



Algoritmid ja andmestruktuurid

- Dünaamiline planeerimine
 - Floyd lühimate teede algoritm
 - Maatriksite korrutamise järjekord



Dünaamiline planeerimine

- Paljude ülesannete puhul võib “jaga ja valitse” strateegia anda küll tulemuse, aga see on väga ebaefektiivne

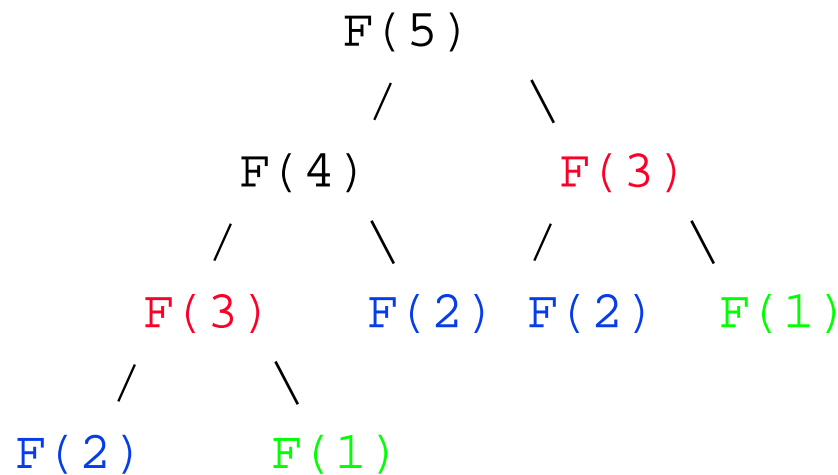
```
int fib( int n ) {  
    if ( n < 2 ) return n;  
    else  
        return fib(n-1) + fib(n-2);  
}
```

- $W(n) = W(n-1) + W(n-2) \in O(1.68...^n)$



Fibonacci arvud parem algoritm

Algoritmi aegluse põhjus -
lahendame samu alamprobleeme korduvalt



💣 Alustame arvutamist väiksematest argumentidest, peame tulemused meeles ja arvutame järjest tulemusi suurematele argumentidele



Fibonacci: dünaamiline planeerimine

- ```
int fib(int n)
{
 int f[n+1];
 f[1] = f[2] = 1;
 for (int i = 3; i <= n; i++)
 f[i] = f[i-1] + f[i-2];
 return f[n];
}
```
- Alustame alamülesannetest
- Salvestame kõik alamülesannete tulemused
- $W(n) \in O(n)$



# Tasuta lõunasöök?

---

- Võimaldab lahendada efektiivsemalt probleeme, kus alamülesannete alamülesanded kattuvad.
- Maksta tuleb alamülesannete lahenduste meelespidamisega.



# Markide paigutamise probleem



- Meil on kasutada 10, 40 ja 50 sendised margid. Leida millises kombinatsioonis tuleks marke panna mingi summa maksmiseks, et markide arv oleks minimaalne.
- Algoritm stiilis “võta suurimaid münte niipalju kui saad ja siis sellele järgneva suurusega jne” (ahne algoritm) ei tööta!
- Sisuliselt otsime lahendust võrrandile
$$a \cdot 50 + b \cdot 40 + c \cdot 10 = \text{summa} \quad \text{või}$$
$$a \cdot 5 + b \cdot 4 + c \cdot 1 = \text{summa}/10$$
nii et  $a+b+c$  oleks minimaalne



## 2.70 minimaase arvu markidega

| <b>50 sendiste<br/>markide<br/>arv</b> | <b>40 sendiste<br/>markide<br/>arv</b> | <b>10 sendiste<br/>markide<br/>arv</b> | <b>vajalike<br/>markide<br/>arv kokku</b> |
|----------------------------------------|----------------------------------------|----------------------------------------|-------------------------------------------|
| <b>5</b>                               | <b>0</b>                               | <b>2</b>                               | <b>7</b>                                  |
| <b>4</b>                               | <b>1</b>                               | <b>3</b>                               | <b>8</b>                                  |
| <b>3</b>                               | <b>3</b>                               | <b>0</b>                               | <b>6</b>                                  |



