Zapiski s predavanj predmeta Algoritmi in podatkovne strukture 2

Alenka Caserman

22. avgust 2006

Kazalo

Del I
Pozor

Manjkajo mi pisani zapiski od 18.5.2006 in 22.5.2006, zato prosim kogarkoli, ki ima ta del zapiskov kolikortoliko urejen, da mi jih fotografira ali skenira in pošlje po mailu. Zaradi manjkajoče vsebine na tem delu zapiskov ni slik, tudi pravilnost teksta je dvomljiva. Hvala!

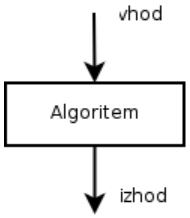
Del II
Uvod

1 Algoritmi

Algoritem je nekakšen program, ki teče na nekem računalniku. Računalnik na katerem algoritem teče pa je lahko bodisi resničen bodisi namišljen (model računanja).

Algoritem:

- dobi neke podatke ⇒ VHOD
- vrne neke podatke ⇒ IZHOD



Slika 1: Diagram preprostega algoritma

Važno: za vsak vhod je izhod natančno definiran oz. določen: $\forall$ vhod: izhod = algoritem(vhod). Kadar to ne velja več, to ni več algoritem ampak procedura (recept za izra cun).

1.1 Modeli računanja (namišljeni računalnik)

Poznamo jih veliko, npr. Türingov stroj, avtomati, RAM, λ-funkcije, rekurzivne funkcije, programi v raznih programskih jezikih, diagram poteka,...

1.1.1 RAM (Random Access Machine)

- stroj z enakopravnim dostopom do pomnilnika
- je dovolj podoben obstoječim, resničnim računalnikom
- procesor:
  - lahko izvaja običajne operacije (aritmetične operacije, operacije nad biti, preusmerjanje - skoki,...)
  - zaporedno izvaja ukaz za ukazom
  - vsaka operacija traja enoto časa
- pomnilnik:
  - celice (besede) so enako dosegljive za procesor
  - vsebuje podatke (program) poljubne velikosti (pomnilniška beseda nima omejene dolžine)

Zanemarimo:

- druge komponente računalnika, npr. V/I enote
- druge podrobnosti, npr. zaokrožitvene napake, ker pomnilniška beseda nima omejene dolžine

PRAM: vzporedni algoritmi, več procesorjev

Prednosti RAM-a:

- dokaj stvarna (realna) ocena porabljenega časa (število izvedenih ukazov, vsak je dolg enoto časa)
- realna ocena poraljenega prostora (število porabljenih celic v pomnilniku)

Pomanjkljivosti RAM-a:

- algoritmi, ki jih pisemo za ta model so dokaj nepregledni (bistvo algoritma se zgubi)

Posledica: RAM si predstavljamo kot **ciljni stroj**, algoritme pa bomo opisali v nekem **višjem programskem jeziku**.

Višji jeziki za zapis algoritmov so namenjeni:

1. **človeskemu bralcu**: prirejeni jezki algolskega tipa, poenostavitve Algola, Pascala, Modula,...
2. **izvajanju na stvarnem računalniku** (Java, C, Oberon-2)

Primer algoritma za dvojiško iskanje:

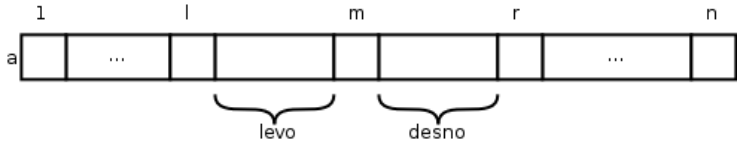
VHOD:

- a - tabela nekih elementov, urejena z relacijo manjši
- n - št. elementov v tabeli
- x - element za katerega nas zanima, če je v tabeli, ě ga ni pa kam bi sodil

IZHOD:

- indeks iskanega elementa v a (če $x \in a$), sicer pa negativni indeks, kamor bi element sodil (če $x \notin a$)

Bistvo algoritma: Če x sodi v del med indeksoma l in r, potem poglej, čr je na **sredini** med l in r. Če ni, nadaljuj iskanje v tistem delu kamor sodi po velikosti, tj. **levo** ali **desno** od srednjega.



Koda:

Listing 1: Koda za metodo binarnega iskanja

```
1  PROCEDURE BinSearch(VAR a: ARRAY OF INTEGER;
2                      x,n: INTEGER) : INTEGER
3  VAR l,m,r : INTEGER;
4  BEGIN l:=1, r:=n;...iscemo v celotni tabeli a
5        WHILE l<=r DO...dokler obstaja nepregledan del tabele a
6              m:=(l+r) DIV 2;
7              IF a[m]>x THEN r:=m-1;
8              ELSE l:=m+1;
9        END
10       IF (r > 0) && (a[r] =x) THEN RETURN r;
11       ELSE RETURN -l;
12 END BinSearch;
```

Sled programa Je tabela parov izbrane vrstice programa v delovanju in vrednosti izbranih spremenljivk, ki priprada programu (algoritmu) in konkretnemu vhodu. Uporabljamo jo za opazovanje delovanja programa.

Npr. za BinSearch in za $a = (9, 13, 27, 32, 41, 48), x = 18$:

št. izbrane vrstice programa v delovanju	vrednosti izbranih spremenljivk

Tabela 1: Tabela sledi

vrstica	l	m	r
4	1	-	6
6	1	3	6
7	1	3	2
6	1	1	2
8	2	1	2
6	2	2	2
8	3	2	2
6	3	2	2

Tabela 2: Tabela sledi za BinSearch

Drevo sledi je sestavljeno iz več sledi(vsaka sled za drug vhod).

2 Preverjanje pravilnosti algoritmo/programov

V praksi je velikokrat pomembno vedeti, če je algoritem pravilen.
Kako dokazati oz. preveriti pravilnost algoritma?
Dva pristopa:

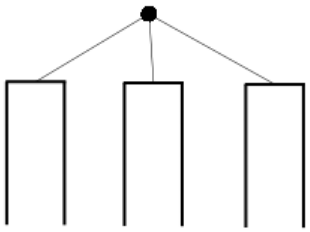
1. s poskusi
2. z logično analizo

2.1 Preverjanje pravilnosti s poskusi

Zamisel Na izbranih (nekaterih) vhodih preveri, če so sledi OK (pravilne, taksne kot morajo biti).

Velja Če kaka sled ni OK $\Rightarrow$ algoritem ni pravilen oz. če so vse mozne sledi OK $\Rightarrow$ algoritem je pravilen. Vseh možnih sledi pa je lahko ogromno:

- končno



Slika 2: Drevo sledi

- neskončno
  - številno mnogo
  - kontinuum

Največkrat pa je vseh možnih sledi preveč.

Primer: množenje s seštevanjem ⇒ kako pomnožiti dve števili med seboj brez operacije množenja

Zamisel: $x * y = y + y + ... + y (x > 0)$

Algoritem

Listing 2: Koda za metodo množenja s seštevanjem

```
1 PROCEDURE Muliply(x,y : INTEGER) : INTEGER
2 VAR u,z : INTEGER;
3 BEGIN z:=0, u:=x;
4         REPEAT z:=z+y; u:=u-1;
5         UNTIL u:=0;
6         RETURN z;
7 END Multiply;
```

Število vhodov je $2^{16} \cdot 2^{16} = 2^{32}$ (ker je INTEGER v Oberonu 16-biten). Iz tega sledi, da bi bilo treba preveriti 2^{32} sledi, kar je v praksi nesmiselno. Preverjanje s poskusi v praksi ni zelo uporabno, razen za iskanje napak oz. dokazovanje nepravilnosti. Potreben je drugačen pristop.

2.2 Preverjanje pravilnosti programov z logično analizo

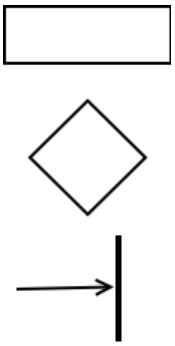
Zamisel Z logičnim sklepanjem ugotoviti ali iz lastnosti algoritma in lastnosti vhoda sledi tudi željena lastnost algoritma.
Kako izraziti lastnost?
S predikatnim računom 1. reda:

- stavki - lastnosti
- pravila sklepanja

2.2.1 Logična analiza na diagramih poteka

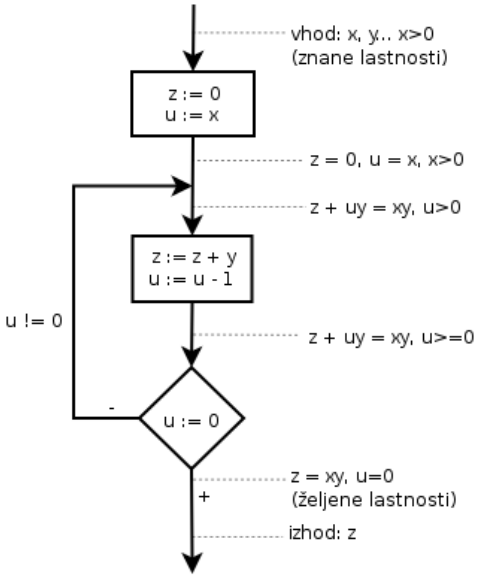
Diagram poteka je:

- usmerjen graf
- en vhod, en ali več izhodov
- 3 vrste vozlišč

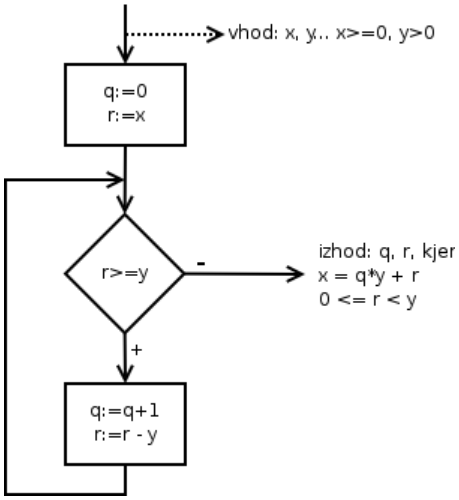


Slika 3: Simboli za diagram poteka

Primer: Množenje s seštevanjem



Primer: Deljenje s pomočjo odštevanja



Kako v načelu poteka dokazovanje pravilnosti?
Izhajamo iz:

- znanih lastnosti vhoda
- željenih lastnosti izhoda

Vsakemu vozlišču pripišemo množico trditev:

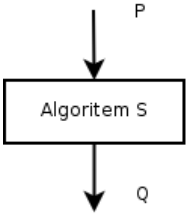
- po eno trditev na vsako vhodno povezavo
- po eno trditev na vsako izhodno povezavo

Kako?
Tako, da velja sklep:

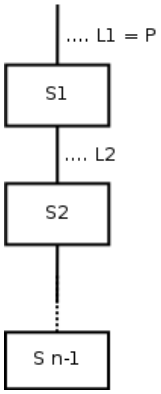
> Če pred izvedbo S velja A, potem po izvedbi S velja C.

To je odvisno od tipa in pomena S.

Cilj Dani algoritem



opisati kot zaporedje korakov

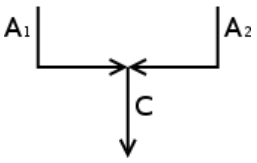


tako da velja: $\forall : i = 1, 2, ...u - 1$, če pred S_i velja L_i, po S_i velja L_{i+1}.

2.2.2 Aksiomi

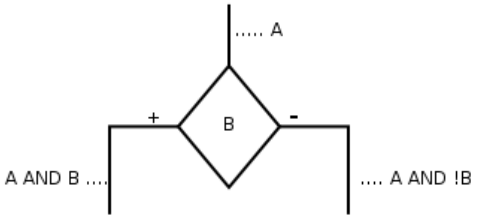
- za **stekališče**:

  Če velja nekaj pred stekališčem, naj velja tudi po njem.



  Kjer za C velja: $A_1 \Rightarrow C \wedge A_2 \Rightarrow C$ (npr. $C = A_1 \vee A_2$).

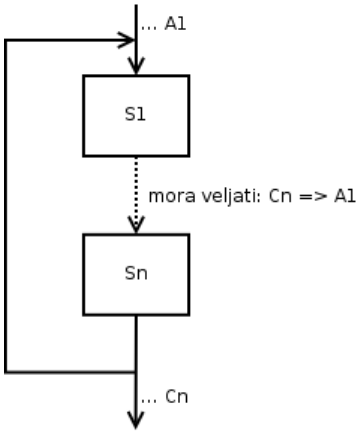
- **pogojni stavek**



- **prirejanje**: po prirejanju $z = f(x)$ dobimo konsekvens tako, da v antecedensu vse proste pogoje

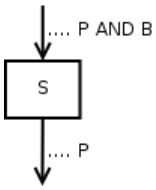
Opozorilo: Vsakič ko prispevamo v $\bigotimes$, tam velja $z + uy = xy, u > 0$. To trditev imenujemo **zančna invarianta**. Zanjo velja:

- je na začetku zanke
- je veljavna vsakokrat ob izvedbi zanke

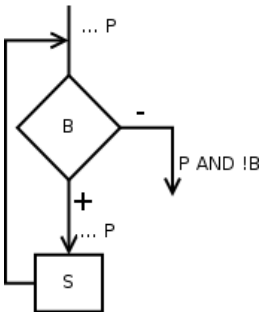


2.2.3 Pravila sklepanja za večje poddiagrame

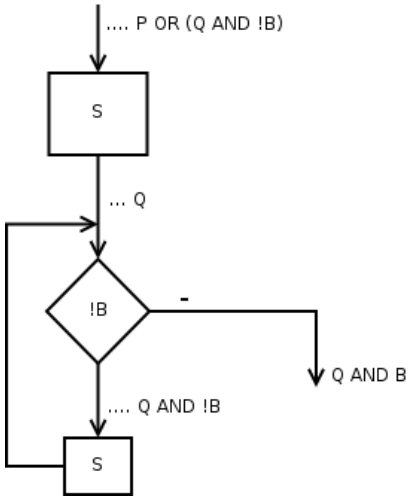
- zanka WHILE, če vemo da velja:



potem



- zanka REPEAT, če vemo da velja:



2.2.4 Logična analiza programa v programskem jeziku

- tehniko enostavno prilagodimo
- prirejanje: $(*P_w^v*)v := w(*P*)$
- sestavljanje skokov: če $(*P*)S_1()*Q*$ in $(*Q*)S_2(*R*)$, potem $(*P*)S_1, S_2(*P_v*)$
- pogojni stavek: če $(*P \wedge B*)$ S $(*P*)$, potem $(*P*)$ WHILE B DO S END $(*P \wedge \overline{B}*)$
- repeat: če $(*P*)$ S $(*Q*)$ in $(*Q \wedge \overline{B}*)$ S $(*Q*)$, potem $(*P*)$ REPEAT S UNTIL B $(*Q \wedge B*)$

2.3 Preverjanje ustavljanja

Pravilnost in ustavljivst sta dva različna pojma.

Trditev o pravilnosti programa:

Če program pride do izstopne točke, potem velja trditev Q.

Trditev o ustavljivosti programa:

Program po končnem številu korakov pride do izstopne točke.

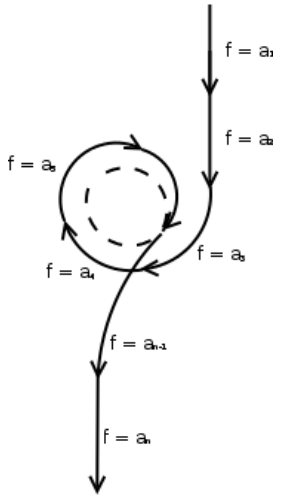
Kako dokazati trditev o ustavljivosti?
Poskušamo s tehniko monotono padajoča funkcija.

Tehnika:

1. izberemo neko **funkcijo f(...)**, ki je odvisna od nekaterih spremenljivk programa in ima lastnost $f(\ldots) \geq lim$ v vsaki točki programa
2. diagram poteka **razdelimo na enostavne odseke** (vsak tak odsek, ki nima zanke), tako da veljata:

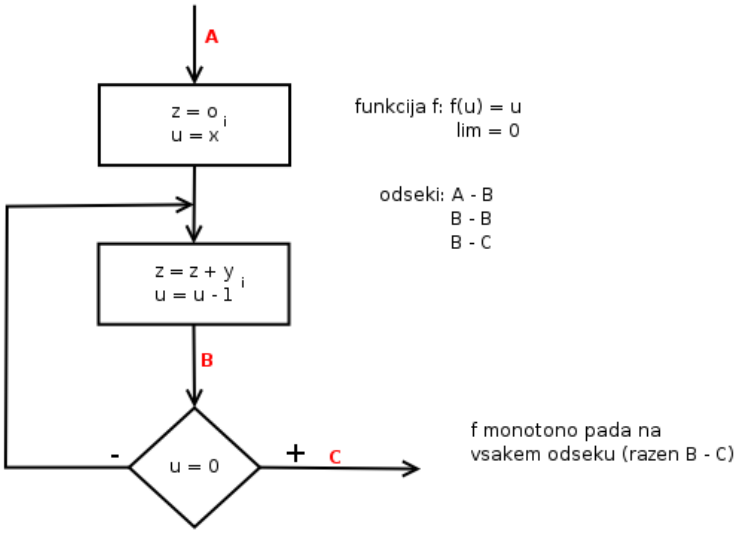
- vsako pot skozi diagram sestavljajo nekateri odseki (ne nujno vi in se lahko ponovijo večkrat)
- na vsakem odseku se vrednost funkcije f strogo zmanjša (razen mogoče na zadnjem, izstopnem odseku)

Če nam uspe izvesti oba koraka lahko zaradi 1. in 2. točke sklepamo, da nobena pot skozi diagram ne more imeti neskončno odsekov. Iz tega sledi, da se program ustavi.



Ker $\forall\, i:\ (a_i > lim \wedge a_i > a_{i+1})$, je na vsaki poti le končno mnogo odsekov.

Primer: množenje s seštevanjem



V praksi je tako formalno dokazovanje pravilnosti zelo težko.

3 Poraba časa in prostora

3.1 Poraba časa

Naj bo T(s) čas za izvedbo stavka S. Osnovna lastnost, ki je predpostavljena:

$$T(S_1, S_2) = T(S_1) + T(S_2)$$

Velja za računske modele, kjer se S_2 začne šele, ko se je izvedel S_1 (model zaporednega računanja), torej velja tudi za RAM. V današnjih računalnikih pa imamo več procesorjev, ki včasih lahko izvedejo S_1 in S_2 vzporedno. Tam naša predpostavljena osnovna lastnost ne velja vedno.

3.2 Poraba prostora

Pomnilni prostor sestavljajo razni deli:

- prostor za program
- prostor za spremenljivke
- prostor za sklad
- prostor za dinamično zasedanje/sproščanje

Naj bo M(X) prostor za spremenljivko X. Osnovna lastnost:

$$M(X_1 \cup S_2) = M(X_1) + M(X_2)$$

Velja za model računalnika, ki ne prireja istega pomnilnika različnim spremenljivkam. Ponovno je osnovna lastnost nekoliko pesimistična, vendar nam za analizo programov zadošca.

3.3 Asimptotična rast funkcij

Funkcija f(n) je po velikostnem redu asimptotično:

- **omejena navzgor**: z g(n), če $\exists\ c,\ u_0 > 0,\ \forall\ n > n_0 :\ f(n) \leq cg(n)$, to zapišemo kot $f(n) = \Theta(g(n))$
- **omejena navzdol** s $h(n)$, če $\exists\ d, n_1 > 0$: $\forall\ n_1 :\ f(n) \geq dh(n)$. To zapišemo kot $f(n) = \Omega(h(n))$.
- **enaka funkciji** $k(n)$ (asimptotično enaka!), če $f(n) = \Theta(k(n)) \wedge f(n) = \Omega(k(n))$, krajše: $f(n) = \Theta(k(n))$

Opomba: $\Theta(g(n))$ je množica vseh funkcij, ki so asimptotično omejene navzgor z $g(n)$, $f(n) \in \Theta(g(n))$.

3.4 Ocenjevanje porabe časa

Običajne poenostavitve:

- npr. vsi stavki porabijo enako mnogo časa (uniformna cena): $T(S_1) = T(S_2) = T(S_3) = \ldots = 1$
- npr. nekateri stavki rabijo 1, drugi pa 0 (štejemo le nekatere stavke)

Primer: Dvojiško iskanje
Denimo, da štejemo le število izvedb stavka IF.

```
1  while l <= r
2  {
3          m = (l + r) / 2;
4          if(a[m] > x)
5          {
6                  r = m - 1;
7          }
8          else
9          {
10                 l = m + 1;
11         }
12 }
```

Jedro zanke WHILE se izvede $\lceil logn \rceil$-krat. Čas za izvedbo algoritma je $\Theta(logn)$.

Zanke se v praksi pogosto pojavljajo, zato se pri ocenjevanju časa pojavljajo vrste:

$$\sum_{i=1}^{n} T_i$$

Pri čemer je:

- n - število ponovitev jedra zanke
- T_i - čas, ki ga rabi jedro zanke

Nekatere vrste:

- aritmetična vrsta:

$$\begin{aligned} \textstyle\sum_{i=1}^{n} i &= 1 + 2 + 3 + ... + n = \\ &= \tfrac{1}{2}(n+1)n = \\ &= \Theta(n^2) \end{aligned} \tag{1}$$

- geometrijska vrsta:

$$\begin{aligned} \textstyle\sum_{i=1}^{n} x^i &= 1 + x + x^2 + x^3 + ... + x^n = \\ &= \begin{cases} n+1, \text{ če je } x = 1 \\ \frac{n^{n+1}-1}{x-1}, \text{ sicer} \end{cases} \end{aligned} \tag{2}$$

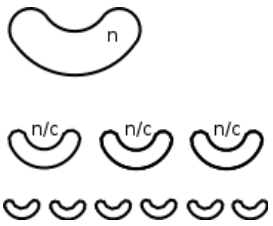
- harmonična vrsta:

$$\begin{array}{ll} \sum_{i=1}^{n} \frac{1}{i} & = 1 + \frac{1}{2} + \frac{1}{3} + ... + + \frac{1}{n} = \\ & = logn + \Theta(1) \end{array} \tag{3}$$

V praksi so pogoste tudi rekurzivne definicije algoritmov, torej potrebujemo tudi rekurzivne enačbe za računanje časovne zahtevnosti takih algoritmov.

