTIPLY sudėtingumas yra $\Theta(n^3)$. Bendras kodo laikas $\Theta(\lg n)+\Theta(\lg n)+\Theta(n)=\Theta(n)$. Lygiagretaus veikimo laikas $\Theta(n^3)/\Theta(n)=\Theta(n^2)$.

```
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
```

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

$$M_\infty(n) = M_\infty(n/2) + \Theta(\lg n) = \Theta(\lg^2 n).$$

Išlygiagretinimo koeficientas $M_1(n)/M_\infty(n) = \Theta(n^3/\lg^2 n)$ labai aukštas.

16. Daugiagijai rikiavimo algoritmai ir jų vykdymo laikų bei lygiagretinimo koeficientų įvertinimas.

```
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)
```

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

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

Išlygiagretinimo koeficientas: $MS'_1(n)/MS'_\infty(n) = \Theta(\lg n)$, tai yra gan blogai, nes esant keliems procesoriams, bus tiesinis greitis, bet esant šimtams procesorių, greitis nebus efektyvus.

```
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
```

blogiausiu atveju Binary-Search sudėtingumas Θ(lgn).

```
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
```

$$PM_1(n) = PM_1(\alpha n) + PM_1((1-\alpha)n) + O(\lg n) = O(n)$$

$$PM_\infty(n) = PM_\infty(3n/4) + \Theta(\lg n) = \Theta(\lg^2 n)$$

The parallelism of P-MERGE is $PM_1(n)/PM_\infty(n) = \Theta(n/\lg^2 n)$.

17. Amortizacinė algoritmų analizė.

Tikslas surasti viršutinį laiko įvertį, reikalingą atlikti seką veiksmų su duomenų struktūra. Tai atliekama vidurkinant visų operacijų laiką. Tai leidžia parodyti, kad net esant „brangioms“ operacijoms, jos nedaro didelės įtakos. ARBA > Siekiama įvertinti ne atskirų funkcijų sudėtingumą, bet gauti bendrą programos/algoritmo sudėtingumo įvertinimą.

Trys amortizacinės analizės: agregatinė analizė, apskaitos metodas, potencialų metodas.

Agregatinėje analizėje nustatome viršutinę ribą sekos, sudarytos iš n operacijų. Blogiausiu atveju, esant n duomenų, amortizuojanti kaina (vidutinė operacijos kaina) yra T(n)/n. Pvz. turim steką, PUSH ir POP operacijos atliekamos per O(1). Sekos iš n laikas bus Θ(n). Pridedam operaciją MULTIPOP(S,k), kuri pašalins k elementų iš steko S. Blogiausiu atveju MULTIPOP prasuks while ciklą n kartų, jei reikės šalinti visus n elementų, o kviečiant tą metodą n kartų, sudėtingumas gausis $\Theta(n^2)$. Naudojant agregatinę analizę, nors viena MULTIPOP operacija gali būti brangi, bendra PUSH, POP IR MULTIPOP kaina yra Θ(n), nes POP nebus kviečiama daugiau negu PUSH, tai pereis per visus elementus tik. Todėl amortizacinėje analizėje vienos operacijos sudėtingumas Θ(n)/n=O(1).

Apskaitos metodas

Priskiriami skirtingos kainos skirtingom operacijom - operacijų kainas gali būti parinktos didesnės arba mažesnės nei yra iš tikrųjų) – šias kainas vadiname **amortizuojančiomis kainomis**

Kai amortizuojanti kaina viršija aktualią kainą, skirtumą priskiriame „**kreditui**“ – kurį vėliau galėsime kompensuoti iš tų operacijų, kurių amortizuojanti kaina yra mažesnė nei aktuali.

Agregatinėje analizėje visų operacijų kaina vienoda, kai apskaitos metode gali skirtis

Turi būti tenkinama sąlyga: $\sum_{i=1}^{n} \hat{c}_i \geq \sum_{i=1}^{n} c_i$

arba kitaip sakant „**kreditas**“ visada turi būti teigiamas

Potencialų metodas

Vietoje „**kreditų**“ naudojama „**potencinės energijos**“ sąvoka (arba tiesiog potencialas) susieta su duomenų struktūra.

Atliekam n iteracijų. Pradžioje, duomenų struktūra D_0. Kiekvienos iteracijos kaina yra reali kaina i-tosios iteracijas c_i ir D_i duomenų struktūros pokytis lyginant su praėjusios iteracijos duomenų struktūra D_{i-1}.

Potencialo funkcija $\Phi(D_i)$ gražina realią reikšmę, tam tikrai duomenų struktūrai, ir tam tikros iteracijos amortizacinė kaina išreiškiama:

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

Visa amortizuojanti kaina lygi:

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

Praktikoje, ne visuomet žinome, kiek iteracijų bus atliekama, todėl visiems i pareikalaudami, kad $\Phi(D_i) \geq \Phi(D_0)$ - garantuojame, kad „sumokėsime į priekį“, ir reali operacijų kaina neviršys amortizuojančios kainos.

Dažniausiai priimama, kad $\Phi(D_0) = 0$, o $\Phi(D_i) \geq 0$ visiems i.

18. NP sudėtingumas. Ir P ir NP klasės. Uždavinių pavyzdžiai.

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.

--->

NP sudėtingumo uždavinių klasė

$O(n!)$

Tarkime, galim generuoti 10^9 variantų per sekundę, tada penkiolikos daiktų išdėstymas truks 22 min. , 18 daiktų visų variantų (keitinių) sugeneravimas truks 74 paras.

$O(n^2)$ ir $O(2^n)$ sudėtingumo algoritmai. Pirmu atveju sudėtingumas iš auga 4 kartus padidinus 2 kartus imtį, o antruoju atveju dvigubėja pridėjus vieną elementą prie duomenų imties.

P: pvz sudauginti dvi matricas, žinome tikslų polinominį sudėtingumą. Jie yra „išsprendžiami“.
NP: turime n skirtingų objektų a1, a2, ..., an. Reikia sugeneruoti visus skirtingus jų kėlinius. Skirtingų kėlinių yra n!, todėl O(n!).