# Täielik algoritm (jõumeetod, *brute force*)

---

```
findStamps(N):
 n = N/10;
 min = n+1
 for a in 0 ... n:
 for b in 0 ... n:
 for c in 0 ... n:
 if a*5 + b*4 + c*1 == n:
 if a+b+c < min
 min = a+b+c
 save <a,b,c>
 print saved <a, b, c>
```

Keerukus  $O(n^3)$





# Idee 💣

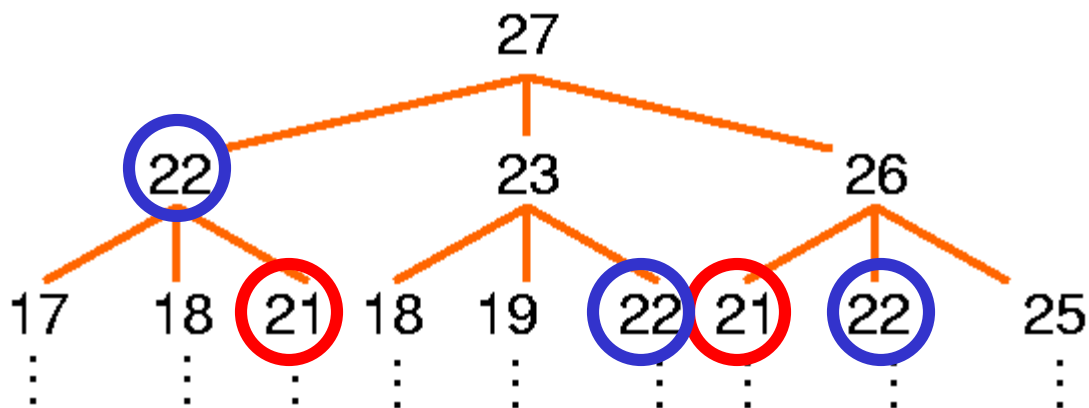
- Tuleb defineerida ülesande lahendus alamülesannete jadana
- Alamülesanne on mingi väiksema summa kokkusaamine.

```
markide_arv(27) =
 markide_arv(22) + 1 | // kui lisati 5
 markide_arv(23) + 1 | // kui lisati 4
 markide_arv(26) + 1 // kui lisati 1
```



# Kasutame rekursiooni

$$M(i) = \min \begin{cases} 0 & i=0 \\ 1+M(i-5) & i \geq 5 \\ 1+M(i-4) & i \geq 4 \\ 1+M(i-1) & i \geq 1 \end{cases}$$



Time:  $> 3^{N/5}$

**Korduvad alamülesanded**

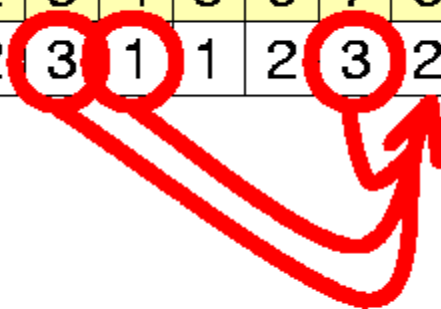


# Alustame alamprobleemidest

for  $i = 0, \dots, N$  do  $M(i) = \min \begin{cases} 0 & i=0 \\ 1+M(i-5) & i \geq 5 \\ 1+M(i-4) & i \geq 4 \\ 1+M(i-1) & i \geq 1 \end{cases} ;$

- Time:  $O(N)$

|      |   |   |   |   |   |   |   |   |   |   |    |    |    |    |    |
|------|---|---|---|---|---|---|---|---|---|---|----|----|----|----|----|
| i    | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 |
| M(i) | 0 | 1 | 2 | 3 | 1 | 1 | 2 | 3 | 2 |   |    |    |    |    |    |



$$1 + \text{Min}(3, 1, 3) = 2$$



# Võrdlus

---

- Jaga ja valitse,  
Tagasivõtmisega algoritm
  - rekursiivne programm
  - top - down
  - jagab ülesande sõltumatuteks alamülesanneteks
  - kasutab rekursiivset seost alamülesannete vahel edaspidi - põhiülesanne enne alamülesandeid
- Dünaamiline programmeerimine
  - iteratiivne programm
  - bottom - up
  - kasutab alamülesannete lahendusi korduvalt
  - kasutab rekursiivset seost alamülesannete vahel tagurpidi - alamülesanded enne põhiülesannet