Rekurzivno rešimo problem velikosti n:

1. tvorimo c podproblemov enake vrste in velikosti $\frac{n}{c}$
2. a ($\leq c$) od teh problemov rešimo
3. rešitve teh problemov uporabimo, da sestavimo rešitev večjega problema
4. če je problem dovolj majhen (trivialen, npr. n=1), porabimo zanj konstanten čas (rešimo ga na drug način, ne rekurziven)



$$T(n) = \begin{cases} aT(\frac{n}{c}) + bn^r, \text{ če je } n > 1 \\ b, \text{ sicer } (n = 1) \end{cases}$$

Če velja $a, b, r \geq 0$ in $c > 0$, potem:

$$T(n) = \begin{cases} \Theta(n^r), \ a < c^r \\ \Theta(n^r logn), \ a = c^r \\ \Theta(n^{log_c a}), \ a > c^r \end{cases} \tag{4}$$

Del III
Urejanje

4 Urejanje in razvrstitev metod za urejanje

Pomen: Če podatke uredimo jih hitreje najdemo.

Naloga: Dani so podatki $a_1, a_2, a_3, \ldots, a_n$.

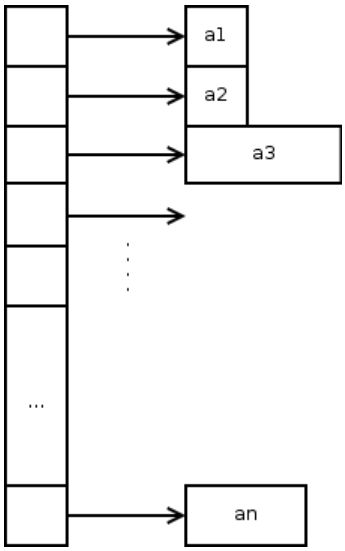
Cilj: Poiskati razporeditev $a_1, a_2, \ldots, a_n$, da bo veljalo $a_1 \leq a_2 \leq \ldots \leq a_{n-1} \leq a_n$.
Urejanje lahko razlikujemo po:

- vrsti podatkov
- relaciji
- številu podatkov
- algoritmu

Vrsta podatkov:

1. a_i so iste dolžine: | a_1 | a_2 | $\ldots$ | a_n |
2. a_i so različnih dolžin (uporabimo kazalce): | a_1 | a_2 | a_3 | a_4 | $\ldots$ |

Urejanje tukaj pomeni premeščanje (enako dolgih) kazalcev.



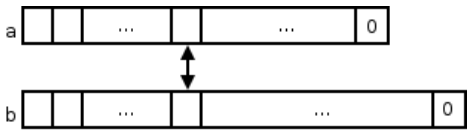
Slika 4: Polje kazalcev

- velik del programa je neodvisen od vrste podatkov $a_i \Rightarrow$ program je enostavnejši
- večja poraba pomnilnika (za tabelo kazalcev)
- manjša hitrost (ker je pri primerjavi a_i in a_j oba treba doseči preko kazalcev)

Relacija $\prec$

- definiramo na več načinov (procedura *FCmpType* v modulu *sort*)
- definiramo podatka na podlagi katerega se izvaja primerjanje - ključ

- v praksi je ključ:
  - število ⇒ relacija $\preceq$ je navadno $\leq$ primerjanje zahteva konstanten čas
  - znakovno zaporedje ⇒ relacija $\leq$ je leksikografsko primerjanje, relacija $\leq$ je lahko že dana (npr. Oberon - 2) ali pa jo je treba implementirati



Slika 5: Primerjanje različno dolgih tabel

Koda

Listing 3: Primer funkcije za primerjanja nizov v jeziku C

```
1 int CompareStr(char[] a, char[] b)
2 {
3         int i = 0;
4         while(a[i] == b[i] && a[i] != 0)
5         {
6                 i++;
7         }
8         return (int)(a[i] - b[i]);
9 }
```

Primerjanje rabi spremenljiv čas(tudi če sta polji enaki).

Število podatkov:

- "malo"⇒ vsi podatki so v notranjem pomnilniku v tabeli(array), takšno urejanje imenujemo notranje urejanje(internal)
- "veliko"⇒ vsi podatki so na zunanjem pomnilniku v datoteki(file) ⇒ zunanje urejanje(external)
  - zaporeden dostop do podatkov s pomikanjem datotečnega okna
  - vidni so le podatki, ki so v oknu (ti so tudi v notranjem pomnilniku)
  - včasih bomo datoteki rekli trak, da se povdari zaporedni dostop (kot npr. magnetni trak)

4.1 Algoritem

- "navadni"⇒ preprosti in relativno počasni
- "izboljšani"⇒ bolj zapleteni in relativno hitrejši

V knjigi so algoritmi napisani v Oberonu.

Algoritmi za urejanje, ki so si podobni po kakšni lastnosti imajo te lastnosti izpostavljene v ustreznem modulu. Najspološnejši nalogi ustreza modul:

```
1  MODULE Sort
2  CONST
3            eq*=0; less*=1; grt*=2;
4  TYPE
5            item*=RECORD(*podatkovni element*) END .... ai
6            PItem*=POINTER TO item;
7            FCmpType*=PROCEDURE(p,q:PItem;r:INTEGER):BOOLEAN;
8  END Sort;
```

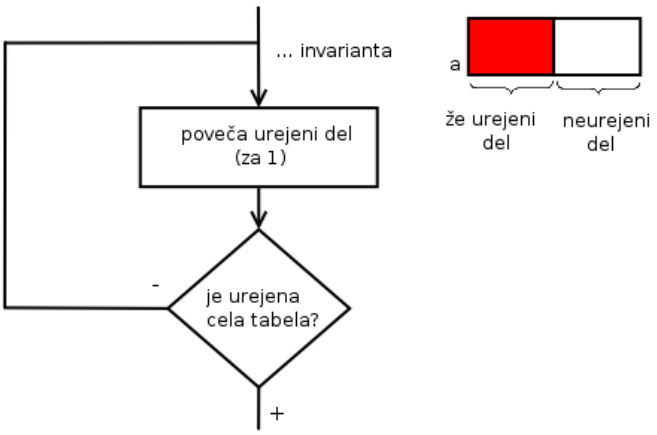
S specializacijo in dedovanjem dobimo module za bolj konkretne naloge:

```
1  MODULE IntSort
2  IMPORT S:=Sort;
3  TYPE
4           PItem*=POINTER TO Item;
5           Item*=RECORD(S.Item) k*:LONGINT END;
6  END IntSort
7
8  MODULE ArrSort
9  IMPORT I:=IntSort; S:=Sort;
10 TYPE
11         APItem*=ARRAY OF I.PIItem;
12         PAPItem*=POINTER TO APItem;
13 END ArrSort;
```

5 Notranje urejanje

Algoritmi za notranje urejanje

- *Navadni:*
  - potrebuje $\Theta(n^2)$ operacij za ureditev tabele velikosti n
  - poleg tabel ne rabijo bistveno več prostora
  - po pristopu:
    - navadno vstavljanje (insetion sort)
    - navadno izbiranje (selection sort)
    - navadne zamenjave (bubble sort)
  - imajo enako zgradbo (glej sliko 6)
  - razlikujejo se le po te, kako razširijo urejeni del
- *Izboljšani:*
  - potrebujejo $\Theta(nlogn)$



Slika 6: Diagram navadnega urejanja

- po pristopu:
  - Shellovo urejanje (izboljšano vstavljanje)
  - urejanje s kopico (izboljšano izbiranje) (Heapsort)
  - urejanje s porazdelitvami (izboljšane zamenjave) (Quicksort)

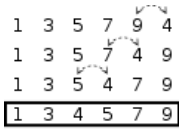
5.1 Navadno vstavljanje

Primer

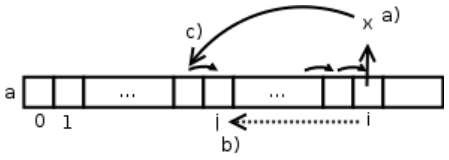
Slika 7: Primer vhodne tabele pri navadnem vstavljanju

1. zapomnimo si 4
2. vsi ki so v urejenem delu večji od 4 naj se (eden po eden) premaknejo v desno za eno mesto
3. vstavimo 4 za prvega, ki se ni premaknil

Splošna situacija



Slika 8: Primer urejanja z navadnim vstavljanjem



Slika 9: Prikaz splošne situacije

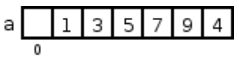
Koda Primer metode za algoritem navadnega vstavljanja:

```
1  void InsertionSort(Integer[] a)
2  {
3      int i, j, x;
4      int n = a.length();
5      for(i=1; i <= n-1; i++)
6      {
7          j = i;
8          x = a[j];
9          while((j >= 1) && (x < a[j-1]))
10         {
11             a[j] = a[j-1];
12             j--;
13         }
14     }
15 }
```

5.1.1 Uporaba čuvaja

Tabela, ki jo hočemo urediti naj bo sedaj $a[1...n]$, a[i] si bomo zapomnili v a[0] (namesto v x), to je ti. čuvaj, da se WHILE zanka ustavi (namesto pogoja $j >= 1$)

Primer



5.1.2 Psevdokoda

```
1  void SentinelSort(Integer[] a)
2  {
```

```
3       int i;
4       Integer n = a.length() - 1;    //toliko je elementov, ki jih urejamo
5
6       for(int i=2; i <= n; i++)
7       {
8           j = i;
9           a[0] = a[j];
10          while(a[0] < a[j-1])
11          {
12              a[j] = a[j-1];
13              j--;
14          }
15          a[j] = a[0];
16      }
17  }
```

5.1.3 Analiza časa

Čas za ureditev polja $a[1...n]$ je:

$$\sum_{i=2}^{n} \underbrace{\text{čas za povečanje urejenega dela } a[1...i-1]}_{\substack{\text{sorazmeren s številom } C_i \\ \text{operacij primerjanja v WHILE}}} = \\ = \sum_{i=2}^{n} C_i \tag{5}$$

C_i je različen, odvisen od podatkov:

- $C_{min,i} = 1$ ...kadar je a[i] že večji od vseh v a[1...]
- $C_{max,i} = i - 1$ ...kadar je a[i] manjši od vseh v a[1...i-1]
- $C_{avg,i} = \frac{1}{2}[c_{min,i} + c_{max,i}]$

Od tod dobimo 3 različne ocene za število vseh primerjav:

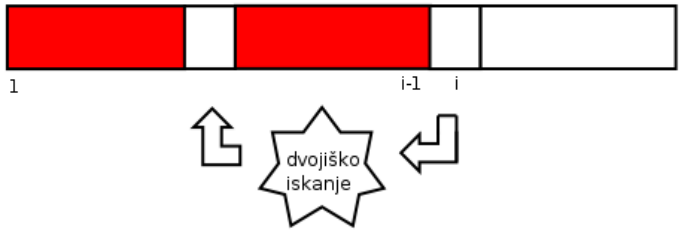
$$\begin{aligned}
C_{min} &= \sum_{i=2}^{n} c_{min,i} = \\
&= \sum_{i=2}^{n} 1 = \\
&= n - 1 = \\
&= \Theta(n) \\
C_{max} &= \sum_{i=2}^{n} c_{max,i} = \\
&= \sum_{i=2}^{n} (i-1) = \\
&= \frac{1}{2}(n-1)n = \\
&= \Theta(n^2) \\
C_{ave} &= \sum_{i=2}^{n} C_{ave,i} = \\
&= \sum_{i=2}^{n} (\frac{1}{2}[c_{min,i} + c_{max,i}] = \\
&= \frac{1}{2} \sum_{1=2}^{n} [1 + i - 1] = \\
&= ... = \\
&= \frac{1}{4}(n^2 + n - 2) = \\
&= \frac{1}{4}(n-1)(n-2) = \\
&= \Theta(n^2)
\end{aligned}$$

Povzetek:

- $c_{min} = \Theta(n)$
- $c_{max} = \Theta(n^2)$ tole nam ni
- $c_{avg} = \Theta(n^2)$ vseč!

Ali lahko algoritem izboljšamo?

Zamisel: Ker je $a[1...i-1]$ že urejen, lahko najdemo pravo mesto za $a[i]$ z dvojiškim iskanjem.



$$n! = \sqrt{2\pi n}(\frac{n}{e})^n \tag{6}$$

Število primerjanj je: $C_i = \lceil log_2 i \rceil$, zato je št vseh primerjanj:

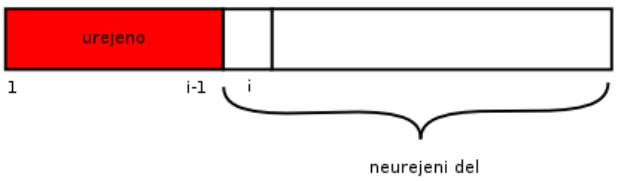
$$\begin{aligned} C &= \sum_{i=2}^{n} \lceil log_2 i \rceil \leq \sum_{i=2}^{n} (log_2 i + 1) = \\ &= log n! + n = \\ &= log[\sqrt{2\pi n}(\tfrac{n}{e})^n] + n = \\ &= log\sqrt{2\pi n} + n log\tfrac{n}{e} + n = \\ &= log\sqrt{2\pi n} + n log n - n log e + n = \\ &= \Theta(n log n) \end{aligned}$$

Toda: Čeprav hitro najdemo pravo mesto za a[i], je še vedno treba vse elemente nad $a[i]$ premakniti v desno. To zahteva $\Theta(i)$, zato bo algoritem v celoti še vedno reda $\Theta(n^2)$

Σ: Pri analizi je treba paziti, katere operacije zanemarimo, še posebej, če nastopajo v zanki

5.2 Navadno izbiranje (Selection Sort)

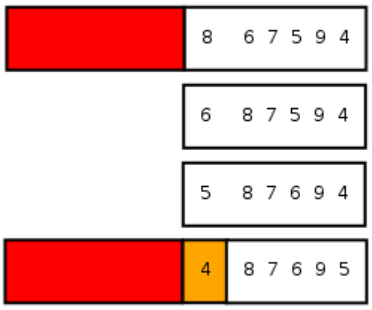
Zamisel:



Med neurejenimi poišči najmanjšega in ga postavi na i-to mesto.

Izvedba: Sprehodi se po $a[j], j = i+1, i+2, ..., n-1$ in če je $a[j]$ manjši pd trenutno najmanjšega (ta je v $a[i]$), jih zamenjaj!

Primer:



Koda: Primer metode za algoritem navadnega izbiranja v Javi:

```
1  void SelectionSort(Object[] a)
2  {
3          long i, j, n;
4          n = a.length();
5          for(int i=0; i <= n-2; i++)
6          {
7                  for(int j=i+1; j <= n-1; j++)
8                  {
9                          if(a[i] > a[j])
10                         {
11                                 Exch(a[i], a[j]);
12                         }
13                 }
14         }
15 }
```

5.2.1 Analiza časa

Število primerjanj:

$$\begin{aligned} C &= \sum_{i=0}^{n-2} C_i = \\ &= /a[i] \text{ primerjav z vsemi } a[i+1, i+2, ..., n-1]/ = \\ &= \sum_{i=0}^{n-2} (n-1-i) = \\ &= \frac{1}{2}(n-1)n = \\ &= \Theta(n^2) \end{aligned}$$

C je neodvisen od podatkov, ker se $a[i]$ primerja z vsemi.

Kaj pa število zamenjav?

Naj bo M_i število zamenjav z $a[i]$:

- $M_{min,i} = 0$ (ko je $a[i]$ že najmanjši med neurejenimi)
- $M_{max,i} = n - 1 - i$ (ko je $a[i]$ največji med neurejenimi)

Od tod sledi:

$$M_{min} = \sum_{i=0}^{n-2} M_{min,i} = 0$$
$$M_{max} = \sum_{i=0}^{n-2} M_{max,i} = \sum(n - 1 - i) = \frac{1}{2}(n-1)n = \Theta(n^2)$$

5.3 Navadne zamenjave (Bubble Sort)

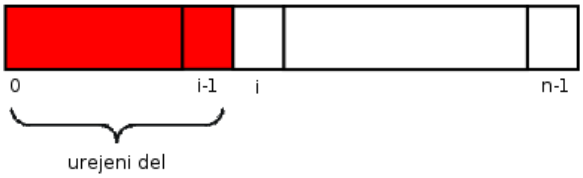
Primer:

8	6	7	4	9	5
8	6	7	4	5	9
8	6	4	7	5	9
8	4	6	7	5	9
4	8	6	7	5	9

1. sprehod: Sprehodimo se od desne v levo po neurejenem delu pri vsakem koraku primerjamo soseda in ju zamenjamo, če je potrebno.
Po 1. sprehodu je $a[0]$ urejen del tabele in vsebuje najanjši element celotne tabele.

Splošno:

- **predpostavka:** po i-tem sprehodu je a[0...i-1] urejen del in vsebuje i najmanjših elementov (celotne) tabele



- sedaj se sprehodimo po neurejenem delu, od desne v levo, po indeksih $j = n - 1, ..., i$, pri čemer $a[j-1]$ in $a[j]$ zamenjamo, če je treba
- ko se sprehod konča je v a[i] najmanjši element neurejenega dela
- $a[i]$ je večji (ali enak) od vseh elementov v urejenem delu
- **posledica:** $a[0...i]$ urejen del tabele, vsebuje i+1 najmanj,ših elementov celotne tabele (zančna invarianta)
  če to ponavljamo od $i = 1$ do $n - 1$ bo tabela urejena

Koda:

Listing 4: BubbleSort

```
1  void BubbleSort(Comparable[] a)
2  {
3      int n = a.length;
4      for(int i = 1; i < n; i++)
5      {
6          for(int j = n-1; j >= i; j--)
7          {
8              if(a[j-1].CompareTo(a[j]) == 1)
9              {
10                 Zamenjaj(a[j-1], a[j]);
11             }
12         }
13     }
14 }
```

Analiza

V 1. sprehodu je n-1 primerjanj.

V 2. sprehodu je n-2 primerjanj.

...

V n-1 sprehodu je 1 primerjanje.

Vseh primerjanj je : $1 + 2 + ... + n - 1 = (n-1)n = \Theta(n^2)$
Poskusimo še izboljšati algoritem sortiranja.

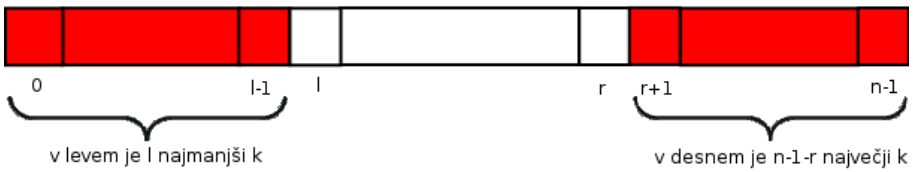
5.4 Izmenične zamenjave (shaker sort)

Opazimo: Naj bo najmanjši element v neurejenem delu tabele, kjer kaže puščica.

Ko ga sprehod v levo doseže, ga zagrabi in odnes s seboj do levega roba.

Kaj pa se dogaja z največjim elementom neurejenega dela?
Vsak sprehod gre čezenj, ob vsakem sprehodu se pomika v desno.

Zamisel: Uvedemo še sprehode v desno, sprehod v desno bo zagrabil največjega v neurejenem delu tabele in ga premaknil na desni rob. V vsakem trenutku bosta **dva** urejena dela tabele:



Izmenično se sprehajamo po neurejenem delu $a[l..r]$ v levo in desno in potisnemo najmanjši oz. največji element na levi oz. na desni rob (najmanjši pristane v a[l], največji pa v a[r] $\Rightarrow$ levi rob se pomakne v desno za 1, desni pa v levo za 1.

Še ena zamisel: Če se pri sprehodu v levo zadnja zamenjava zgodi med $a[i-1]$ in $a[i]$, so po sprehodu očitno $a[0...i]$ urejeni, zato lahko levo rob preskoči na $i+1$. podobno tudi pri desnemu sprehodu vse skupaj končamo, ko postane $l > r$.

Listing 5: Algoritem za sortiranje z izmeničnimi zamenjavami

```
1  void ShakerSort(Comparable[] a)
2  {
3      int i, j, n, l, r;
4      n = a.length;
5      l = 1;
6      r = n-1;
7      i = n-1;
8      do
9      {
10         for(j = r; j >= l; j--)
11         {
12             if(a[j-1].CompareTo(a[j]) == 1)
13             {
14                 Zamenjaj(a[j-1],a[j]);
15                 i = j;
16             }
17         }
18         l = i + 1;
19         for(j = l; j <= r; j++)
20         {
21             if(a[j-1].CompareTo(a[j]) == 1)
22             {
23                 Zamenjaj(a[j-1], a[j]);
24                 i = j;
25             }
26         }
27         r = i-1;
28     }
29     while(l > r);
```

```
30 }
```

Pričakujemo:

- pričakujemo boljši povprečni čas zaradi druge zamisli
- v najslabšem primeru še vedno $\Theta(n^2)$

Navadni algoritmi za notranje sortiranje:

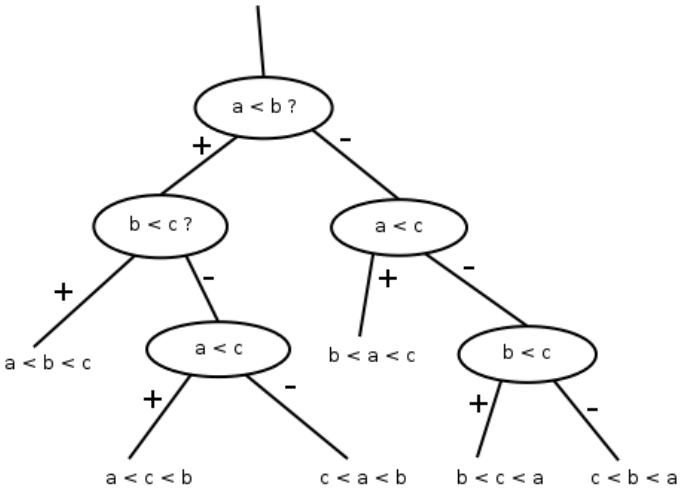
- navadno vstavljanje
- navadno izbiranje
- navadne zamenjave

Vsi zahtevajo $\Theta(n^2)$ časa (merjeno v operacijah primerjanja ali zamenjav).

Ali se da zabele urediti hitreje, npr. v času $\Theta(n)$ ali $\Theta(\sqrt{n})$ ali $\Theta(logn)$?
Ocenimo, najmanj koliko je potrebnih operacij primerjanja.

Kako urediti n.danih št., če je na voljo le operacija primerjanja?

Primer: Dana so št. a, b in c, ki so različna.
Naloga: Kako so urejena po vrsti?
Uporabimo algoritem, ki ga predstavlja naslednji diagram poteka. Diagram



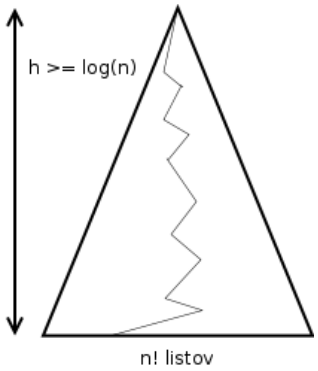
Slika 10: Diagram poteka

poteka je dvojiško drevo, ki ima v notranjih vozliščih primerjanja, v listih pa vse možne ugotovitve (vse možne ureditve št. a,b in c v permutacije).

Splošno: Če so dana št. a1, a2, a3,... an, lahko njihovo urejenost ugotovimo z vsakim diagramom poteka, ki ima obliko dvojiškega drevesa, ki ima v notranjih vozliščih primerjanja, v listih pa vse možne permutacije števil a1, a2,..., an.

Če med izvajanjem algoritma pridemo do vsakega lista, izvemo kako so urejeni vhodni a1, a2,..., an.

Opisano drevo ima n! listov (po en list za vsako permutacijo št. a1, a2,..., an). Vsako dvojiško drevo z N listi ima višino $\geq log2N$. Opisano drevo ima višino $\geq log2N!$ Pri urejanju n elementov $a_1, a_2, ...a_n$ bo potrebnih $\geq log_2 n!$



Slika 11: Višina drevesa

primerjanj.

Število primerjanj:

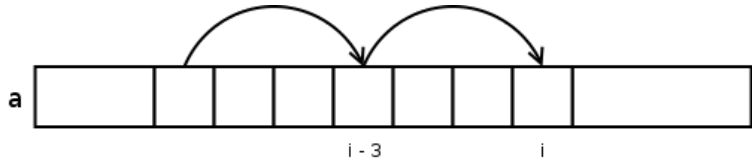
$$\begin{aligned}
\text{št. primerjanj} \quad & \geq log_2 n! \doteq \\
& = / * \text{po Stirlingu } n! \doteq \sqrt{2\Pi n}\left(\tfrac{n}{e}\right)^n * / = \\
& = log\sqrt{2\Pi n}\left(\tfrac{n}{e}\right)^n = \\
& = ... = \\
& = nlogn - \tfrac{1}{2}logn - 1.44n \geq \\
& \geq cnlogn = \\
& = \Theta(nlogn)
\end{aligned}$$

Vsak algoritem za urejanje n števil zahteva vsaj *cnlogn* operacij primerjanja (za ureditev vsaj enega vhoda). Torej je smiselno iskati algoritme, ki rabijo $\Theta(nlogn)$ časa!

5.5 Shellovo urejanje (izboljšano vstavljanje)

Motivacija Pri navadnem vstavljanju je a[i] primerjanj z vsemi, ki so v urejenem delu in so večji od njega (pri tem so vsi ki so večji od njega pomikajo v desno).

Shellova zamisel a[i] naj se primerja z elementi na levi po vrsti ampak z vsakim h-tim (npr. h=3). Sedaj se ti pomikajo desno toda za h mest, če so večji od a[i], a[i] pa gre na mesto tistega, ki se je prestavil zadnji.



Tako da bo a[i] hitreje (z manj primerjanji) prišel v levo.

Ali se urejenost urejenega (levega) dela ne poruši, ko nekateri elementi preskakujejo druge?

Odgovor Ne, če ne zahtevamo več, da je osenčen del v celoti urejen, temveč, da je sestavljen iz več podtabel od katerih je vsaka zase urejena.

Npr.: podtabelo a sestavljajo elementi:

A	B	C	A	B	C	A	B	C	A	B	C

Shellov algoritem Neurejeno tabelo razdeli na več porazdeljenih podtabel (A, B, C). Vsako podtabelo uredi z navadnim vstavljanjem. Ko so vse podtabele urejene (vsaka zase), vse skupaj ponovi z manjšim številom podtabel. Algoritem se konča , ko je št podtabel enako 1.

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
7	2	9	1	16	4	15	5	11	6	3	8	10	12	14	13

h1 = 8 => podtabele (a1, a9), (a2, a10), (a3, a11), (a4, a12), (a5, a13), (a6, a14), (a7, a15), (a8, a16)

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
7	2	3	1	10	4	14	5	11	6	9	8	16	12	15	13

h2 = 4 => podtabele (a1, a5, a9, a13), (a2, a6, a10, a14), (a3, a7, a11, a15), (a4, a8, a12, a16)

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
7	2	3	1	10	4	9	5	11	6	14	8	16	12	15	13

h3 = 2 => podtabele (a1, a3, a5, a7, a9, a11, a13, a15), (a2, a4, a6, a8, a10, a12, a14, a16)

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
3	1	7	2	9	4	10	5	11	6	14	8	15	12	16	13

h4 = 1 => podtabela je ena sama = cela tabela

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

Primer:

Vprašanje: Kakšno naj bo zaporedje?
V prejšnjem primeru:

$$h_1 = 8$$
$$h_2 = 4$$
$$h_3 = 2$$
$$h_4 = 1, \text{ torej } t = 4$$

Odgovor: Važno je, da za zaporedje velja:

$$h_1 > h_2 > h_3 > ... > h_{t-1} > h_t = 1.$$

Toda nekatera zaporedja so bolj ugodna od drugih (dokazano z eksperimenti), npr.:

1. ...121 40 13 4 1... $h_{n-1} = 3h_n + 1, h_t = 1$
2. ...31 15 7 3 1... $h_{n-1} = 2h_n + 1, h_t = 1$

Kateri od njih naj bo začetni h (h_1) oz. koliko je t?

1. $t = \lfloor log_3 n \rfloor - 1$
2. $t = \lfloor log_2 n \rfloor - 1$

Analiza: Glej knjigo Knuth D.: TAOCP [2]
Shellovo urejanje ima pri zaporedju:

1. časovno zahtevnost $\Theta(n^{1.5})$
2. časovno zahtevnost $\Theta(n^{1.26})$

Pozor: $n^{1.26}$ je še vedno asimptotično gledano večja funkcija kot $nlogn$.

5.6 Urejanje s kopico

5.7 Motivacija

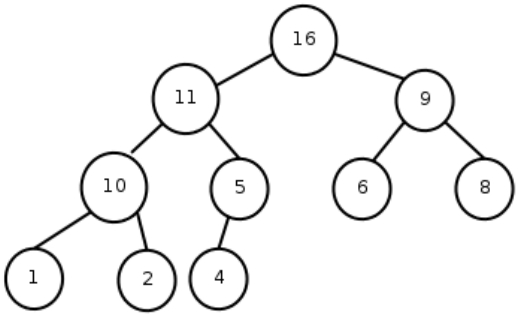
Pri pregledu neurejenega dela bi bilo treba nabrati več koristnih informacij in jih uporabit. Navadna urejanja "nimajo spomina".

Kako? Neurejeni elementi naj bodo v kopici.

5.8 Kopica (heap)

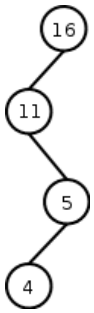
- je dvojiško drevo (vozlišča hranijo elemente a[1], a[2], ... ,a[n] oz. njihove ključe)
- ki je urejeno da za vsako vozišče velja, da je večje od vseh elementov v njegovih dveh poddrevesih (če obstajata)
- je levo uravnoteženo vse veje drevesa so dolge d ali d-1 (za nek d) in vse daljše veje do zbrane na levi strani

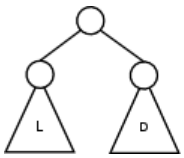
Primer:



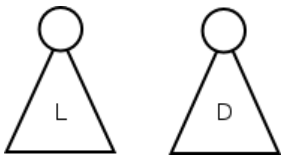
Slika 12: Primer kopice

Opazimo:

- ključi vzdolž poljubne veje so urejeni, npr: 

- največji ključ je v korenu drevesa
- če je drevo 

kopica, sta kopici tudi levo in desno poddrevo



Slika 13: Levo in desno poddrevo

5.8.1 Uporaba pri urejanju

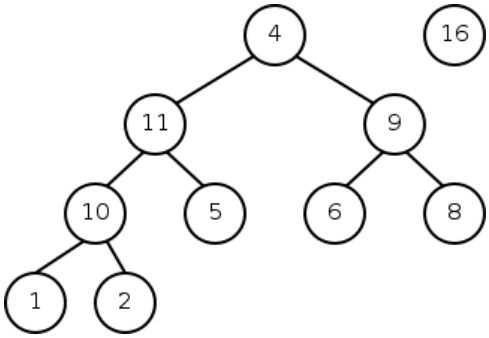
1. izloči koren in ga nekam shrani
2. namesto njega postavi zadnji list

3. popravi dobljeno drevo v kopico (koren se pogrezne tja kamor sodi s tem da se zamenjuje z večjim od sinov)

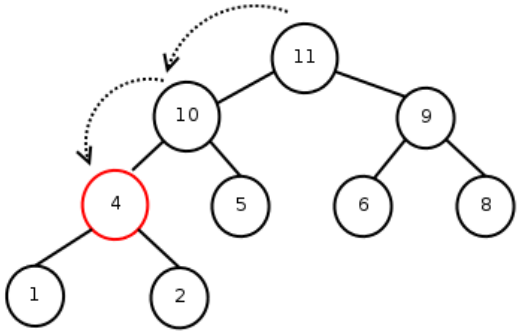
Izločeni koreni so urejeni po velikosti.

Primer

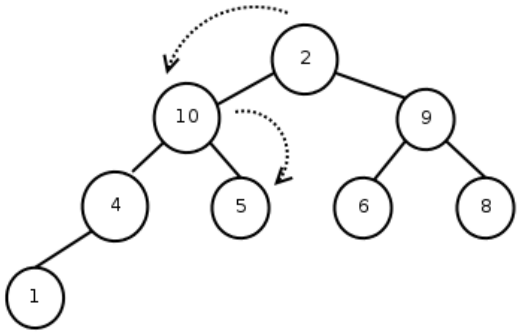
po 1)... do 2):



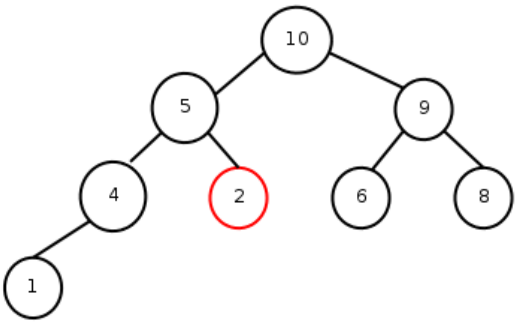
pogrezanje 4:



ponovimo 1)...2) (izločimo 11):



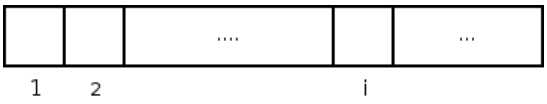
in popravimo kopico:



Popravljamo dokler ne izpraznimo kopice.

Podatkovna struktura:

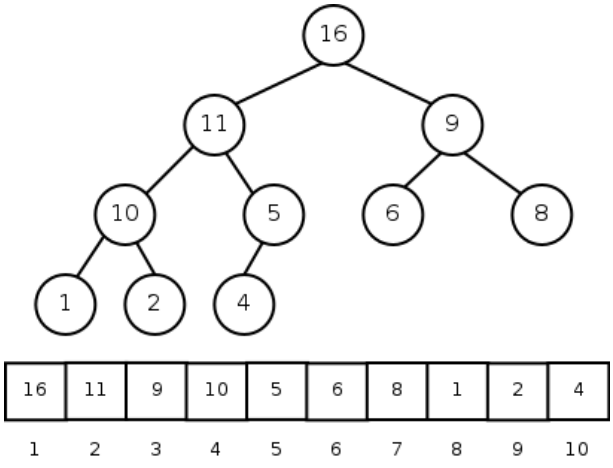
- kopico realiziramo s tabelo a



  kjer:

  - koren kopice je v a[1]
  - če je neko vozlišče kopice v a[i], sta levi in desni sin (če obstajata) shranjena v a[2i] in a[2i + 1]

Primer

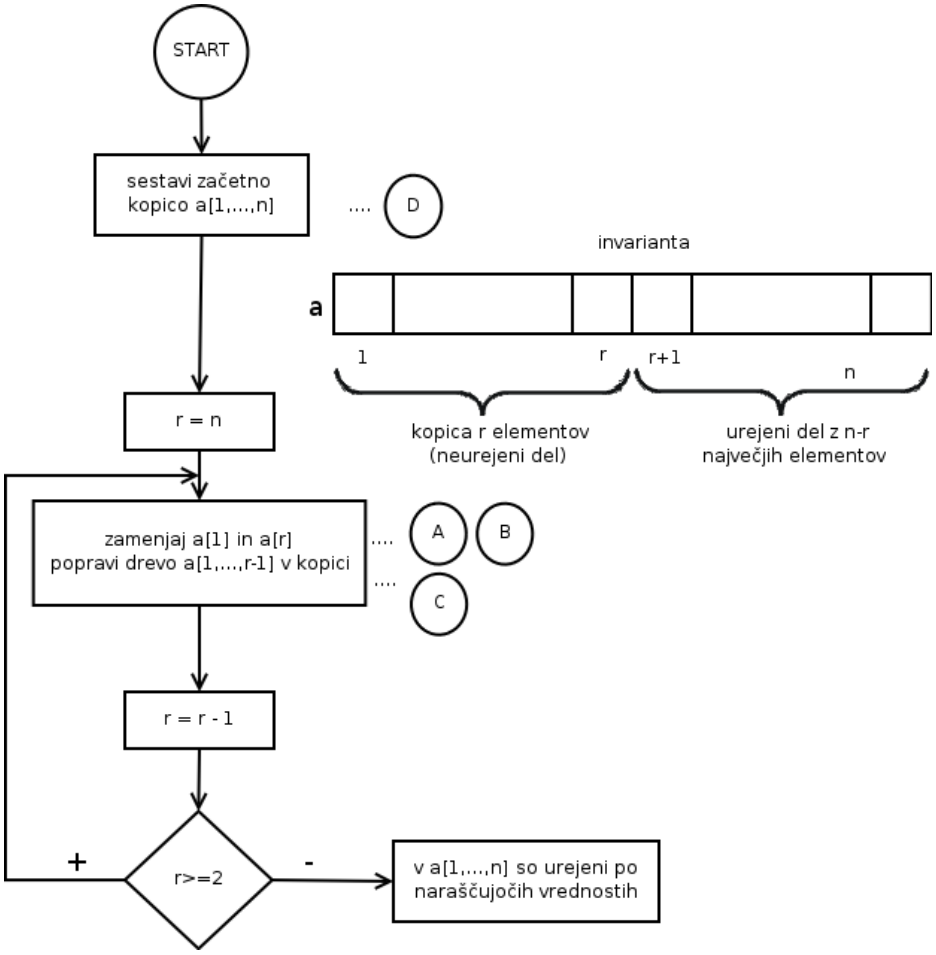


Torej:

- če je $s > 1$, je $a[s]$ sin nekoga
- njegov oče je v celici $a[s/2]$

- s sodo št. $\Rightarrow a[s]$ je levi sin
- z liho št. $\Rightarrow a[s]$ je desni sin

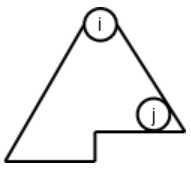
Urejanje:



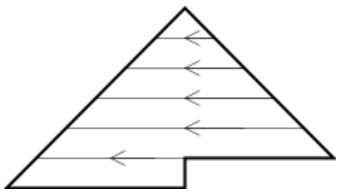
Urejanje poteka na mestu (in situ), kar je ugodno.

Kako izvesti korak C in D?

Denimo, da bi imeli na voljo proceduro PopraviVKopico(i,j), ki bi znala drevo popraviti v kopico ob predpostavki, da sta obe poddrevesi vozlišča že kopici.



Tedaj bi bil C kar klic PopraviVKopico(1,r-1).
Sestavljanje začetne kopice bi začeli od zadnjega očeta (ker so listi že kopice) po nivojih navzgor do korena, pri čemer pri vsakem vozlišču pokličemo proceduro PopraviVKopico.

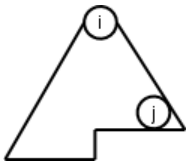


Torej je D:

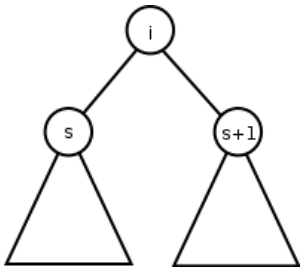
```
1  void SestaviZacetnoKopico()
2  {
3          int i;
4          for(i = n/2; i > 0; i--)
5          {
6                  PopraviKopico(i, n);
7          }
8  }
```

Kakšna je PopraviVKopico(i,j)?

- poglejmo poddrevo



- če je i list, je že (trivialna) kopica
- če ni list, ima vsaj enega sina: vozlišče $s = 2i$, če je $s + 1 \leq j$, pa tudi desnega
- predpostavimo, da sta L in D obe kopici



- pozornost usmerimo na večjega od sinov (denimo da bo to sin z indeksom s)

- poddrevo, kjer je sedaj oče, je morda treba spet popraviti v kopico, zato zopet pokličemo proceduro PopraviVKopico (rekurzija)

```
1  void PopraviVKopico(int i, int j)
2  {
3     int s;
4     if (i <= j/2) .... i ima sina
5     {
6        s = 2*i;  //s je levi
7        if ((s+1) <= j)   //ce obstaja desni sin
8        {
9           if (a[s] < a[s+1])
10          {
11             s++;         //naj s oznacuje vecjega
12          }
13       }
14       if (a[i] < a[s])
15       {
16          Zamenjaj(a[i], a[s])    //zamenjaj
17          PopraviVKopico(s, j)    /* popravi se drevo
18                                     kamor je prisel oce */
19       }
20    }
21 }
```

5.8.2 Celoten algoritem za heapsort

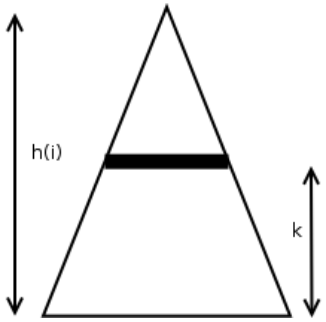
```
1  void HeapSort(int[] a)
2  {
3          int r;
4          SestaviZacetnoKopico();
5          for (int i=n; i >= 2; i--)
6          {
7                  Zamenjaj(a[i], a[r]);
8                  PopraviVKopico(1, r-1);
9          }
10 }
```

5.8.3 Analiza

- PopraviVKopico(i,j):
  - primerja in zamenjuje vozlišča vzdolž ene veje, zato rabi $T(i) = \Theta(k(i))$ operacij, kjer je k(i) višina vozlišča i, tj. $h(i) = log2$(št. elementov v poddrevesu vozlišča i)
- SestaviVKopico:
  - kliče PopraviVKopico v zanki, zato rabi

- velja: to je reda $\Theta(n)$
- dokaz: na višini k je kvečjemu $\lceil \frac{n}{2^k} \rceil$ vozlišč, zato:

$$\begin{aligned}
\textstyle\sum_{i=\lfloor \frac{n}{2} \rfloor}^{2} &= \text{/seštevamo po višinah/} \\
&= \textstyle\sum_{k=1}^{h} (\frac{n}{2^k} k) = \\
&= n \underbrace{\textstyle\sum_{k=1}^{h} (\frac{k}{2^k})}_{} = \\
&\text{konvergira ko gre } h \to \infty \\
&= n \cdot \text{konst.} = \\
&= \Theta(n)
\end{aligned}$$



- Heapsort:
  - SestaviVKopico ..... $\Theta(n)$
  - zanka rabi:

$$\begin{aligned}
\textstyle\sum_{r=n}^{2} (1 + T(r-1)) &= \textstyle\sum_{r=1}^{n-1} (1 + T(r)) = \\
&\leq n - 1 + \textstyle\sum_{r=1}^{n-1} T(r) \\
&\leq n - 1 + \textstyle\sum_{r=1}^{n-1} log_2(r) = \\
&n - 1 + log_2(n-1)! = \\
&\text{/Stirling/} \\
&\leq n - 1 + log_2(\sqrt{2\Pi(n-1)}(\tfrac{n-1}{e})^{n-1}) \leq \\
&\leq ...nlogn = \\
&= \Theta(nlogn)
\end{aligned}$$

  - algoritem je asimptotično optimalen glede števila primerjanj

5.9 Urejenje s porazdelitvami (Quicksort)

Motivacija Pri metodi navadnih zamenjav se sprehajamo od leve proti desni in menjavamo sosednja elementa. Primer: a[1, 2, 99, 98,...81, 80, 3, 100]. Opazimo, da bo potrebnih bo veliko zamenjav, da bo 3 prišla na svoje mesto!