# Dünaamilise planeerimise algoritmi tegemine

⌚ Defineeri optimaalse lahendi saamine rekursiivselt alamülesannetest

- määratle alamülesannete jada
- leia seos ülesande ja tema alamülesande vahel
- defineeri alamülesannete lahenduste hoidmiseks vajalik andmestruktuur

$$\text{for } i = 0, \dots, N \text{ do } M(i) = \min \left\{ \begin{array}{ll} 0 & i=0 \\ 1+M(i-5) & i \geq 5 \\ 1+M(i-4) & i \geq 4 \\ 1+M(i-1) & i \geq 1 \end{array} \right\} ;$$

$M[i]$  - optimaalne markide arv väärtuse  $i$  jaoks

$V[i]$  - viimasena valitud margi väärtus



# Dünaamilise planeerimise algoritmi tegemine



- Lahenda see rekursioon altpoolt -  
alates elementaarsetest alamülesannetest  
– Salvesta vajalikud alamülesannete tulemused

```
void stamps(int value) {
 int M[value], V[value];
 for(int i=1; i <= value; i++) {
 M[i] = ∞;
 if((i >= 5) && (M[i] > M[i-5]+1))
 { M[i] = M[i-5]+1; V[i]=5 }
 if((i >= 4) && (M[i] > M[i-4]+1))
 { M[i] = M[i-4]+1; V[i]=4 }
 if((i >= 1) && (M[i] > M[i-1]+1))
 { M[i] = M[i-1]+1; V[i]=1 }
 }
}
```



# Dünaamilise planeerimise algoritmi tegemine

⌚ Konstrueeri lahend salvestatud tulemuste alusel

Prindib summa  $n$  saamiseks viimasena lisatud margi  $V[n]$  ja optimaalse markide komplekti lahendi viimase margi võrra väiksema ülesande jaoks kuni kogu summa on kaetud

```
void printstamps(int n) {
 while(n > 0) {
 print V[n];
 print ", ";
 n = n - V[n];
 }
}
```



# Alamülesannete optimaalsusprintsiiip

---

- Dünaamiline programmeerimine sobib paljude ülesannete jaoks, kus on vaja leida optimaalne lahend.
- Sobib ainult juhul kui on täidetud **alamülesannete optimaalsusprintsiiip**:

*Kui lahend on optimaalne, siis on kõik selle saamiseks kasutatud alamülesannete lahendid optimaalsed oma alamülesannete jaoks.*

- Optimaalsusprintsiiip ei kehti iga probleemi korral. Kehtivust tuleb igal juhul eraldi näidata!





# Tõestus markide paigutamise algoritmile

---

- **Teoreem:** Kui viimane kleebitud mark väärtusega  $v$  moodustas optimaalse lahenduse väärtuse  $S$  jaoks, siis moodustasid eelmised margid optimaalse lahenduse väärtuse  $S-v$  jaoks.
- **Tõestus:** Kui lahend ilma viimasena valitud margita (summa  $S-v$ ) poleks olnud optimaalne, siis saaksime parandada ka lõpptulemust (summa  $S$ ), mis on vastuolus algtingimusega



# Kõikide alamülesannete lahendid on optimaalsed

Eeldades et

- Meie lahend on optimaalne
- Kasutatud alamülesande lahend ei olnud optimaalne

⇒ Kasutades alamülesande paremat lahendit peaksime saama veel parema lahenduse

⊗ Vastuolu, MOTT

|      |   |   |   |   |   |   |   |   |   |   |    |    |    |    |    |
|------|---|---|---|---|---|---|---|---|---|---|----|----|----|----|----|
| i    | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 |
| M(i) | 0 | 1 | 2 | 3 | 1 | 1 | 2 | 3 | 2 |   |    |    |    |    |    |