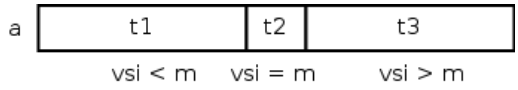
Zakaj kar takoj ne zamenjamo te 3 in 99?

Zamisel Možne naj bodo zamenjave med poljubno oddaljenimi elementi.

Kako možnost oddaljenih zamenjav čimbolj izkoristiti?

Zamisel

1. Tabelo z oddaljenimi zamenjavami na grobo uredimo glede na neko mejno vrednost m, tj. elemente tabele a porazdelimo (premečemo) v tri podtabele t1, t2, t3, kjer:



2. Potem na podoben način uredimo t1 in t3 (na mestu), npr. z rekurzivnim klicem iste procedure (rekurzivni razcep)

5.9.1 Koda

```
1  void Quicksort(t)
2  {
3          if(t.length==1)
4                  return t;
5          else
6          {
7                  m = izberi med elementi tabele t;
8                  porazdeli tabelo t v podtabele t1,
9                    t2 in t3 glede na m;
10                 return(Quicksort(t1); t2; Quicksort(t3));
11         }
12 }
```

Opombe Algoritem kliče sebe (rekurzivno) dvakrat (rekurziven razcep).

1. **Kako (hitro) izbrati m?**
   V konstantnem času ($\Theta(1)$) so možnosti:
   - npr. m = prvi element tabele (lahko slabo), ker je lahko $|t1| << |t3|$
   - m je srednji element v tabeli (kot npr. v knjigi prof. Vilfana [1])
   - m je naključno izbrani ($\Rightarrow$ uporabiti je treba generator naključnih števil $\Rightarrow$ izguba časa)
   - m = 1/3 (prvi, srednji, zadnji)

   Najbolje je, če sta t1 in t3 (približno) enako dolga, toda v času $\Theta(1)$ ne moremo najti m, ki bi to zagotovljal.

2. **Kako (hitro) porazdeliti tabelo t v podtabele t1, t2 in t3 (glede na m)?**

   (a) **Prva možnost:**
   najprej iz t[] sestavimo $[t1, t2 \cup t3]$, nato iz tega $[t1, t2, t3]$

(b) **Druga možnost:**
Ostanemo pri $[t1, t2 \cup t3]$ (tj. prihranimo si iskanje meje med t2 in t3), v glavnem programu pa kličemo:

```
return(Quicksort(t1); Quicksort(t2 u t3))
```

Zamisel: Z indeksoma i in j se sprehodimo od levega in desnega roba tabele, dokler ne najdemo dveh elementov, ki sodita v nasprotno podtabelo, elementa zamenjamo in nadaljujemo sprehod.

Listing 6: Primer metode, ki razdeli elemente v podtabele

```
1  void Partition(t, p, z, m)
2  {
3          int i, j;
4          i=p;
5          j=z;
6          while(i<j)
7          {
8                  while(t[i]<m && i<=z)
9                          i++;
10                 while(t[j]>=m && j>=p)
11                         j--;
12                 if(i<j)
13                 {
14                         zamenjaj(i, j);
15                         i++;
16                         j--;
17                 }
18         }
19 }
```

Opomba Ko se partition konča, je t1 v $t[p, ..., i-1]$ in $t2 \cup t3$ v $t[j+1, ..., z]$. $Partition(t, ...)$ zahteva $\Theta(|t|)$ operacij primerjanja in $\pm$.

Listing 7: Celoten algoritem za hitro sortiranje

```
1  void Quicksort(t)
2  {
3          if(t ima en element)
4          {
5                  return t;
6          }
7          else
8          {
```

```
 9                    m = izberi med elementi tabele t;
10                    porazdeli t na t1 in (t2 unija t3);
11                    return(Quicksort(t1); Quicksort(t2 u t3));
12            }
13    }
```

Naj bo $T(|t|)$ čas za ureditev tabele t, tedaj:

$$T(|t|) = T(|t1|) + T(|t2 \cup t3|) + \Theta(|t|) + \Theta(1) \tag{7}$$

Rešitev te enčbe je odvisna od tega, kako uravnoteženo je porazdeljevanje.

Najslabši čas Naj bo vsaka porazdelitev (na vsakem koraku algoritma) maksimalno neuravnotežena, tj. da razdelitev vrne:

$$t[t1, t2 \cup t3]$$

Enačba (7) se glasi:

$$T_{max}(n) = T_{max}(1) + T_{max}(n-1) + \Theta(n) + \Theta(1), \text{ oziroma}$$
$$T_{max}(n) = T_{max}(n-1) + \Theta(n)$$

Računamo:

$$\begin{aligned} Tmax(n) &= /*\, razvijemo \,*/ = \\ &= \Theta(1) + \Theta(2) + ... + \Theta(n) = \\ &= \textstyle\sum_{h=1}^{n}(\Theta(h)) = \\ &= \Theta(vsota[1,n](h)) = \\ &= \Theta(n^2) \end{aligned}$$

Najboljši čas Predpostavimo naj bo porazdelitev najbolj uravnotežena kar se le da (na vsakem koraku algoritma), tj. vrne porazdelitev na sve enako dolgi podtabeli:

$$t[t1, t2 \cup t3].$$

Enačba (7) je torej:

$$T_{min}(n) = T_{min}(n/2) + T_{min}(n/2) + \Theta(n) + \Theta(1) \text{ oziroma}$$

$$T_{min}(n) = 2 * T_{min}(n/2) + \Theta(n)$$

Ta enačba je oblike:

$$T(n) = a * T(n/c) + b * n^r = \begin{cases} \Theta(n^r); \text{ če } a < c^r \\ \Theta(n^r * log(n)); \text{ če } a = c^r \\ \Theta(n * log_c(a)); \text{ če } a > c^r \end{cases}$$

Pri nas:

$$a = c = 2r = 1$$

Rešitev: $T_{min}(n) = \Theta(n * log(n))$

Kaj pa v povprečju?

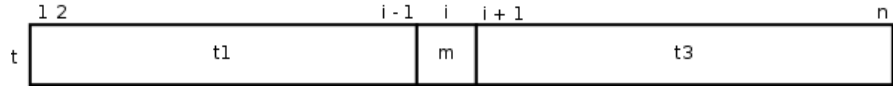
Povprečen čas Predpostavimo, da so vsi elementi tabele različni.
Posledica:

- tabela t_1 in t_3 karseda veliki (ker je v t_2 kvečjemu en element)
- s tem bo tudi povprečen čas karseda velik (pesimističen) $\Rightarrow$ analiza ne bo preveč optimistična

Naj bo $T_{avr}(n)$ povprečen čas za ureditev n elementov tabele t. Velja:

$$T_{avr}(0) = T_{avr}(1) = b = \Theta(1)$$

Naj bo m, ki ga izbiramo i-ti ($i = 1, 2, ...n$) najmanjši v tabeli, torej:



Enačba (7) pri takšnem m je:

$$T(n) = T(i-1) + T(n-i) + \Theta(n),$$

toda m je lahko katerikoli po vrsti, t. i je lahko katerikoli izmed 1, 2, 3,... n.

Predpostavka Vse vrednosti za i so enako verjetne.
Če upoštevamo vse vrednosti i in vzamemo povprečje časov pri vseh možnih m, dobimo rekurzivno enačbo za povprečni čas T_{ave}:

$$\begin{aligned} T_{ave}(n) &\leq \tfrac{1}{n}\textstyle\sum_{i=1}^{n}[T_{ave}(i-1) + T_{ave}(n-i)] + \Theta(n) = \\ &= \tfrac{1}{n}[T_{ave}(0) + T_{ave}(1) + ... + T_{ave}(n-1) + \\ &\quad T_{ave}(n-1) + T_{ave}(n-2) + ... + T_{ave}(1) + T_{ave}(0)] + \Theta(n) = \\ &= cn + \tfrac{2}{n}\textstyle\sum_{i=0}^{n-1} T_{ave}(i) \end{aligned}$$

Trditev

$T_{avr}(n) \leq 2(b+c) \cdot nlog(n)$ je čas za ureditev trivialnih tabel

Dokaz (indukcija po $n \geq 2$)

- $n = 2$:

$$T_{avr}(2) = c \cdot + \frac{1}{2}\sum_{i=0}^{1} T_{ave}(i) = 2c + b + b = 2(b+c)$$

- n-1 (indukcijska predpostavka):

$$T_{ave}(n-1) \leq 2(b+c)(n-1)\log(n-1)$$

- n:

$$\begin{aligned} T_{avr}(n) &\leq cn + \tfrac{2}{n}\textstyle\sum_{i=0}^{n-1} T_{ave}(i) = \\ &= cn + \tfrac{1}{2}(T_{ave}(0) + T_{ave}(1) + \textstyle\sum_{i=2}^{n-1} T_{ave}(i) = \\ &= /^* \text{ po indukcijski predpostavki } ^*/ = \\ &\leq cn + \tfrac{4b}{n} + \tfrac{2}{n} \cdot 2(b+c)\textstyle\sum_{i=2}^{n-1} i \ln i \leq \\ &\leq /^* \text{ ker velja } \textstyle\sum_{i=2}^{n-1} i \ln i \leq \int_2^n x \ln x \, dx \leq \tfrac{n^2 \ln n}{2} - \tfrac{n^2}{4} \, ^*/ \\ &\leq cn + \tfrac{4b}{n} + \tfrac{2}{n} \cdot 2(b+c)(\tfrac{n^2 \ln n}{2} - \tfrac{n^2}{4}) = \\ &= 2(b+c)n\ln n - bn - cn + cn + \tfrac{4b}{n} = \\ &\leq 2(b+c)n\log n \end{aligned}$$

Qed (kot je bilo dokazano).
Povprecni čas $T_{ave}(n)$ za ureditev tabele z n elementi je $\Theta(n \log n)$

5.9.2 Povzetek

Quicksort:

- najslabši čas: $\Theta(n^2)$
- najboljši čas: $\Theta(n \log n)$
- povprečni: $\Theta(n \log n)$
- v praksi se dobro obnaša
- v povprečju je teoreticno optimističen, toda v praksi je boljši od drugih

5.9.3 Poraba prostora in rekurzija

Ko t razdelimo na $t_1 + t_2 \cup t_3$ lahko nadaljujemo z urejanjem ene podtabele, npr. t_1, urejanje druge podtabele ($t_2 \cup t_3$) pa moramo odložiti dokler prva ni urejena. Podatke o "odloženem delu"lahko shranimo:

- implicitno:
  - z mehanizmom rekurzije
  - ta skrbi za shranitev podatkov (podproblema) do klica procedure
  - podrobnosti so pred programerjem skrite
  - sodobni jeziki omogočajo rekurzijo
- ali eksplicitno:
  - program sam skrbi za shranitev podatkov o odloženemu problemu na lastnem skladu
  - programer mora implementirati lasten sklad in procedure za delo z njim
  - algoritem ni več rekurziven
  - primer na str. 52:
    - tip stack
    - procedure Init, Push, Pop in Empty

5.9.4 Iskanje k-tega elementa po velikosti

Naloga V neurejeni tabeli z n elementi poišči k-tega po velikosti ($k \in \{1, 2, 3, ...n\}$).

Prva možnost

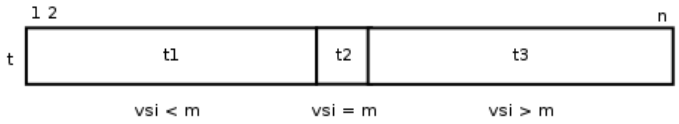
- tabelo uredimo
- rezultat je element, ki je na k-tem mestu (pred njim jih je k-1)
- čas za rešitev je $\Theta(nlogn)$

Motivacija Ker uredimo celo tabelo, se pri $k \ll n$ velik del tabele ureja po nepotrebnem.

Ali se da nalogo rešiti hitreje? Da!

Zamisel (2. možnost)

- tabelo porazdelimo glede na neko vrednost m na tri dele:



- s štetjem elementov v tabelah t_1 in t_2 lahko ugotovimo v kateri od tabel je iskani k-ti element
- ko to ugotovimo, nadaljujemo z iskanjem v tisti podtabeli
- očitno znamo en problem nadomestiti s podobnim manjšim problemom ⇒ možno, da je rešitev rekurzivna, toda ni nujno

Enostavnejša rešitev Rekurzija se izteče, ko je v tabeli en element, m pa izberemo naključno.

Koda

```
1  void Select(k, t)
2  {
3          if(t.length == 1)
4          {
5                  return (edini) element tabele t;
6          }
7          else
8          {
9                  m = nakljucni element tabele t;
10                 porazdeli t na t_1, t_2 in t_3 glede na m;
11                 if(k <= |t_1|)
12                 {
13                         Select(k, t_1);
14                 }
15                 else if
16                 {
17                         return m;
18                 }
19                 else
20                 {
21                         Select(k - |t_1| - |t_2|, t_3);
22                 }
```

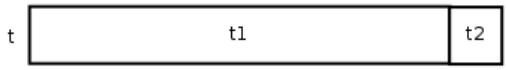
```
23          }
24  }
```

Analiza

1. **Najboljši čas**
   Če imamo srečo, je m vsakokrat mediana (srednja vrednost), zato sta $t_1 \sim t_3 \cong \frac{|t|}{2}$. Zato se tabela v kateri se nadaljuje iskanje vsakokrat približno razpolovi.
   Dolžine tabel, v katerih se izvaja Select, padajo takole: $n, \frac{n}{2}, \frac{n}{4}, ... \frac{n}{2^i}, ..., 1$.
   Za porazdelitev i-te tabele od njih rabimo $\Theta(\frac{n}{2^i})$ korakov, kjer je $i = 0, 1, 2, ...[log_2 n]$.
   Celoten čas je torej:

$$\sum_{i=0}^{log_2 n} \Theta(\frac{n}{2^i}) = \Theta(\sum_{i=0}^{log_2 n} \frac{n}{2^i}) = \Theta(n) \sum_{i=0}^{log_2 n} \left(\frac{1}{2}\right)^i = \Theta(n)$$

2. **Najslabši čas**: $\Theta(n^2)$
   Kako to vidimo?
   Če vsakokrat tabelo razdelimo v:



   je čas enak:

$$\begin{aligned} T(n) &= \text{čas porazdelitev} + \text{čas za reševanje t} = \\ &= cn + T(n-1) = \\ &= c(n + n - 1 + ... + 2 + 1) = \\ &= \Theta(n^2) \end{aligned}$$

3. **Povprečen čas:** $\Theta(n)$
   $\Rightarrow$ **Izpopolnjena rešitev:**

```
1   procedure Select(k, t)
2   {
3       if(|t| < min)    //ce je tabela majhna
4       {
5           Uredi(t);
6           return k-ti element v t;
7       }
8       else
9       {
10          m = izberi element v t;
11          porazdeli t v t1, t2 in t3 glede na m;
12          if (k <= |t1|)
13          {
14              Select(k, t1);
```

```
15          }
16          else if (k <= |t1| + |t2|)
17          {
18              return m;
19          }
20          else
21          {
22              Select(k - |t2| - |t1|, t3);
23          }
24      }
25  }
```

S primerno izbiro m in min lahko dosežemo, da Select(k, t) rabi $T(n) = \Theta(n)$, kjer $n = |t|$.

1. **Kako določiti m?**
   Dobro je, če sta oba t1 in t3 bistveno manjša od t, ker se bo gotovo nadomestil z manjšim problemom.

   Brez dokaza: v času $T(\frac{|n|}{5})$ se da najti takšen m, ki zagotvalja, da $|t1| \leq \frac{3}{4}n$ in $|t3| \leq \frac{3}{4}n$.
   Kako?

   (a) t razdeli na peterke

   (b) v vsaki peterki poišči mediano (srednjega po velikosti). Naj bo M tabela vseh median, srednja po velikosti naj bo m.

   (c) $m = Select(\lceil \frac{|M|}{2} \rceil, M)$

2. **Kako določiti min?**
   Eksperimentalno: min = 50
   Analiza:

   - za $n \leq min : T(n)$
   - za n > min:

   $$\begin{aligned} T(n) &= \text{čas za računanje m} \\ &\quad + \text{čas za razdelitev tabele glede na m} \\ &\quad + \text{čas za enega od klicev nad t1 ali t3} \\ &\quad \leq T(\tfrac{n}{5}) + cn + T(\tfrac{3}{4}n) \end{aligned}$$

   Za rešitev rekurzivne enačbe velja: $T(n) = \Theta(n)$.

6 Zunanje urejanje

Ponovitev

- podatkov "malo"⇒ vse uredimo v notranjem pomnilniku ⇒ **notranje urejanje**
- podatkov "veliko"⇒ vsi so na zunanjem pomn. v tabeli ⇒ **zunanje urejanje**

- npr. magnetni trak, disk - dostop je zaporeden (s pomikanjem datotečnega okna, vidni so le podatki, ki so v oknu)
- okno je vmesnik (buffer) v notranjem pomnilniku
- podatke v oknu lahko uredimo z notranjim urejanjem
- notranje urejanje ni uporabno, ker je dostop zaporeden (razen pri urejanje podatkov v oknu)

6.1 Algoritmi za zunanje urejanje

- navadno zlivanje (osnova vseh ostalih)
- izboljšave navadnega
  - uravnoteženo
  - večsmerno
  - naravno
  - polifazno

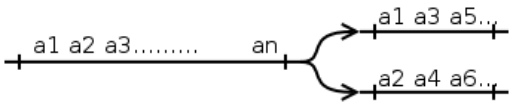
6.2 Navadno zlivanje

Bistvo:

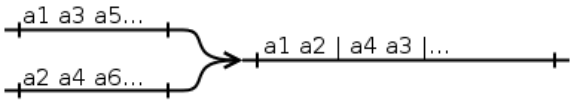
- na vhodnem traku je zaporedje podatkov, ki jih želimo urediti
- na voljo sta tudi dva izhodna trakova

Prvi korak:

- vhodni trak zaporedno beri in prebrane elemente porazdeli (izmenično zapisuj na izhodna trakova)



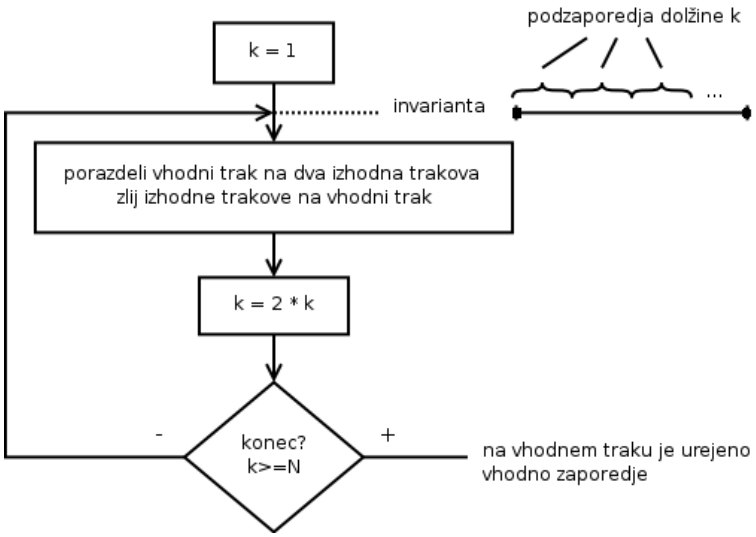
- izhodna trakova oba beri in vsaka dva prebrana elementa zlij v urejen par in ga izpiši na vhodni trak



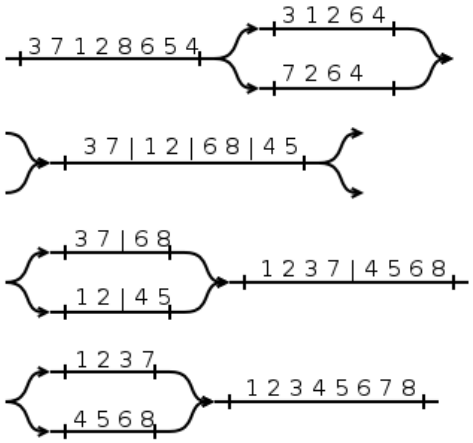
Splošni korak:

- na vhodnem traku so urejena zaporedja dolžine k
- porazdeli jih (izmenino jih zapiši) na dva izhodna trakova
- zlij po dve zaporedji (dolžine k) z izhodnih trakov v eno zaporedje dolžine 2k na vhodni trak

Diagram:



Primer:



Zlivanje dveh zaporedij dolžine k:

$$|b_1\ b_2 ... b_k|$$
$$|c_1\ c_2 ... c_k|$$

- beri prva dva elementa obeh zaporedij
- izpiši manjšega od njiju (*)

```
1 if (izpisani element ni zadnji v svoji k-terki)
2 {
3     beri njegovega naslednjika in se vrni na (*)
4 }
5 else
```

```
6 {
7     izpisi preostanek drugega zaporedja
8 }
```

6.2.1 Analiza navadnega zlivanja

Operacije, ki se pojavljajo so:

- branje podatka ai z vhodnega traku
- primerjava podatkov ai in aj
- izpis podatka ai na trak

Dogovor:

- pri oceni časa upoštevajmo le branje s traku
- ker:
  - primerjanje je bistveno hitrejše od branja s traku
  - zapisovanj na trak je toliko kot branj (⇒ zato je zadosti, če upoštevamo branja)

Kdaj in koliko beremo v traku?

Vsak prehod sestavljata:

- porazdelitev .... ki zahteva N branj z vhodnega traku
- zlivanje..... ki zahteva $\frac{N}{2}$ branj z vsakega izhodnega traku ⇒ vsak prehod rabi 2N branj s trakov

Koliko je vseh prehodov?
$\lceil log_2 N \rceil$, ker se ob vsakem prehodu dolžina k-terk podvoji.
Navadno zlivanje: $2N\lceil log_2 N \rceil = \Theta(Nlog_2 N)$ branj. Dosegli smo magično mejo $\Theta(NlogN)$.
Ugovor: upoštevali smo le operacije branja! Upoštevajmo vse operacije:

- branj: $\Theta(NlogN)$
- zapisovanj: tudi $\Theta(NlogN)$, ker:
  - pri vsaki porazdelitvi izpišemo $2 * \frac{N}{2} = N$-krat
  - pri zlivanju: N izpisov na traku ⇒ 2N izpisov
  - prehodov: $\lceil log_2 N \rceil$ ⇒ vseh izpisovanj je $2N\lceil log_2 N \rceil = \Theta(NlogN)$
  - primerjanj je $\Theta(Nlog_2 N)$, ker:
    - pri vsakem zlivanju $\leq N$ primerjanj
    - prehodov je $\lceil log_2 N \rceil$
    - ⇒ $\Theta(Nlog_2 N)$ primerjanj

Navadno zlivanje ima torej časovno zahtevnost:

$$\Theta(cNlog_2 N) = \Theta(Nlog_2 N)$$

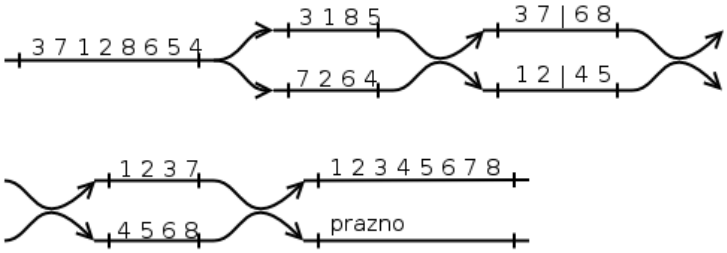
6.3 Izboljšave navadnega zlivanja

6.3.1 Uravnoteženo zlivanje

Opazimo Porazdelitev ne permutira elementov, zato ne prispeva h končni ureditvi elementov, hkrati pa zahteva polovico vseh branj.
Ali se porazdeljevanju lahko ognemo?

Zamisel Namesto da zlita zaporedja (po vrsti) izpisujemo na **en** trak, jih že takoj izmenično izpisujmo na **dva** traka, torej porazdelitev izvajamo hkrati pri zlivanju.

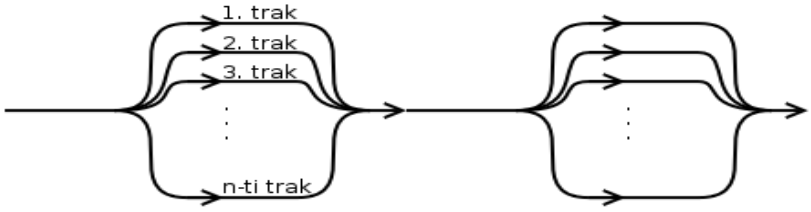
Primer Potrebujemo 4 trakove: 2 vhodna in 2 izhodna.



Opazimo Število branj s traku se prepolovi, ker odpade $Nlog_2N$ branj vhodnega traku pri porazdeljevanju.
Metoda, kjer porazdeljevanje izvajamo sočasno ob zlivanju imenujemo **uravnoteženo zlivanje**.

6.3.2 Večsmerno zlivanje

Zamisel Denimo da zopet zlivamo na en trak, toda za porazdeljevanje uporabimo n trakov, kjer je $n > 2$.



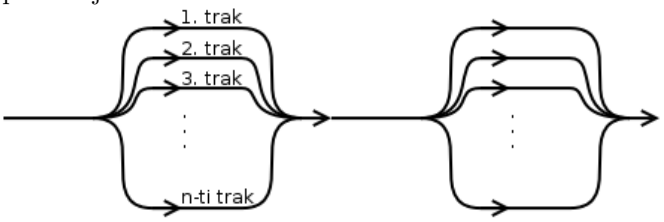
Časovna zahtevnost

- za vsako porazdelitev je potrebnih N branj
- zlivanje z n trakov zahteva $n\frac{N}{n} = N$ branj
- po zlitju zaporedij dolžine k z izhodnih trakov dobimo novo zaporedje z $n * k$ elementi
- $\Rightarrow$ potrebnih je $\lceil log_n N \rceil$ prehodov

- $\Rightarrow$ SKUPAJ: $2N\lceil log_n N\rceil$ branj s traku.
  V primerjavi z navadnim zlivanjem (n = 2) je to $log_2 n$-krat hitrejši algoritem

Obe izboljšavi lahko upoštevamo hkrati - imamo večsmerno uravnoteženo zlivanje:

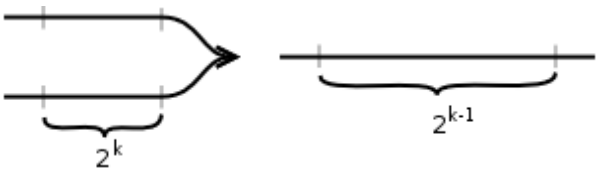
- potrebujemo 2n trakov



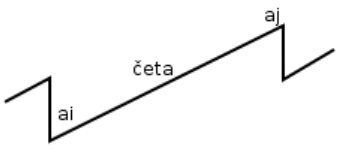
- število branj se prepolovi, ker se ognemu branju N elementov pred porazdeljevanjem
- čas: $cNlog_n N = \Theta(NlogN)$

6.3.3 Naravno zlivanje

Opazimo Doslej smo zlivali zaporedja z **enakimi** dolinami, npr. pri 2-smernem (n=2) zlivanju v k-tem prehodu zlivamo zaporedja dolžine 2^{k-1} v zaporedja dolžine 2^k.
Toda včasih je več kot 2^{k-1} elementov že urejenih, zato je škoda, da ne zlijemo vseh urejenih skupaj, ker bi dobili urejeno zaporedje, ki bi bilo daljše od 2^k. Vseh N bi lahko hitreje uredili.

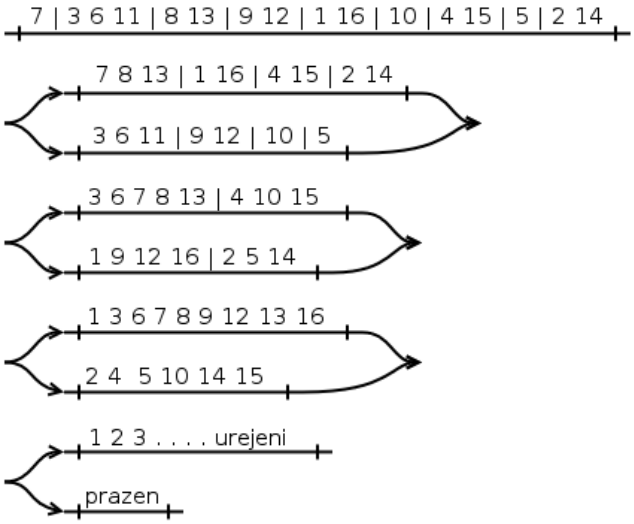


Zamisel **Četa** je vsako urejeno zaporedje, ki se ga ne da razširiti, ne da bi izgubilo svojo urejenost $a_{i-1} > a_i \leq a_{i+1} \leq ... \leq a_j > a_{j+1}$.



Namesto z zaporedji predpisane dolžine se bomo ukvarjali s **četami** (porazdeljevanje, zlivanje).

Primer



Analiza Ko zlijemo 2 četi nastane nova četa. Pri vsakem prehodu se število čet prepolovi (če je n=2). Če je bilo v vhodnem nizu C čet, bo potrebnih $\lceil log_2 C \rceil$ prehodov.
Seveda: $1 \leq C \leq N$. Bolj kot je vhod urejen, manjši je C.
Pri vsakem prehodu je N branj elementov. Iz tega sledi, da je število korakov (čas):

$$N\lceil log_2 C \rceil = \Theta(NlogC)$$

Prednost Ko je vhod že delno (dobro) urejen (manjši C), je manjši tudi čas urejanja.
Vse doslej opisane izboljšave lahko kombiniramo.

6.3.4 Polifazno urejanje

Motivacija Pri uravnoteženem večsmernem (tudi naravnem) urejanju imamo **2n trakov**. Z n vhodnih trakov se zaporedja berejo, zlita pa se izpisujejo na n izhodnih trakov.
Toda, ko se neko zaporedje izpisuje na nek izhodni trak, ostali izhodni trakovi **čakajo**. Torej v vsakem trenutku čaka n-1 izhodnih trakov.

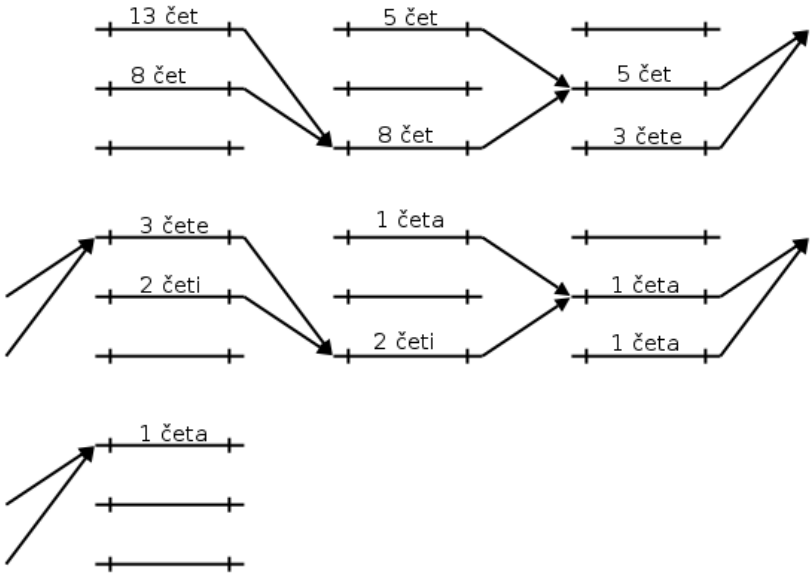
Kako bi trakove bolje izkoristili?

Zamisel Imejmo le n trakov, vsak trak bo lahko zamenjal svojo vlogo - včasih bo vhodni, drugič pa izhodni. V vsakem trenutku bo n-1 vhodnih in le en izhodni trak.

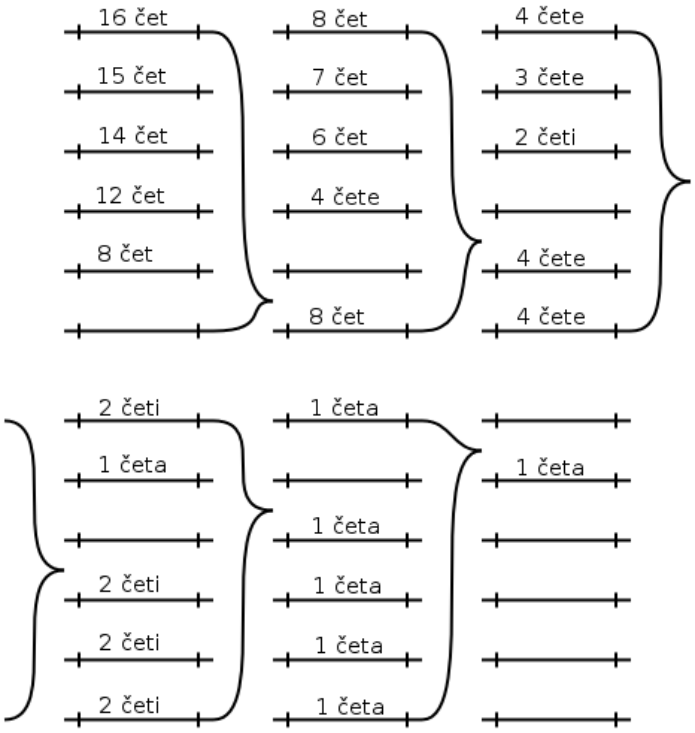
Kako to zagotoviti?
Vhodne trakove zlivajmo na izhodnega tako, da bo vedno eden med vhodnimi usahnil oz. da se bo izpraznil, na koncu zlivanja na njem ne bo nobenih podatkov več. Ko nek vhodni trak usahne, takoj postane naslednji izhodni. Prejšnji izhodni pa postane vhodni.

Primer 1

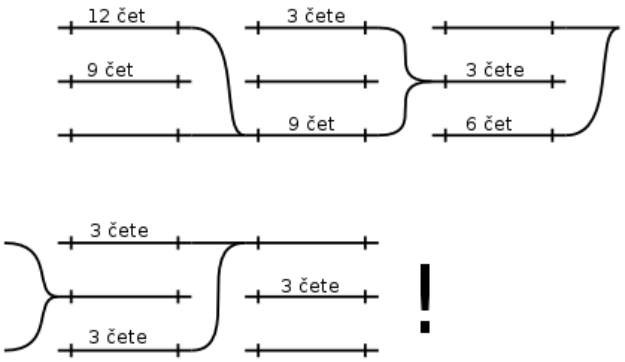
- n = 3
- 1. in 2. trak imata 13 oz. 8 čet
- 3. trak je izhodni
- po c čet z vsakega vhodnega traku zlijemo v c čet na izhodni trak



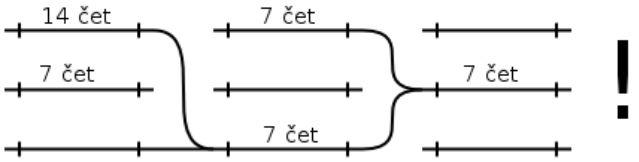
Primer 2 n = 6



Primer 3



Primer 4



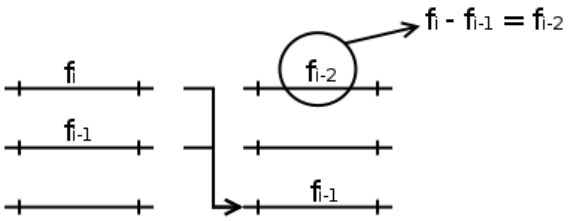
Nekatere začetne porazdelitve čet po vhodnih trakovih so ugodne, tj. vodijo do ene čete na enem traku, druge porazdelitve pa ne.

Katere začetne porazdelitve so ugodne?
Poglejmo primer 1, opazimo števila: $0, 1, 1, 2, 3, 5, 8, 13$. V njih prepoznamo ti. Fibbonacijeva števila (reda 1), ki so definirana:

$$f_0 = 0$$
$$f_1 = 1$$
$$f_i = f_{i-1} + f_{i-2}, \text{ če je } i \geq 2$$

Trditev Da bi se polifazno urejanje pri n=3 uspešno zaključilo, morata biti začetni števili čet na vhodnih trakovih dve zaporedni Fibbonacijevi števili reda 1.

Dokaz



Kaj storiti, če celotno število čet na začetku ni ravno neko Fibbonacijevo število?

1. Manjkajoče čete zapišemo v obliki "navideznih čet", toliko da dopolnemo celotno število čet do naslednjega Fibbonacijevega števila.
   (a) **Kako jih porazdeliti?**
   Pri zlivanju čet se število čet karseda povečuje. Zlivaj jih med seboj, če se le da. Poskušamo jih čimbolj enakomerno porazdeliti po trakovih.
   (b) **Kako jih predstaviti?**
2. Predurejanje (opisano v knjigi)
   Trajanje postopka je odvisno od začetnega števila čet (od f_i oz. i).

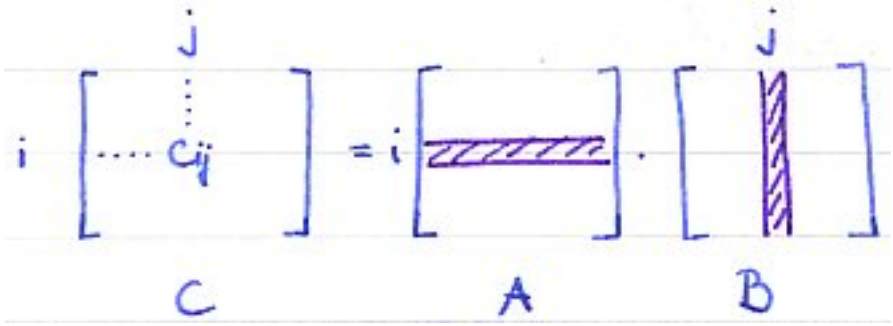
Zamisel Zmanjšati skušamo i oz. začetno število čet. Če je na voljo dovolj notranjega pomnilnika z notranjim urejanjem delno uredimo podatke, da zmanjšamo začetno število čet.

Del IV
Algoritmi z rekurzivnim razcepom

7 Množenje matrik

Uvod: Dani sta matriki A in B reda $n \times n$. Produkt je matrika $C = A \cdot B$ reda $n \times n$, s komponentami:

$$c_{ij} = \sum_{k=1}^{n} A_{ik} \cdot B_{kj}$$



Najpreprostejši algoritem: Izračunaj po vrsti vseh n^2 elementov cij po definiciji.
Npr:

```
1 for i = 1 to n do
2     for j = 1 to n do
3         for k = 1 to n do
4             cij = cij + Aik . Bkj
5         end
6     end
7 end
```

Časovna zahtevnost:

$$\left.\begin{array}{l} n^3 \text{ seštevanj} \\ n^3 \text{ množenj} \end{array}\right\} \Theta(n^3)$$

7.1 Winogradovo množenje matrik

Winogradov algoritem: Zmanjša št. množenj za $\sim 50\%$, toda asimptotično je še vedno reda $\Theta(n3)$

Ali se da matriki reda n x n zmnožiti z manj kot n3 množenji?
Odkril je naslednji algoritem: Naj bo n sodo št. (če ni ga povečaj za 1, A in B

pa povečaj za prazno vrstico in stolpec).
Najprej izračunaj količine:

$$\left.\begin{array}{l}\Pi_{i,k,j} = (A_{i,2k-1} + B_{2k,j}) \cdot (B_{2k-1,j} + A_{i,2k}) \\ \varrho_{i,k} = A_{i,2k-1} \cdot A_{i,2k} \\ \sigma_{k,j} = B_{2k,j} \cdot B_{2k-1,j}\end{array}\right\} \begin{array}{l}\text{pri vseh } 1 \leq i,j \leq n \\ \text{in pri } 1 \leq k \leq \frac{n}{2}\end{array}$$

Nato jih porabimo za računanje komponent C_{ij} takole:

$$C_{i,j} = \sum_{k=1}^{\frac{n}{2}} (\Pi_{ikj} - \varrho_{i,k} - \sigma_{k,j})$$

Dokaz:

$$\begin{array}{lll} C_{ij} & = \sum_{k=1}^{\frac{n}{2}} & (\Pi_{ikj} - \varrho_{i,k} - \sigma_{k,j}) = \\ & = \sum_{k=1}^{\frac{n}{2}} & (A_{i,2k-1}B_{2k-1,j} + A_{i,2k-1}A_{i,2k} + B_{2k,j}B_{2k-1,j} \\ & & +B_{2k,j}A_{i,2k} - A_{i,2k-1}A_{i,2k} - B_{2k,j}B_{2k-1,j}) = \\ & = \sum_{k=1}^{\frac{n}{2}} & (A_{i,2k-1}B_{2k-1,j} + B_{2k,j}A_{i,2k}) = \\ & = \sum_{k=1}^{\frac{n}{2}} & (A_{i,2k-1}B_{2k-1,j} + A_{i,2k}B_{2k,j}) = \\ & = \ldots = & \\ & = \sum_{i=1}^{n} & (A_{il}B_{lj}\ldots) \Rightarrow \text{ komponenta produka } C = A \cdot B \end{array}$$

Časovna zahtevnost:

- Π_{ikj} izračunamo z enim množenjem (in dvema seštevanjema) $\Rightarrow$ skupaj $\frac{1}{2}$ množenj $\Rightarrow \frac{1}{2}n^3$
  $\Pi_{ikj} = \frac{1}{2}n^3$
- ϱ_{ik} izračunamo z enim množenjem
- $\sigma_{kj} = \frac{1}{2}n^2$ množenj
- $\Rightarrow$ skupaj $\frac{n^2}{2}$ množenj

Za izačun vseh Π, ϱ, σ rabimo $\frac{1}{2}n^3 + n^2$ množenj, kar je asimptotično gledano praktično 50% manj kot pri klasičnem algoritmu.
Seštevanja je $n^3 + 3n^2 2n$, kar je več kot pri klasičnem algoritmu, vendar to ni problem, ker množenje časovno prevladuje.