$$1 + \text{Min}(3, 1, 3) = 2$$



# Floyd'i lühimate teede algoritm

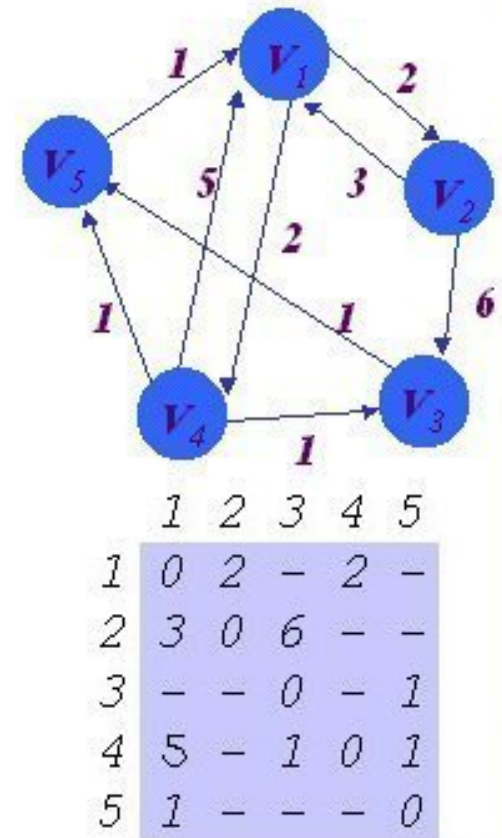
---

- On antud suunatud märgendatud graaf, mis on esitatud naabrusmaatriksina  $W[i][j]$ .
- Kõigi tippude vahel ei pruugi serva olla.
- Leida lühimad teed igast tipust igasse teise tippu
- Naiivne algoritm - vaatame läbi kõikide tippude kohta kõik võimalikud teed kõigisse tippudesse ja leiame parima.  
 *$n$  tipuga täieliku graafi puhul umbes  $n!$  teed*
- Floyd'i algoritm töötab  $O(n^3)$  keerukusega



# Graafi esitus naabrusmaatriksina

- $n$  tipuga graafi jaoks on  $n \times n$  maatriks
- maatriksi elemendi  $W[i][j]$  väärtus tähistab serva tipust  $i$  tippu  $j$  märgendi väärtust
- kui kahe tipu vahel serva ei ole, siis -  
(või  $\infty$ ,  $\text{maxInt}$ )
- maatriksi peadiagonaali ( $W[i][i]$ ) väärtused on 0
- vastuseks tahame lühimate teede pikkusi maatriksis  $D[i][j]$
- Kuidas defineerida alamülesandeid nii et optimaalsusprintsip oleks täidetud?
- Graafi tippude lisamine ühekaupa?
- ei tööta hästi kuna
  - tipu lisamisel tuleb senine töö ümber arvutada
  - leida kõik teed uude tippu ja sellest mujale





# Rekursiivne lahendamine

## Animatsioon

- Lähendusprotseduur

- $D^{(k)}[i][j]$  lühimad teed,  
mis läbivad tippe  $1 \dots k$
- $D^{(0)}[i][j] = W[i][j]$
- $D^{(n)}[i][j] = D[i][j]$

- Variandid - tee tipust  $i$  tippu  $j$  läbi tippude  $1 \dots k$

- $D^{(k)}[i][j] = D^{(k-1)}[i][j]$  tee ei läbi tippu  $k$
- $D^{(k)}[i][j] = D^{(k-1)}[i][k] + D^{(k-1)}[k][j]$  tee läbib tippu  $k$

- Rekursiivne seos

- $D^{(k)}[i][j] = \min (D^{(k-1)}[i][j], D^{(k-1)}[i][k] + D^{(k-1)}[k][j])$



# 🕒 Floyd'i algoritm - lühimate teede pikkus

```
void floyd (int n, const number W[][], number D[][])
```

```
{ index i, j, k;
```

```
D = W;
```

```
for (k = 1; k <= n; k++) // $D^{(1)} \dots D^{(k)}$
```

```
 for (i = 1; i <= n; i++) // $D[1][1] \dots D[n][n]$
```

```
 for (j = 1; j <= n; j++)
```

```
 D[i][j] = min(D[i][j], D[i][k] + D[k][j]);
```

```
}
```

**$D^{(k)}$  väärtused arvutatakse  $D^{(k-1)}$  väärtuste kaudu. Kas ja miks võime muuta massiivi  $D^{(k)}$  väärtusi  $D^{(k)}$  arvutamise käigus?**