Implementacija tega algoritma na rekurziven način je lahko problematična, ker algoritem predvideva, da je množenje komutativno, kar pa ne velja za vse matrike.

7.2 Rekurzivno množenje matrik

Predpostavimo: n je potenca št. 2 (sicer matrike dopolni naslednjega takega št.)
Matriki A in B razdelimo na podmatriki velikosti $\frac{n}{2} \times \frac{n}{2}$ in pomnožimo podmatrike:

$$A \cdot B = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \begin{bmatrix} B_{11} & B12 \\ B_{21} & B_{22} \end{bmatrix} = \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} = C,$$

kjer je

$$C_{11} = A_{11}B_{11} + A_{12}B_{21}$$
$$C_{12} = A_{11}B_{12} + A_{12}B_{22}$$
$$C_{21} = A_{21}B_{11} + A_{22}B_{21}$$
$$C_{22} = A_{21}B_{12} + A_{22}B_{22}.$$

8 Diskretna Fourierjeva transformacija

Motivacija Dana imamo dva polinoma

$$p(x) = a_0 + a_1x + a_2x^2 + \ldots + a_{77}x^{77}$$
$$q(x) = b_0 + b_1x + b_2x^2 + \ldots + b_{94}x^{94}$$

pri čemer so števila a_i in b_j so dana.

Koliko je produkt $p(x)q(x)$?
Če hočemo, da je $p(x)q(x)$ koeficientno predstavljen, tj. če želimo

$$p(x)q(x) = c_0 + c_1x + \ldots + c_{171}x^{171}$$

moramo izračunati 172 koeficientov $c_0, c_1, ...c_{171}$. Za to je potrebnih veliko množenj (koliko?), npr. za izračun c_{61} jih je potrebnih 62. Zato ker:

$$c_{61} = a_0b_{61} + a_1b_{60} + \ldots + a_{61}b_0 = \sum_{n=0}^{61} a_nb_{61-n}$$

Polinom p(x)q(x) pa je natančno določen s svojo vrednostno predstavitvijo, tj. s seznamom svojih vrednosti v 172 izbranih točkah $x_1, x_2, ...x_172$. Če bi bila funkcija $p(x)$ in $q(x)$ vrednostno predstavljena na $x_1, x_2, \ldots x_{172}$, torej če:

$$p(x) \equiv (p(x_1),\ p(x_2), \ldots p(x_i), \ldots p(x_{172}))$$
$$q(x) \equiv (q(x_1),\ q(x_2), \ldots q(x_i), \ldots q(x_{172}))$$

bi njun produkt izračunali enostavno. Zmnožili bi istoležne predstavnike $p(x)$ in $q(x)$. Za to bi potrebovali 172 množenj.

Toda kaj storiti, če $p(x)$ in $q(x)$ nista predstavljena vrednostno, ampak koeficientno, tj. $[a_0, a_1, a_2, ...a_n]$ in $[b_0, b_1, b_2, ...b_m]$?

1. **Ali lahko hitro sestavimo njuni vrednostni predstavitvi?**
2. **Ali in kako vrednostno predstavitev produkta pretvorimo v koeficientno?**
3. **Katera naj bodo števila $x_1, x_2, ...x_172$?**

S sliko

$$\begin{array}{rcc} p(x) \equiv [a_0, a_1, a_2, ...a_{77}] & \Longrightarrow & (p(x_1), p(x_2), ...p(x_{??})) \\ q(x) \equiv [b_0, b_1, b_2, ...b_{94}] & \Longrightarrow & (q(x_1), q(x_2), ...q(x_{172})) \\ & & \Downarrow \\ p(x)q(x) = [c_0, c_1, c_2, ...c_{171}] & \Longleftarrow & (p(x_1)q(x_1), p(x_2)q(x_2), ...p(x_{172})q(x_{172})) \end{array}$$

Na to odgovarja diskretna Fourierjeva transformacija (na kratko DFT). DFT preslika eno predstavitev polinoma v drugo. DFT je zelo uporabna:

- množenje polinomov
- računalniska aritmetika velikih natančnosti
- računalniska tomografija
- jedrska magnetna resonanca

Iz te pomembnosti DFT sledi, da mora biti algoritem za DFT enostaven in **hiter**.

8.1 Naivni algoritem

- ima zahtevnost $\Theta(n^2)$, kjer je n velikost koeficientne predstavitve
- vhod: polinom $p(x) = [a_0, a_1, ...a_{n-1}]$, stopnja n in števila $x_1, x_2, x_3, ...x_n$
- izhod: vrednosti $p(x_1), p(x_2), ...p(x_n)$
- algoritem: Vrednost $p(x_i) = (...((a_{n-1}x_i + a_{n-2})x_i + a_{n-3})x_i + ... + a_1)x_i + a_0$ izračunamo z n-1 množenji.

  Npr. $((a_3x + a_2)x + a_1)x + a_0$ izračunamo s tremi množenji.

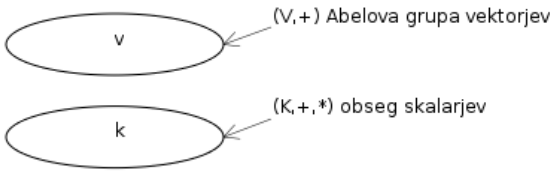
  Vseh n vrednosti polinoma $p(x_1), p(x_2), ...p(x_n)$ izračunamo z $\Theta(n^2)$ množenji.

Toda, da se izračunati tudi v času $\Theta(nlogn)$, kar je pri velikih n bistveno boljše.

8.2 Diskretna Fourierjeva transformacija (splošno)

Naj bo V_k vektorski prostor V nad obsegom K.

Torej:



Definirano je množenje skalarja in vektorja: $K * V \to V$. Naj bo $n = dim(V_k)$ dimenzija vektorskega prostora V_k.

Definicija Naj bo v K element w z lastnostmi:

$$w^n = 1 \tag{8}$$

$$w^i \neq 1 \text{ za } i = 1, 2, 3, ...n - 1 \tag{9}$$

Pravimo, da je w n-ti primitivni koren enote.

Trditev: $(8) \wedge (9) \iff \sum_{i=0}^{n-1} w^{p_i} = \begin{cases} n;\ p = n \\ 0;\ 1 \leq p \leq n-1 \end{cases}$

Dokaz: V knjigi Osnovni algoritmi na strani 84.

Primer: Če $K \in C$ (kompleksna števila), je n-ti primitivni koren enote:

$$w = e^{i\frac{2\Pi}{n}}, \text{ kjer je } i = \sqrt{-1}.$$

Diskretna Fourierjeva transformacija prostora V_k je linearna transformacija tega prostora z matriko F, katere elementi so

$$F_{ij} = w^{ij}, \text{ kjer } 0 \leq i, j \leq n-1, \text{ w pa n-ti primitivni koren.}$$

Opomba: Zaradi (8) vidimo, da velja

$$w^{ij} = w^{ij \bmod n},$$

torej se v F pojavijo le vrednosti $1, w^1, w^2, ...w^{n-1}$.

Če F predstavlja DFT prostora V_k bi matrika F^{-1} predstavljala inverzno transformacijo prostora V_k.

Toda: Ali F^{-1} obstaja?

Trditev: Če k vsebuje element $\frac{1}{n}$ (multiplikativni inverz od n), potem F^{-1} obstaja, njeni elementi pa so:

$$F_{ij}^{-1} = \frac{1}{n} w^{-ij}$$

Dokaz: Pomnožimo F in F^{-1}.

Primer za n=2: Kakšen je primitivni koren enote?

$w = -1$, ker:

- $w^2 = (-1)^2 = 1$ (glej 8)
- $w^1 = (-1)^1 \neq 1$ (glej 9)

DFT ima matriko:

$$F = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix},$$

inverz pa je

$$F^{-1} = \frac{1}{2} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}.$$

Opomba: Tu je $F^-1 = \frac{1}{2}F$, v splošnem pa to ne velja.

Primer za n=4: Naj bo w 4. primitivni koren enote. Torej:

$$F = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & w & w^2 & w^3 \\ 1 & w^2 & 1 & w^2 \\ 1 & w^3 & w & w^1 \end{bmatrix} \text{ in } F^{-1} = \frac{1}{4} \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & w^{-1} & w^{-2} & w^{-3} \\ 1 & w^{-2} & 1 & w^{-2} \\ 1 & w^{-3} & w^{-2} & w^{-1} \end{bmatrix},$$

ker je $w = e^{i\frac{2\Pi}{4}} = e^{i\frac{\Pi}{2}}$, je $w^2 = e^{i\Pi} = -1$.

Podobno: $w^{-2} = e^{-i\Pi} = -1$.

Nadalje: $w^{-1} = w^{-1+4} = w^3 = w^{3i\frac{\Pi}{2}} = w^{i\Pi} w^{i\frac{\Pi}{2}} = -w$ ter $w^{-3} = w^{-3+4} = w$. Zato

$$F = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & w & -1 & -w \\ 1 & -1 & 1 & -1 \\ 1 & -w & -1 & w \end{bmatrix} \text{ in } F^{-1} = \frac{1}{4} \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & -w & -1 & w \\ 1 & -1 & 1 & -1 \\ 1 & w & -1 & -w \end{bmatrix}.$$

Če bi bil $a = [1, 1, 2, 2]^t$ vektor, bi bila njegova DFT slika vektor:

$$Fa = \begin{bmatrix} & & & \\ & & & \\ & & & \\ & & & \end{bmatrix} \begin{bmatrix} 1 \\ 1 \\ 2 \\ 2 \end{bmatrix} = \begin{bmatrix} 6 \\ -1 \\ -w \\ -1 + w \end{bmatrix}$$

8.3 Diskretna Fourierjeva transformacija in polinomi

Doslej je bil V_k splošen vektorski prostor.

Sedaj bo V_k bolj konkreten, izbrali si bomo enega od možnih, to bo prostor polinomov ene spremenljivke po modulu x^n.

Bazo takšnega V_k sestavljajo polinomi $x^i, i = 0, 1, ...n-1$. S to bazo lahko izrazimo vsak polinom stopnje $\leq n-1$ in sicer v obliki:

$$\sum_{i=0}^{n-1} a_i x^i, a_i \in K.$$

Torej je V_n prostor polinomov stopnje $\leq n-1$.

Naj bo $p(x) \in V_k$.

Koeficientna predstavitev p(x) je vektor $a = [a_0, a_1, ...a_{n-1}]$ za katerega velja $p(x) = \sum_{i=0}^{n-1} a_i x_i$.

Vrednostna predstavitev p(x) je n-terka $(p(x_0), p(x_1), ...p(x_{n-1}))$, kjer so $x0, x1, ...x_{n-1} \in K$.

Trditev DFT preslika koeficientno predstavitev polinoma v vrednostno predstavitev na točkah $x_i = w^i$. Transformiramo jo z matriko F.

$$F_a = i \begin{bmatrix} & \vdots & \\ \cdots & w^{ij} & \cdots \\ & \vdots & \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_{n-1} \end{bmatrix} = i \begin{bmatrix} & \vdots & \\ \cdots & \sum_{j=0}^{n-1} a_j w^{ij} & \cdots \\ & \vdots & \end{bmatrix}$$

Na koncu smo dobili vrednostno predstavitev polinoma p(x) na točkah $w^0, `w^1, ...`w^{n-1}$. Sedaj lahko začnemo spet razmišljati o produktu dveh polinomov.

8.4 Konvolucija polinomov

Definicija Naj bosta p(x) in q(x) polinoma in $a = [a_0, a_1, ...a_{n-1}]$ in $b = [b_0, b_1, ...b_{n-1}]$ njuni koeficientni predstavitvi. Konvolucija polinomov p(x) in q(x) je koeficientna predstavitev produkta $p(x) \cdot q(x)$. Označimo jo z $a \cdot b$.

Računajmo

$$p(x) \cdot q(x) = (\sum_{i=0}^{n-1} a_i x^i)(\sum_{j=0}^{n-1} b_j x^j) = \ldots = \sum_{u=0}^{2n-2} (\sum_{l=0}^{n-1} a_l b_{k-l}) x^k$$

Če je k-l izven intervala [0, n-1] vzamemo bk-l = 0.

Torej $a \cdot b = [a_0, a_1, ...a_{n-1}] \cdot [b_0, b_1, ...b_{n-1}]$ je vektor $c = [c_n, c_1, ...c_{2n-2}]$ z 2n-1 komponentami, ker je $c_k = \sum_{l=0}^{n-1} a_l b_{k-l},\ k = 0, 1, 2, ...2n-2$.

Primer za n=5 Imamo polinoma $a = [a_0, a_1, ...a_4]$ in $b = [b_0, b_1, ...b_4]$, torej je konvolucija $c = [c_0, c_1, ...c_8]$.

$$\begin{aligned}
c_0 &= a_0 b_0 \\
c_1 &= a_0 b_1 + a_1 b_0 \\
c_2 &= a_0 b_2 + a_1 b_1 + a_2 b_0 \\
c_3 &= a_0 b_3 + a_1 b_2 + a_2 b_1 + a_3 b_0 \\
c_4 &= a_0 b_4 + a_1 b_3 + a_2 b_2 + a_3 b_1 + a_4 b_0 \\
c_5 &= a_0 b_5 + a_1 b_4 + a_2 b_3 + a_3 b_2 + a_4 b_1 + a_5 b_0 \ldots
\end{aligned}$$

Trditev $a \cdot b$ lahko izračunamo z množenji, kjer je $a = [a_0, a_1, ...a_4]$ in $b = [b_0, b_1, ...b_4]$.

Dokaz Za izračun c_k je potrebnih $\begin{cases} k+1 \text{ množenj } zak = 0, 1, ...n-1 \\ 2nk1 \text{ množenj za } k = n, n+1, ...2n-2 \end{cases}$

Skupaj množenj za vse $c_k, k = 0, 1, ...2n-2$:

$$\begin{aligned}
& \textstyle\sum_{k=0}^{n-1}(k+1) + \sum_{k=n}^{2n-2}(2n-k-1) = \\
= \;& 1 + 2 + \ldots + n + (n-1) + (n-2) + \ldots + 1 = \\
= \;& \tfrac{n(n+1)}{2} + \tfrac{(n-1)n}{2} = \\
= \;& n^2
\end{aligned}$$

Qed: Konvolucija ima časovno zahtevnost $\Theta(n^2)$.

Ali bi se dalo $a \cdot b$ izračunati hitreje?
Da, toda z uporabo DFT. DFT in konvolucijo povežemo z naslednjim izrekom o konvoluciji.

8.5 Konvolucija polinomov in DFT

Priprava Naj bo $p(x) = sum_{i=0}^{n-1} a_i x^i$. Koeficientna predstavitev bi bila $a = [a_0, a_1, ...a_{n-1}]$. Izkazalo se bo, da je koristno, če jo dopolnimo z ničlami (pišemo jo kot vektor stolpec). Koeficientna predstavitev polinoma p(x) bo torej

$$a = [a_0, a_1, ...a_{n-1}, \underbrace{0, 0, ...0}_{\text{n ničel}} \ ] .$$

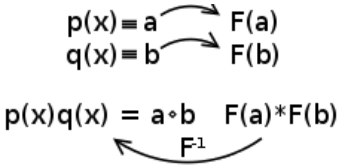
Definicija Če sta $u = [u_1, u_2, ...u_k]^t$ in $v = [v_1, v_2, ...v_k]^t$ je

$$u * v = [u_1 v_1, u_2 v_2, ...u_k v_k].$$

Izrek o konvoluciji Naj bosta $p(x) = \sum_{i=0}^{n-1} a_i x^i$ in $g(x) = \sum_{j=0}^{n-1} b_j x^j$ polinoma stopnje n-1, njuni koeficientni predstavitvi pa $a = [a_0, a_1, ...a_{n-1}, 0, 0, ...0]^t$ in $b = [b_0, b_1, ...b_{n-1}, 0, 0, ...0]^t$. Naj bo F DFT stopnje 2n. DFT transformiranki sektorjev a in b naj bosta $F(a) = Fcdota = [u_0, u_1, ...u_{2n-1}]$, kjer $u_i = p(w^i)$ in $F(b) = Fcdotb = [v_0, v_1, ...v_{2n-1}]$, kjer je $v_i = q(w^i)$, kjer je w 2n-ti primitivni koren enote.

Tedaj velja: $a \cdot b = F^{-1}(F(A) * F(B))$.

Slika :



Dokaz je v knjigi Osnovni algoritmi, str. 88.

8.6 Rekurzivni algoritem za DFT

Naj bo n = 2r in w n-ti primitivni koren enote.

1. Problem DFT(a, n): Iz dane koeficientne predstavitve $a = [a_0, a_1, ...a_{n-1}]$ polinoma
$$p(x) = \sum_{i=0}^{n-1} a_i x^i$$
izračunati vrednost
$$p(x) = \sum_{i=0}^{n-1} a_i x^i$$

v točkah

$$w^0, w^1, ...w^{n-1}.$$

2. V p(x) so koeficienti ob sodih potencah x^i:

$$a_0, a_2, a_4, ...a_{n-2} \text{ vseh je } \frac{n}{2} = r$$

V p(x) so koeficienti ob lihih potencah x^i:

$$a_1, a_3, a_5, ...a_n \text{ vseh je } \frac{n}{2} = r$$

Iz teh sestavimo dva nova polinoma, oba stopnje r-1:

$$p(x) = a_0 + a_2x^2 + a_4x^3 + a_6x^4 + \ldots + +a_{n-1}x^{r-1} = \Pi_0$$

$$p(x) = a_1 + a_3x^2 + a_5x^3 + a_7x^4 + \ldots + +a_{n-1}x^{r-1} = \Pi_1$$

3. p(x) lahko izrazimo s p_0 in p_1:

$$p(x) = p_0(x^2) + xp_1(x^2)$$

4. Problem DFT(a, n) tj. izračunaj $p(\cdot)$ v n točkah $w^0, w^1, ...w^{n-1}$ se zaradi točke 3 prevede na dva koraka:

   (a) Izračunaj dva polinoma p_0 in p_1, v n točkah $(w^0)^2, `(w^1)^2, ...`(w^{n-1})^2$.

   (b) Iz dobljenih vrednosti sestavi (na osnovi točke 3) vrednosti polinoma p v $w^0, w^1, ...w^{n-1}$.

5. Velja: $(w^{k+r})^2 = w^{2k}$. Dokaz: $(w^{k+r})^2 = w^{2k} + w^{2r} = w^{2k}$ Qed.

6. To pomeni, da sta med n točkami:

$$(w^0)^2, (w^1)^2, (w^2)^2, ...(w^k)^2, ...(w^{r-1})^2, ...(w^{k+r})^2, ...(w^{n-1})^2,$$

   po 2 enaki: $(w^k)^2 = w^{2k} = (w^{k+r})^2$, zato jih je le $\frac{n}{2} = r$ različnih. Torej bo treba (4.1) vrednosti p_0 in p_1 računati le v $r = \frac{n}{2}$ točkah. To je v točkah $(w^0)^2, (w^1)^2, ...(w^{r-1})^2$ oz. $(w^2)^0, (w^2)^1, ...(w^2)^{r-1}$.

7. Ker je w n-ti primitivni koren, je $w^2 = \frac{n}{2}$ = r-ti primitivni koren enote. Dokaz za $C : w^{i\frac{2\Pi}{n}}$: je $w^2 = e^{i\frac{2\Pi}{\frac{n}{2}}} = e^{i\frac{2\Pi}{r}}$ Qed.

8. Če upoštevamo točke 6 in 7, se 4.1 po novem glasi:

   (a) Izračunaj dva polinoma p_0 in p_1 v točkah $\Psi^0, \Psi^1, ...\Psi^{r-1}$. Toda 8.1 je sestavljena iz dveh podproblemov, ki sta enake vrste kot osnovni problem DFT(a, n), le da sta pol manjša.

9. Z 8.a je odprta možnost za rekurzivno reševanje. Toda: učinkovitost bo odvisna od učinkovitosti točke 4.2 tj. od zlaganja delnih rešitev v končno.

10. V točki 4.2 je treba iz vrednosti p_0 in p_1, ki jih izračuna 8.1. sestaviti vse:

$$p(w^0), p(w^1), ...p(w^{r-1}); p(w^r), ...p(w^{n-1})$$

11. Prva polovica so $p(w^k)$, druga pa $p(w(rk))$, kjer je $k = 0, 1, ...r-1$.

12. Računanje teh poteka po točki 3:

$$p(w^k) = p_0(w^{2k}) + w^k p_1(w^{2k}) = p_0(\Psi) + w^k p_1(\Psi^n)$$

Za izračun vseh r vrednosti $p(w^n)$ je potrebnih r množenj (in r seštevanj).

13. Pri računanju druge polovice, tj. $p(w^{r+k})$ sploh ni množenja (le odštevanje), ker lahko uporabimo že izračunane produkte.

Dokaz

$$\begin{aligned} p(w^{r+k}) &= p_0(w^{2(r+k)}) + w^{r+k} p_1(w^{2(r+k)}) = \\ &= p_0(w^{2k}) + w^{r+k} p_1(w^{2k}) = \\ &= p_0(\Psi^k) + w^{r+k} p_1(\Psi^k) = \\ &= /w^{r+k} = -w^k/ = \\ &= p_0(\Psi^k) - w^k p_1(\Psi^k) \end{aligned}$$

[1] Qed

Sklep Toda 4.2 lahko izvedemo z $r = \frac{n}{2}$ množenj in $2r = n$ seštevanj/odštevanj.

8.6.1 Psevdokoda rekurzivnega algoritma za DFT

```
1  PROCEDURE RekurzivnoResiDFT(a,n)          //T(n)
2  BEGIN
3      IF n=1 THEN RETURN a
4      ELSE
5          Pripravi w, Pi0, Pi1;    //O(n)
6          RekurzivnoResiDFT(Pi0, n/2);     //T(n/2)
7          RekurzivnoResiDFT(Pi0, n/2);     //T(n/2)
8          sestavi vrednost p v vseh tockah        //O(n)
9      END
10 END
```

Časovna zahtevnost

$$T(n) = 2T(\frac{n}{2}) + \Theta(n)$$

Rešitev je:

$$T(n) = \Theta(nlogn)$$

Del V
Požrešni algoritmi

Uvod Optimizacijski problem je naloga "poišči najboljši element v množici".

[1] Ti produkti so bili izračunani v koraku 12, tu jih moramo le odšteti.

Vprašanja:

- množica
  1. **Kaj so elementi množice?**
  2. **Koliko jih je?**
     - štev no
       - končno
         - "malo"
         - "veliko"
       - neskončno
     - neštevno
- najboljši
  1. **Kako elementom določimo vrednost?**
  2. **Katerim elementom množice U lahko določimo vrednost?**
     - vsem?
     - le dopustnim (ki izpolnjujejo določene kriterije - kakšne?)
  3. **Kaj pomeni najbolši?**
     - dopustni z največjo vrednostjo?
     - dopustni z najmanjšo vrednostjo?

9 Osnovni algoritem za iskanje najboljsega elementa

Vhod: množica $U = \{u_1, u_2, \ldots, u_n\}$
Izhod: indeks m in vrednost v najboljšega elementa v U
Metoda: "na silo"preišči celo množico U

Psevdokoda:

```
1 begin
2 m = 1, v = vrednost(u_m)
3 for i = 2 to n do
4         if dopusten(u_i) && boljsi(vrednosti(u_i),v) then
5                 m = i;
6                 v = vrednost(u_i);
7         end if
8 end for
9 end
```

Opomba: Algoritem ni slab, če je n majhen.
Včasih je potrebno poiskati najboljšo podmnožico od U. Prilagodimo osnovni algoritem za iskanje najboljšega elementa v $P(U)$.

Algoritem: "Iskanje najboljše podmnožice"
Vhod: množica $U = \{u_1, u_2, \ldots, u_n\}$
Izhod: podmnožica $S_m \subseteq U$, ki je najboljša (S_x pomeni podmnožica od U, ki vsebuje tiste elemente, kot pove binarno zapisano število x)
Metoda: preišči celo $P(U)$, tj. vse $x = 0, 1, \ldots, 2^n - 1$

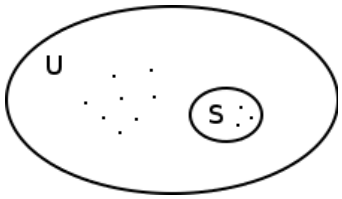
Psevdokoda:

```
1  int m = 0;
2  int v = vrednost(S_m);
3  int x;
4  for(x = 1; x <= pow(2, n-1); i++)
5  {
6          if(dopustna(S_x) && boljsa(vrednosti(S_x), v))
7          {
8                  m = x;
9                  v = vrednost(S_m);
10         }
11 }
```

Ali ne gre drugače kot z grobo silo (pregledovanje cele $P(U)$)?
Kdaj se da najti najboljšo podmnožico, ne da bi bilo treba pregledati celo P(U)?
Odgovorov je več:
Denimo da velja:

- vrednost podmnožice je vsota vrednosti vseh njenih elementov
- dodajanje elementa v podmnožico ne postavi pod vprašaj izbire njegovih predhodnikov

Takrat je smiselna ti. **požrešna metoda**.



Med dopustnimi elementi izberi in dodaj v S tistega, ki najbolj izboljša vrednost S.

10 Požrešni algoritem (osnovna zgradba)

Vhod: množica $U = u_1, u_2, ...u_n$ in predpostavke 9
Izhod: najboljša podmnožica $S \leq U$ in njena vrednost v
Metoda: požrešna
Algoritem: Naj bo $U_{i1}, U_{i2}, U_{i3}, ...U_{in}$ in ureditev elementov po padajočih vrednostih

```
1  v = -inf;          //ali neko najmanjse st.
2  S = 0;
3  for(int k=1; k <= n; k++)
4  {
5          if(dopustna(Unija(S, u_ik)))
```

```
6           {
7                   S = Unija(S, u_ik);
8                   v = v + vrednost(u_ik);
9           }
10  }
```

Opombe:

- včasih ne uredimo elementov na začetku temveč pri sami množici iz S "izračunamo"dopustne elemente/kandidate za včlanitev in med njimi poiščemo najboljšega
- včasih je relativno težko dokazati, da je rešitev S res najboljša
- požrešno metodo lahko uporabimo tudi, če niso izpolnjene zahteve *

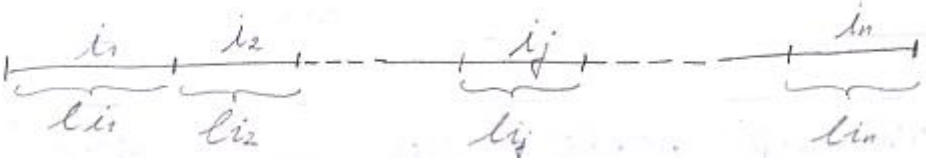
Toda: Rešitev S, ki jo dobimo, je v splošnem suboptimalna.

11 Razvrščanje zapisov na magnetni trak

Uvod: Danih je n zapisov z dolžinami $l_1, l_2, ...l_n$. Zapisati jih želimo na magnetni trak v takšnem zaporedju, da bo povprečen čas branja enega zapisa najmanjši.

Predpostavke: Ko končamo brati zapis, moramo brati vse od začetka traku do konca želenega traku. Trak se pomika s konstantno hitrostjo. Verjetnost, da bo treba brati zapis i, je enaka pri vseh i.

Problem: Naj bodo zapisi na traku v zaporedju $i_1, i_2, ...i_n$.



Da preberemo j-ti zapis (zapis ij) moramo prebrati vse pred njim, zato je čas za branje j-tega sorazmeren vsoti $l_{i_1} + ... + l_{ij}$:

$$t_j = c \sum_{k=1}^{j} l_{i_k}$$

Naj bo p_j verjetnost, da bo zahtevano branje j. zapisa. Povprečen (oz. pričakovani) čas za branje enega zapisa je torej:

$$\bar{t} = \sum_{j=1}^{n} p_j \cdot t_j$$

Pri nas je po predpostavki $p_j = \frac{1}{n}$ za vsak j:

$$t = \frac{1}{n} \cdot \sum_{j=1}^{n} t_j = \frac{c}{n} \cdot \sum_{j=1}^{n} \sum_{k=1}^{j} l_{i_k} (\text{povprečen čas za branje enega zapisa})$$

Naša naloga: Poiskati zaporedja $i_1, i_2, ..., i_n$, ki minimizira zgornji izraz.

Razmislek: Krajši zapisi naj bodo na začetku.
Rešitev naloge je zaporedje $i_1, i_2, ..., i_n$, za katerega je $l_{i_1} \leq l_{i_2} \leq ... \leq l_{i_n}$.

Algoritem za razvrstitev zapisov je požrešen, ker k že zapisanim (množica S) vedno "požrešno"doda najkrajšega med preostalimi.

Ali je požrešna metoda res optimalna?

Dokaz (s protislovjem): Če $i_1 ... i_n$ ni optimalen je pa $j_1, j_2, ... j_n$.
Qed.

12 Razvrščanje poslov na enem stroju

Uvod:

- n poslov
- za i-tega poznamo vrednost v1 in čas izgotovitve ti
- na voljo je 1 stroj
- za vsak posel rabi enoto časa

Naloga: Iščemo podmnožico $J \subseteq \{1, 2, ...n\}$ poslov (in tudi vrstni red njihovega izvajanja), ki je (vse) pravočasno končano in maksimizira vrednost $\sum_j v_j$ končanih poslov.

Definicija: Množica $J \subseteq \{1, 2, ..., n\}$ je dopustna, če jo je možno končati v vsaj enem vrstnem redu.

Kako ugotovimo ali je J dopustna?
Ali moramo preveriti vsa možna zaporedja elementov iz J?
Teh je $|J|!$ Ne! Zadošča, da preverimo ti. požrešno zaporedje.

Definicija: Požrešno zaporedje (poslov iz $J = j_1, ..., j_k$) je permutacija $(i_1, ..., i_k)$ teh poslov za katero je $ti1 \leq ti2 \leq ... \leq tik$.

Izrek: Množica poslov J je dopustna $\Leftrightarrow$ požrešno zaporedje (teh poslov) je dopustno.

Kako naj gradimo J? Poskušajmo požrešno.

Izrek: Optimalno množico J dobimo tako (na začetku $J = \emptyset$), da ji v vsakem koraku dodamo posel z največjo vrednostjo med preostalimi posli, pri čemer mora J ostati dopustna. (vrstni red izvajanja opisuje požrešno zaporedje).
Dokaz: knjiga str. 106 Qed.

Algoritem:

Vhod: množica poslov 1, 2, ... n, Vi, ti za vsak i
Izhod: množica J poslov, ki je dopustna in ima max. vrednost
Metoda: požrešna
Algoritem:

```
1 (p1,p2,...pn) = urediPoslePoVelikosti; //torej, da velja
2 J = 0; V = 0;                          //Vp1 >= Vp2 >= ... >= Vpn
3 for k = 1 to n do
4     if dopustna (J unija {pk}) then
5         J = J unija {pk};
6         V = V + Vpk
7     end if
8 end for
```

Del VI
Linearno programiranje

13 Problem maksimalnega pretoka

Intuitivno

- izvor
- tekočina v omrežje
- ponor v katerem ponika tekočina
- tekočina se ne izgublja drugje v omrežju
- celoten pretok od izvora do ponora je odvisen od omrežja

Kolikšen je največji možen pretok?

Natančneje Dan je označen usmerjen graf G(V, E):

- $V = 1, 2, ...n$ so vozlišča pri čemer je:
  - 1 je izvor
  - n je ponor
- $E \subseteq V \times V$ usmerjene povezave
- po cevi $(i, j) \in E$ lahko teče tekočina le od i do j (kot v žilah)

- vsaka cev $(i,j) \in E$ ima kapaciteto $c_{ij} (\in \Re^+) \Longrightarrow$ največji možni pretok po (i,j)

Denimo da od izvora priteka V enot tekočine na sekundo v omrežje. Ta se razporedi po ceveh omrežja. Pretok skozi povezavo $(i,j) \in E$ naj bo x_{ij}.

Lastnosti

- Pretok ne more biti večji od kapacitete pretoka.

$$0 \leq x_{ij} \leq c_{ij} \tag{10}$$

- $\Longrightarrow$ i $\Longrightarrow$: $\sum_j x_j i = \sum_j xij$
  Vse kar v i priteče iz njega tudi odteče. Izjemi sta:
  - i = 1
  - i = n

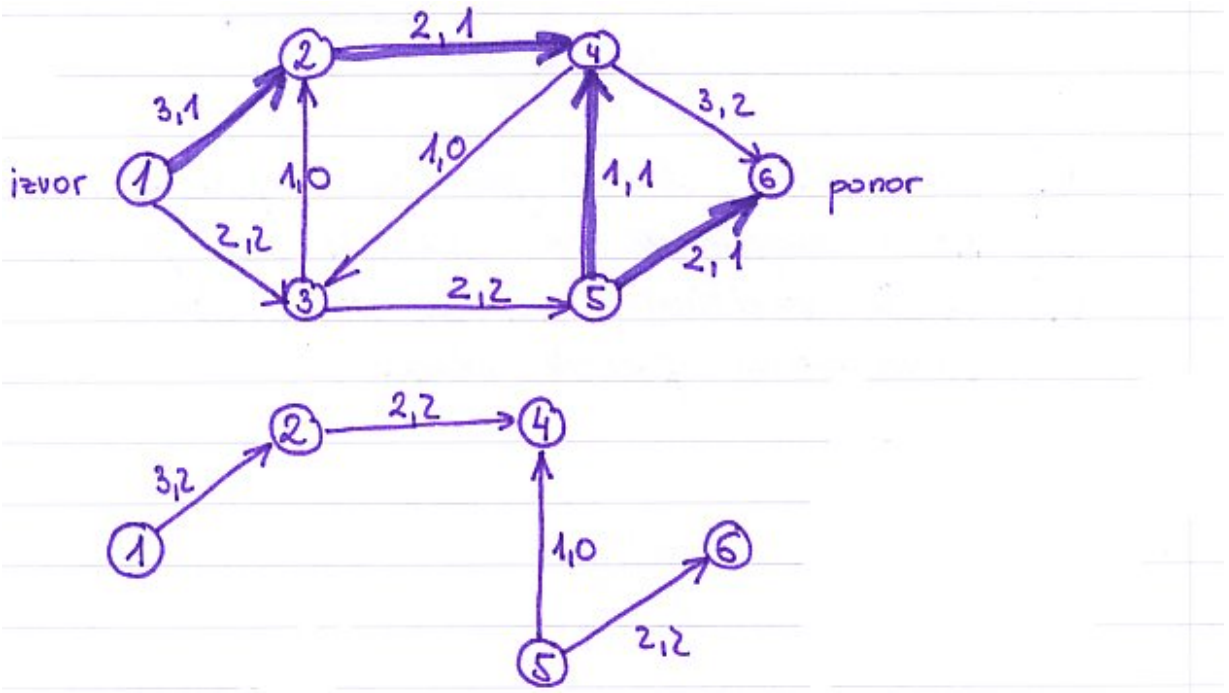
  Povedano še drugače:

$$\sum_j x_{ji} - \sum_j x_{ij} = \begin{cases} 0, \text{ če } i \neq 1, n \\ -v, \text{ če } i = 1 \\ v, \text{ če i=n} \end{cases} \tag{11}$$

- količina V predstavlja celoten pretok
- nabor $\chi = (x_{ij})$, ki izpolnjuje točki (1) in (2) opisuje razporeditev V po omrežju

1. **Ali bi bil celotni pretok V med izvorom in ponorom lahko še večji?**
2. **Kakšen je maksimalni pretok skozi omrežje ($v_{\mathbf{max}}$)?**
3. **Kateri je največji V, za katerega še obstaja porazdelitev , ki zadošča enačbi 10 in 11 in kakšna je ta porazdelitev?**

Primer



Na tej poti podobnega posega ne moremo ponoviti. Pot je **zasičena**.

Formalizirajmo Naj bo P neka "neurejena"pot iz 1 v n (na njej so povezave ne glede na svojo usmeritev).

Pozitivna povezava Povezava je **pozitivna**, če kaže v smeri ponora.

Negativna povezava Povezava je **negativna**, če kaže v smeri izvora.

Zasičenost povezave **Pozitivna povezava** je zasičena, če je njen $x_{ij} = c_{ij}$. **Negativna povezava** je zasičena, če je njen $x_{ij} = 0$.

Zasičenost poti Pot P je **zasičena**, če vsebuje vsaj eno zasičeno povezavo. Če je P **nezasičena**, imajo vse pozitivne povezave $x_{ij} < c_{ij}$ in vse negativne povezave $x_{ij} > 0$. Nezasičeno pot P lahko **zasičimo** (kot v primeru) in povečamo celoten pretok skozi omrežje.

1. **Za koliko lahko povečamo pretok?**
2. **Kako najdemo vse nezasičene poti?**
3. **Če jih zasičimo vse, ali tako dobimo maskimalni pretok v_{max}?**

Pretok preko P in zato celoten pretok čez omrežje lahko povečamo za

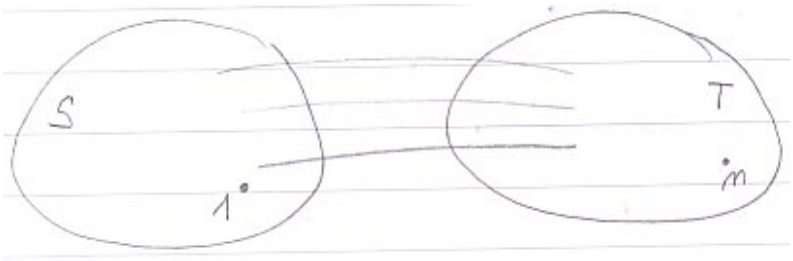
$$min\{min(x_{ij}), min(c_{ij} - x_{ij})\}.$$

Zakaj?

Kvečjemu za toliko lahko negativne povezave zmanjšajo svoj "nasprotni tok"oz. kvečjemu za toliko lahko pozitivne povezave povečajo svoj tok.

Pri dokazovanju kasnejših izrekov bomo rabili pojem prereza:

Definicija: (S, T) je $(1, n)$ prerez omrežja $G(V, E)$, če: $S \cup T = V \wedge S \cap T = \emptyset$ in $1 \in S$ in $n \in T$.



Definicija: Kapaciteta c(S, T) prereza (S, T) je c(S, T)=(?suma po i(?je element)S,j(?je element)T)cij.
Komentar: največ toliko lahko S pošilja v T.

Izrek: Vse poti P iz izvora do ponora so zasičene $\Rightarrow$ celoten pretok je maksimalen.

Zamisel za metodo: Po vrsti zasititi vse nezasičene poti. Ker je poti končno mnogo, v končnem času dobimo maksimalen pretok.

Toda: To ni čisto res!! Odvisno je od števil c_{ij}.
Namreč: za nekatere irracionalne vrednosti cij, se metoda ne konča v končnem mnogo korakih (Ford - Fulkerson).

Toda: Če so vrednosti c_{ij} cela števila, potem je predlagana metoda OK.

$\Downarrow$

Predlagana metoda vrne maksimalni pretok, ki je tudi celo število.

13.1 Algoritem ("gradniki"):

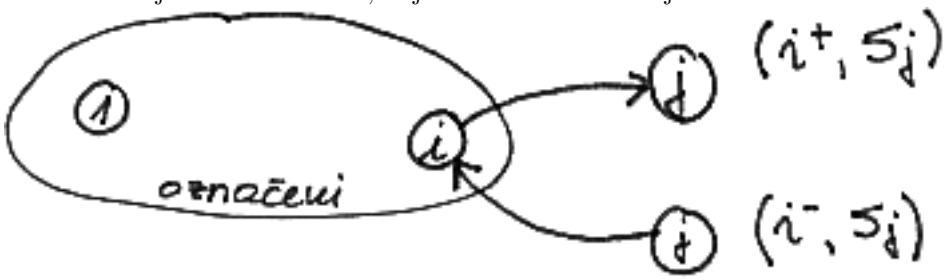
Zamisel: Začni v izvoru in postopoma označuj ostala vozlišča, dokler ne označiš ponora. Oznake naj bodo take, da bo možno iz njih razkriti nezasičene poti. Nezasičeno pot zasititi.
Vsako vozlišče i je lahko:

- neoznačeno
- označeno
  - nepregledano (ima neoznačenega soseda j)
  - pregledano (vsi sosedje od i so označeni)

Začetek Izvor 1 je označen, vendar nepregledan. Njegova oznaka je posebna in stalna (-,oo). Ostala vozlišča so neoznačena.

Označevanje

- naj bo i označeno in nepregledano vozlišče, j pa njegov neoznačen sosed.
- temu sosedu j dodelimo oznako, ki je odvisna od smeri i-j.



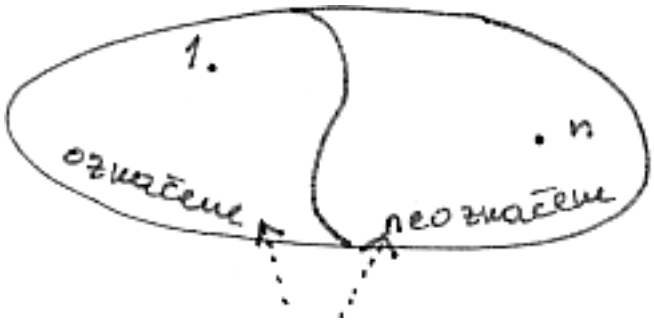
- prva komponenta bo omogočala, da izsledimo prednika točke j (torej točko i) na poti nazaj v 1
- druga komponenta Sj je definirana:

$$\delta_j = \begin{array}{l} min\{Si, Cij - Xij\}, \text{ če } i \Rightarrow > j \\ min\{si, Xij\}, \text{ če } i \Leftarrow j \end{array}$$

  Sj pove, da bi vzdolž poti $1 - ... - i - j$ lahko povečali pretok za δ_j.