# 🕒 Floyd'i algoritm - lühimate teede pikkus

```
void floyd (int n, const number W[][], number D[][])
```

```
{ index i, j, k;
```

```
D = W;
```

```
for (k = 1; k <= n; k++) // $D^{(1)} \dots D^{(k)}$
```

```
 for (i = 1; i <= n; i++) // $D[1][1] \dots D[n][n]$
```

```
 for (j = 1; j <= n; j++)
```

```
 D[i][j] = min(D[i][j], D[i][k] + D[k][j]);
```

```
}
```

**Tulemuseks lühima tee pikkus igast punktist teise, aga mitte teekonda ennast**



# 🕒 Floyd'i algoritm - lühimad teed koos pikkustega

```
void floyd (int n, const number W[][], number D[][],
 index P[][]) // suurima indeksiga läbitud tipp
{ index i, j, k;
 for (i = 1; i <= n; i++)
 for (j = 1; j <= n; j++)
 P[i][j] = 0;

 D = W;

 for (k = 1; k <= n; k++) // D(1) ... D(k)
 for (i = 1; i <= n; i++) // D[1][1] ... D[n][n]
 for (j = 1; j <= n; j++)
 if (D[i][k] + D[k][j] < D[i][j])
 { P[i][j] = k;
 D[i][j] = D[i][k] + D[k][j];
 }
}
```

**Lisame massiivi P, mis võimaldab leida ka teekonna**





# Floyd'i algoritm - Lühimate teede leidmine

---

```
void path (index i, j) {
 if(P[i][j] != 0) {
 path(i, P[i][j]);
 print P[i][j];
 print ",";
 path(P[i][j], j);
 }
}
```

- Dünaamilise planeerimise algoritmid kodeerivad lahenduse tihti vahetulemuste massiivi ja tulemus tuleb sealt peale põhialgoritmi lõppu konstrueerida või üles otsida.



# Floyd'i algoritm - optimaalsusprintsiiip

---

Kui alamülesandele  $D^{(k-1)}$  leiduks parem lahendus, siis võiks saaksime parema lahenduse ka oma ülesandele  $D^{(k)}$ .

$\Rightarrow$  Kui  $D^{(k)}$  on optimaalne, siis on seda ka  $D^{(k-1)}$ .



# Floyd vs Dijkstra lühima tee algoritmid

---

## Floyd

- Igast tipust igasse tippu
- Keerukus  $O(|V|^3)$
- Leiab õige teepikkuse ka negatiivse kaaluga servadega
  - Negatiivse kaaluga tsüklid pole lubatud

## Dijkstra

- Ühest tipust igasse tippu
- $O(|E| + |V| \log |V|)$  või  $O(|V|^2)$ 
  - Igast tipust
  - $O(|V|(|E| + |V| \log |V|))$  või  $O(|V|^3)$
- Negatiivse kaaluga servad pole lubatud



# Maatriksite ahela korrutamine

---

- On vaja korrutada rida maatrikseid:

$$A1 \ A2 \ A3 \ \dots \ A_n$$

- Maatriksite korrutamine on assotsiatiivne, st

$$A1 \ ( \ A2 \ A3 \ ) = ( \ A1 \ A2 \ ) \ A3$$

- Tulemuse võib saada kahel erineval viisil. Erinevalt täisarvude korrutamisest on järjekorral tähtsus
  - $n \times m$  ja  $m \times p$  mõõtudega maatriksite korrutamise hind on  $O(nmp)$  elementaarset korrutamisoperatsiooni
  - $n \times m$ ,  $m \times p$  ja  $p \times r$  maatriksite korrutamise hinnaks on  $O(mpr + nmr)$  või  $O(nmp + npr)$



# Korrutamise järjekord on oluline

- Valesti valitud järjekord võib olla kallis, näiteks:  
A1 (10x100), A2 (100x5), A3 (5x50)
- ( A1 A2 ) A3 hind on  
A1A2      10x100x5= 5000  $\Rightarrow$  A1 A2 (10x5)  
(A1A2)A3    10x5x50 = 2500  $\Rightarrow$  A1A2A3 (10x50)  
Kokku                = **7500**
- A1 ( A2 A3 ) hind on  
A2A3      100x5x50= 25000  $\Rightarrow$  A2A3 (100x5)  
A1(A2A3) 10x100x50 = 50000  $\Rightarrow$  A1A2A3 (10x50)  
Kokku                = **75000**
- Korrutamise järjekord on võimalik enne tegelikku korrutamistpaika panna maatriksite suuruse järgi




# Optimaalsusprintsip kehtib

- Kui jagame oma korrutamiste ahela kahe alamahela korrutamiseks:

$$(A_1 A_2 A_3 \dots A_k) (A_{k+1} \dots A_n)$$

siis saab tulemus olla optimaalne ainult juhul, kui alamahelad on korrutatud optimaalses järjekorras. Kui kumbagi alamahelat saaks korrutada odavamalt, siis muutuks odavamaks ka lõpptulemus.

- Optimaalse tulemuse leidmiseks peame võrdlema kõiki paare( $k=1\dots n$ ). 
  - Võrdlemiseks on vaja teada optimaalseid lahendusi alamahelatele  $A_1 \dots A_k$  ja  $A_k \dots A_n$



# Rekursiivne seos

- $(A_1 A_2 A_3 \dots A_k) (A_{k+1} \dots A_n)$  optimaalse lahenduse saamiseks on vaja teada optimaalseid lahendusi alamahelatele  $A_1 \dots A_k$  ja  $A_k \dots A_n$
- $m(i,k)$  – optimaalne korrutamise järjekord ahelale  $A_i \dots A_j$ 
  - $m(1,n)$  – probleemi lahendus
  - $m(i,i) = 0$
  - $m(i,j) = \min_{i < k < j} (m(i,k) + m(k+1,j) + \dim_{i-1} * \dim_k * \dim_j), \quad i < j$
- Valime minimaalse korrutamiste arvu kõigist  $k$  võimalikest väärtustest kasutades väiksemate ülesannete tulemusi.



# Keerukus

- $m(i,i) = 0$
- $m(i,j) = \min_{i < k < j} (m(i,k) + m(k+1,j) + \text{dim}_{i-1} * \text{dim}_k * \text{dim}_j), \quad i < j$
- Rekursiivse lahenduse keerukus on  $O(2^{n-1})$ .
- Vale lähenemine, kasutame alamülesannete lahendusi
  - $m(i,j)$  erinevaid väärtusi on  $O(n^2)$  ( $i, j = 1 \dots n$ )
  - Iga  $m(i,j)$  arvutamise keerukus on  $O(n)$
  - Kokku  $O(n^3)$





# Algoritm

```
int minmult (int n, const int d[], index P[][])
{ index i, j, k, diagonal;
 int M[1...n][1...n];
 for (i = 0; i <= n; i++)
 M[i][i] = 0;
 for (diagonal = 1; diagonal <= n - 1; diagonal++)
 for (i = 1; i <= n - diagonal; i++)
 { j = i + diagonal;
 M[i][j] = mink (M[i][k] + M[k+1][j] + dim[i - 1] *
 dim[k] * dim[j])
 P[i][j] = minimumi andnud k väärtus
 }
 return M[1][n]
}
```

[Animatsioon](#)



# Memoization

---

- Top-down lahendamine vahetulemuste salvestamisega
- Rekursiivselt väljakutsutud alamülesannete tulemused salvestatakse
- Enne uue alamülesande lahendamist kontrollitakse vastuse olemasolu.
- Võrdlus
  - on vaja teada tulemuste indekseerimise struktuuri  
võimalik on kasutada ka paisksalvestust (*hash* tabelit)
  - + vähendab vahetulemuste arvu, kui kõikvõimalikke alamülesandeid pole vaja lahendada
- Võimaldab elegantse lahenduse DP keerukusega



## *Memoization* markide ülesandel

---

$$\text{for } i = 0, \dots, N \text{ do } M(i) = \min \begin{cases} 0 & i=0 \\ 1+M(i-5) & i \geq 5 \\ 1+M(i-4) & i \geq 4 \\ 1+M(i-1) & i \geq 1 \end{cases} ;$$

Rakendame seost rekursiivselt

Puhverdame  $M(i)$  väljakutseid



# Dünaamilise planeerimise algoritmi tegemine

---

- Analüüsi, kuidas ülesande lahendus sõltub alamülesannetest. Kas on korduvaid alamülesandeid?
- Kirjuta välja rekursiivne seos suurema ülesande lahendus jaoks alamülesannete kaudu
- Planeeri andmestruktuur alamülesannete tulemuste hoidmiseks
- Lahenda alamülesanded väiksemast suuremani