Vidimo:

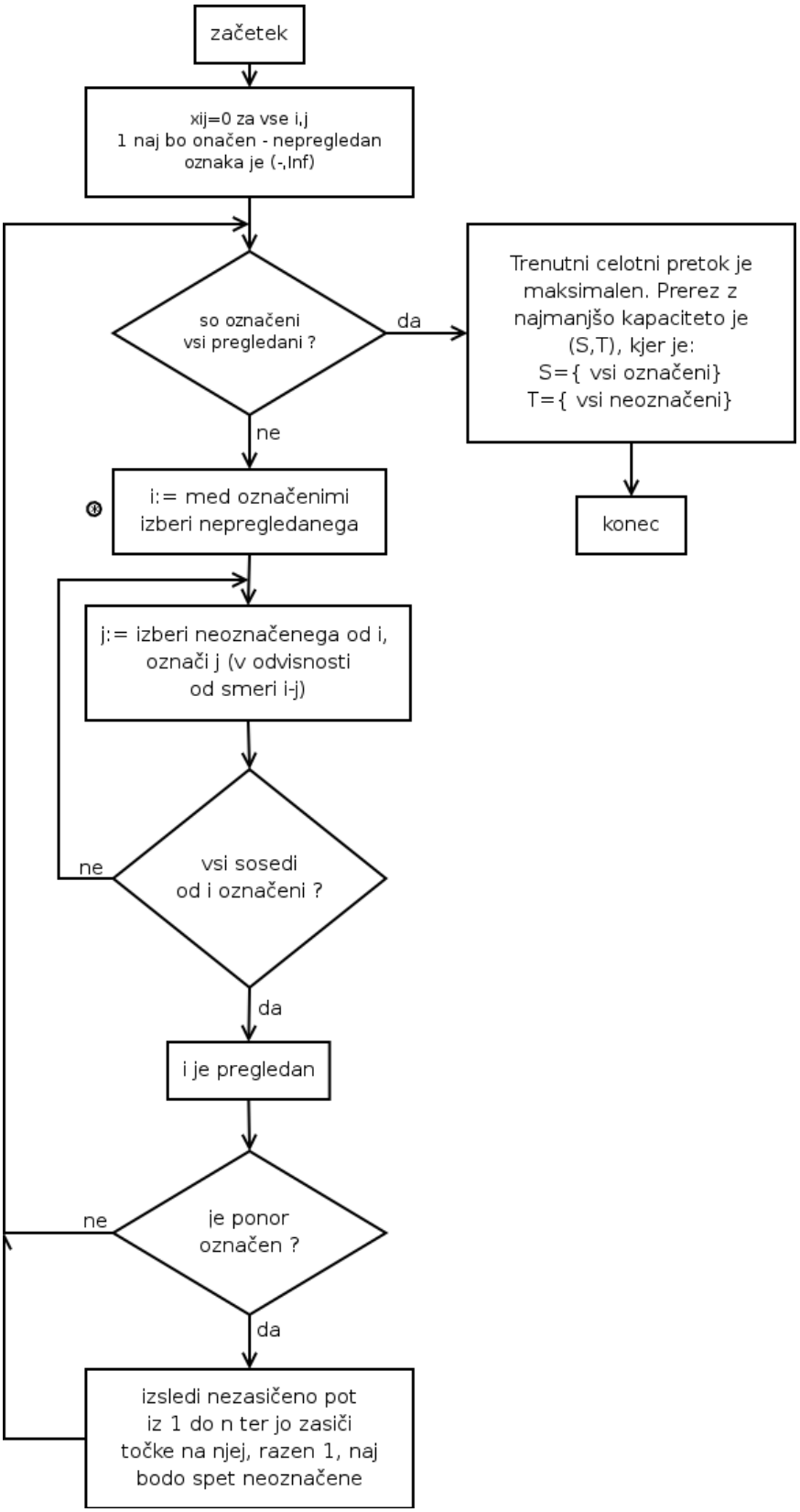
- ko uspemo označiti ponor $[mi]$ in je v njegovi oznaki $Su > 0$. smo našli neoznačeno pot
- to pot zasitimo tako, da se od [mi] pomikamo nazaj v 1 (s pomočjo prvih komponent oznake)
- to pot zasitimo tako, da:
  1. pozitivnim povezavam (vzdolž te poti!) njihove x_{ij} povečamo za δ_n
  2. negativnim povezavam njihove Xij zmanjšamo za Su
- oznake v teh točkah na poti (razen v 1) zbrišemo. Razlog: poskušali bomo najti še kakšno drugo pot iz 1 do n, pri tem nam te oznake ne bodo koristile
- zaključimo tedaj, ko nikakor ne moremo več označiti ponora n. Kdaj je to? Tedaj, ko so točke razdeljene v dve množici:



Slika 14: prerez (S,T) je prerez z najmanjšo kapaciteto C(S,T)

13.2 Ford-Fulkersonov algoritem (diagram)

Vhod: omrežje $G(V, E)$, celoštevilčne kapacitete c_ij
Izhod: maksimalen pretok čez G od 1 do n
Metoda: Ford & Fulkerson


začetek
xij=0 za vse i,j
1 naj bo onačen - nepregledan
oznaka je (-,Inf)
so označeni
vsi pregledani ?
da
Trenutni celotni pretok je
maksimalen. Prerez z
najmanjšo kapaciteto je
(S,T), kjer je:
S={ vsi označeni}
T={ vsi neoznačeni}
konec
ne
i:= med označenimi
izberi nepregledanega
j:= izberi neoznačenega od i,
označi j (v odvisnosti
od smeri i-j)
vsi sosedi
od i označeni ?
ne
da
i je pregledan
je ponor
označen ?
ne
da
izsledi nezasičeno pot
iz 1 do n ter jo zasiči
točke na njej, razen 1, naj
bodo spet neoznačene

Časovna zahtevnost

- naj bo V končni maksimalni pretok
- do njega smo prišli z zasičenjem največ V poti (če je vsako zasičenje povečalo pretok za najmanj kar je možno: za 1, zaradi predpostavke, da so $c_{ij} \in \mathbb{N}$
- vsaka od neoznačenih poti je dolga kvečjemu $m = |E|$
- vsaka povezava se je tej poti med njeno gradnjo dodala z nekim konstantnim številom operacij (med označevanjem)
- časovna zahtevnost algoritma je $\Theta(m * v)$, kjer je $m = |E|$

Opomba:

- če ne bi veljala predpostavka $c_{ij} \in \mathbb{N}$, obstajajo patološki primeri z iracionalnimi c_{ij}, kjer ne bi mogli zasititi vseh poti v končno mnogo korakih (Ford & Fulkerson)
- ni nam všeč, da je časovna zahtevnost odvisna samo od rezultata, ki ga računamo
- to sta izboljšala Edmonds & Karp na $\Theta(m^2; n)$ z enostavnim posegom: "vozlišča naj postanejo pregledana v istem vrstnem redu, kot so postala označena (glej *)".

Izboljšave algoritma

Raziskovalec/-ci	**Leto**	**Čas. zahtevnost**
Ford-Fulkerson	1956	$\Theta(m * v)$
Edmonds-Karp	1969	$\Theta(m^2 * u) m = \|E\|$
Dinič	1970	$\Theta(m * u^2 u = \|V\|$
Karzanov	1973	$\Theta(n^3)$
Chomsky	1976	$\Theta([korenodm] * n^2)$
Malkotra et. al.	1978	$\Theta(n^3)$
Galil	1978	$\Theta(n^(5/3); m(2/3))$
Galil & Naamod	1979	$\Theta(m * n * log^2(n))$
Sleator & Tarjan	1980	$\Theta(m * n * log(n))$
Goldberg & Tarjan	1985	$\Theta(m * n * log(n^2/m))$

(→ možnost raziskovalne naloge)

Iztočnica: Ta problem maksimalnih pretokov lahko rešujemo z linearnim programiranjem.

14 Linearno programiranje

- problem maksimalnega pretoka sodi med probleme LP
- to so pomembni problemi, ker
  - so pogosti v praksi
  - poznamo zanje učinkovite algoritme

Primer

- imamo $100m^2$ kartona
- radi bi izdelovali embalažo - male in velike škatle
- za malo škatlo potrebujemo $1/2m^2$ kartona
- za veliko škatlo pa $3/4m^2$ kartona
- malo škatlo prodamo za 1 SIT, veliko pa za 2 SIT

Koliko malih (x_1) in koliko velikih (x_2) izdelati, da bo dobiček s prodajo maksimalen?
Iščemo število x_1 in x_2, ki izpolnjuje pogoje:

$$\begin{aligned} x_1 &\leq 0 \\ x_2 &\leq 0 \\ 0,5 \cdot x_1 + 0,75 \cdot x_2 &\leq 100 \end{aligned}$$

in ki maksimizirata ciljno funkcijo.

$$1x_1 + 2x_2 ....\text{dobiček}$$

V matrični obliki zapišemo:

- iščemo vektor-stolpec $\vec{x} = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$, ki zadovoljuje pogoju

$$\begin{bmatrix} -1 & 0 \\ 0 & -1 \\ 0,5 & 0,75 \end{bmatrix} \cdot \vec{x} \leq \begin{bmatrix} 0 \\ 0 \\ 100 \end{bmatrix}$$

  in ki minimizira produkt $[1, 2]\,\vec{x}$

Splošno: Naj bo V_n n-dimenzionalni vektorski prostor nad obsegom K.

Definicija problema linearnega programiranja Naj bo

- A matrika reda $m \times n$
- $\vec{b}$ vektor dolžine m
- $\vec{c}$ vrstični vektor dolžine n

Poiskati je treba $\vec{x} \in V_n$, ki:

- izpolnjuje pogoj $A\vec{x} \leq \vec{b}$
- in ki maksimizira produkt $\vec{c} \cdot \vec{x}$ (ciljna funkcija)

Vprašanja:

1. **Kakšna je množica** $K = \left\{\vec{x} \in V_n | A\vec{x} \leq \vec{b}\right\}$**?**
   Konveksne poliedrske množice (KPM).
2. **Kako opišemo tiste elemente iz K, kjer ciljna funkcija** $\vec{c}\vec{x}$ **doseže maksimalno vrednost?**
3. **Kako te elemente učinkovito poiščemo?**

Definicija Naj bo $\vec{a} \in V_n$ in $\vec{b} \in K$.
Tedaj je:

- $\vec{x} \in V_n | \vec{a}\vec{x} = \vec{b}$ hiperravnina prostora V_n
- $\vec{x} \in V_n | \vec{a}\vec{x} \leq \vec{b}$ zaprti podprostor prostora V_n
- $\vec{x} \in V_n | \vec{a}\vec{x} < \vec{b}$ odprti podprostor prostora V_n

	$n=2$	$n=3$	$n=4$	
$\vec{a}\vec{x} = \vec{b}$	premica	ravnina	3-dim. prostor	...
$\vec{a}\vec{x} \leq \vec{b}$	zaprta polravnina	zaprti podprostor	$\vdots$	
$\vec{a}\vec{x} < \vec{b}$	odprta polravnina	odprti podprostor		

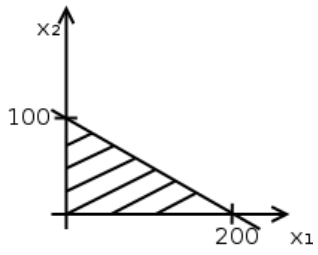
Primer

n = 2

$-x_1 \leq 0$...ordinata (x_2) in vse desno od nje

$-x_2 \leq 0$

$0.5x_1 + 0.75x_2 \leq 100$

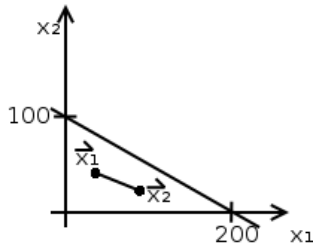


Opazimo Točke, ki zadoščajo vsem trem pogojem, ležijo v preseku vseh treh pripadajočih podprostorov. **Velja splošno!!!**

Trditev Naj bo A matrika $m \times n$ in $\vec{b}$ vektor dolžine m. Množico $K = \vec{x} \in V_n | A\vec{x} \leq \vec{b}$ je presek največ m zaprtih polprostorov (od V_n).

Kateri so ti polprostori?
To so polprostori $\vec{x} \in V_n | \vec{a}\vec{x} = b$, kjer je $\vec{a}$ vrstica matrike A, b pa je ustrezni skalar v $\vec{b}$.
Množica K se imenuje **poliedrska množica**.

Motivacija Izberimo 2 poljubna vektorja $\vec{x_1}$ in $\vec{x_2}$, ki sta v K.

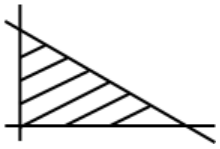


Opazimo V zasenčenem delu so tudi vektorji, ki so "med njima". Vsakega lahko zapišemo kot $(1-\lambda)\vec{x_1} + \lambda\vec{x_2}$, kjer je $\lambda \in [0,1]$.

Torej Če sta x_1 in x_2 rešitvi, potem je rešitev tudi $(1-\lambda)\vec{x_1} + \lambda\vec{x_2}$. Velja splošno.

Trditev Če velja $A\vec{x_1} \leq \vec{b}$ in $A\vec{x_2} \leq \vec{b}$, potem velja $A[(1-\lambda)\vec{x_1} + \lambda\vec{x_2}] \leq \vec{b}$. To pomeni, da je množica K **konveksna**, zato jo imenujemo **konveksna poliedrska množica**.

Motivacija V prejšnjem primeru je bila KPM omejena.Če pa bi odpravili npr. 3. pogoj, bi bila neomejena.



Kako strogo definirati neomejeno KPM?

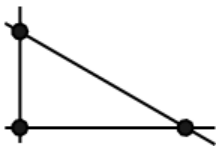
Definicija KPM je **neomejena**, če vsebuje vsaj en poltrak.

14.1 Ekstremne točke konveksne poliedrske množice

Motivacija V zgornjem primeru ($n = 2$) ima KPM tudi ekstremne točke. Za vsako od njih velja:

- je v KPM
- je na presečišču $n = 2$ mejnih premic ($\equiv$ mejnih hiperravnin)

Velja tudi splošno.



Definicija Naj bo dana KPM $K = \vec{x} \in V_n | A\vec{x} \leq \vec{b}$. Točka $\vec{x}$ je **ekstremna** točka KPM K, če velja:

1. $A\vec{x} \leq \vec{b}$ (torej $\vec{x}$ je v K)
2. $B\vec{x} \leq \vec{b_B}$, kjer je B:
   - podmatrika matrike A reda $n \times n$
   - $\vec{b_B}$ je ustrezni podvektor vektorja $\vec{b}$

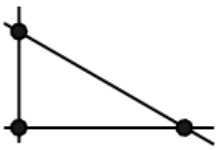
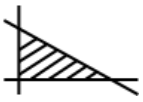
Opomba Če je $\vec{a}$ vrstica matrike B, enačba $\vec{a}\vec{x} = b_B$ določa eno od n mejnih hiperravnin, na katerih presečišču je ekstremna točka. Če je KPM neomejena, štejemo za ekstremno točko tudi "neskončno točko".

Ekstremna točka je torej rešitev sistema $B\vec{x} = \vec{b_B}$ (poleg $A\vec{x} \leq \vec{b}$), kjer je B podmatrika od A.

Kakšna je B?
Če je singularna, potem $B\vec{x} = \vec{b_B}$ niso rešitve oz. rešitev je "neskončna točka KPM".

Predpostavka $\forall n \times n$ podmatrika B matrike A naj bo nesingularna. Če matriki A dodamo še vse pgooje $-x_i \leq 0$, zato je gotovo $m \geq n$.

Koliko je ekstremnih točk v KPM $K = \vec{x} \in V_n | A\vec{x} \leq \vec{b}$, kjer je A vedno $m \times n$?
Podmatrik reda $m \times n$ je $\begin{pmatrix} m \\ n \end{pmatrix}$. Ekstremnih točk je največ $\begin{pmatrix} m \\ n \end{pmatrix}$.

Motivacija V primeru  opazimo, da je vsaka mejna premica tudi neka KPM (toda v prostoru V_1), ekstremna točka na premici pa je hkrati tudi ekstremna točka v 2D prostoru (V_2) $\Rightarrow$ 

Trditev Naj bo dana KPM $K = \vec{x} \in V_n | A\vec{x} \leq \vec{b}$ in naj bo $\vec{a}$ vrstica v A. Tedaj je hiperravnina $H = \vec{x}; \vec{a}\vec{x} = b$ tudi neka KPM K' prostora V_{n-1} (torej $H = K'$). Če je $\vec{x}$ ekstremna točka množice K' je tudi ekstremna točka množice K.

14.2 Maksimum ciljne funkcije na KPM

Na katerih točkah iz KPM $K = \vec{x} \in V_n | A\vec{x} \leq \vec{b}$ doseže ciljna funkcija $\vec{c}\vec{x}$ svoj maksimum? (posplošitev funkcije je $\vec{c}\vec{x} + \delta$)
V KPM je običajno neskončno elementov, ponavadi celo neštevno (kontinuum).

1. **Ali lahko ciljna funkcija doseže svoj maksimum na neskončno mnogo točkah?**

2. **Ali bo treba pregledati neskončno mnogo točk v K, da najdemo tisto, kjer $\vec{c}\vec{x}$ (oz. $\vec{c}\vec{x} + \delta$) doseže maksimum?**

Zadošča, da pogledamo le ekstremne točke v KPM K.

14.2.1 Izrek o maksimumu ciljne funkcije na KPM

Naj bo dana KPM $K = \vec{x} \in V_n | A\vec{x} \leq \vec{b}$ in ciljna funkcija $f = \vec{c}\vec{x} + \delta$. Funkcija f doseže svoj maksimum v neki ekstremni točki množice K.

Dokaz indukcija po n:
glej knjigo str. 122. Qed.

Algoritem Poišči vrednost f v vsaki ekstremni točki. Kot rezultat vrni točko, v kateri je bila vrednost f največja.

Časovna zahtevnost Ekstremnih točk je $\binom{m}{n}$. Izračun f v vsaki točki zahteva m množenj in n seštevanj (izračun skalarnega produkta). Vseh operacij v algoritmu bi bilo $\binom{m}{n} \cdot 2n$. Algoritem bi imel časovno zahtevnost $\Theta\left(\binom{m}{n} \cdot n\right)$. To je lahko zelo veliko:

$$\begin{aligned}\binom{m}{n} &= \tfrac{m(m-1)\ldots(m-n+1)}{n(n-1)\ldots 1} = \\ &= \tfrac{m}{n} \cdot \tfrac{m-1}{n-1} \cdots \tfrac{m-n+1}{1} \geq \\ &\geq \binom{m}{n}^n = \\ &= /*\text{če vzamemo } m = 2n*/ = \\ &= 2^n\end{aligned}$$

Bolj splošno: ker je $m \geq n$ je $m = c \cdot n$, kjer $c \geq 1$, zato $\binom{m}{n}^n = c^n$. Za $c >1$ je to **eksponentno mnogo**.

1. **Kako ta naivni algoritem izboljšati?**

   (a) **Ali obstaja nek "pameten"vrstni red v katerem se splača pregledovati ekstremne točke?**

      i. **Morda bi se izplačalo obiskovati naslednjo ekstremno točko?**

         A. **Kako lahko hitro najdemo sosednjo ekstremno točko?**

         B. **Koliko je sosednjih točk oz. med koliko sosedami bomo izbirali?**

         C. **Kako vemo kdaj lahko obiskovanje končamo, čeprav nismo obiskali vseh ekstremnih točk?**
         Drugače: ali obstaja lokalni kriterij za ugotovitev, da smo našli globalni maksimum?

Kdaj lahko obiskovanje ekstremnih sosed končamo?

14.2.2 Izrek o lokalnem pogoju za globalni maksimum

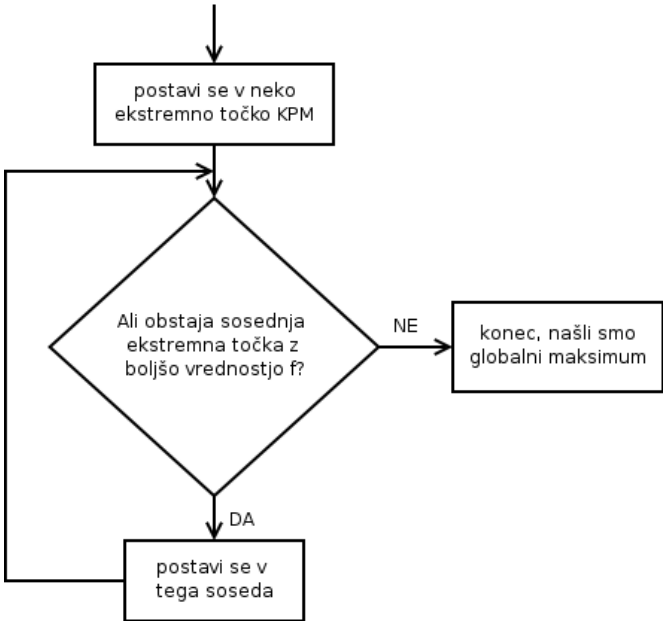
Naj bodo dani:

- KPM $K = \vec{x} \in V_n | A\vec{x} \leq \vec{b}$,
- ciljna funkcija $f(\vec{x}) = c\vec{x} + \delta$ in
- ekstremna točka $\vec{x}_B$ (z ustrezno matriko B).

Če je vrednost f v vseh ekstremnih točkah manjša ali enaka vrednosti $f(\vec{x}_B)$, potem f doseže v $\vec{x}_B$ svoj globalni maksimum.

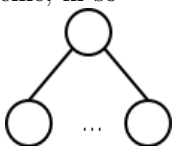
14.2.3 Simpleksni algoritem (G. Dantzig)

Algoritem Simplex



Del VII
Dinamično programiranje in najcenejše poti

Metoda rekurzivnega razcepa Problem razbijemo na podprobleme, ki so neodvisni, zato jih rešujemo ločeno. Delne rešitve sestavimo v končno.



Primer: Quicksort
Če problemi niso neodvisni uporabljamo metodo **dinamičnega programiranja**.

Tu rešitve podproblemov običajno hranimo dalj časa, saj so običajno večkrat potrebne. Reševanje zato poteka od najmanjših podnalog do čedalje večjih.

Dinamično programiranje temelji na ti. **načelu optimalnosti**.
Naj bo $O_1, O_2, ..., O_n$ zaporedje odločitev pri iskanju rešitve. Pravimo, da je to **zaporedje optimalno**, če nas privede do optimalne rešitve.

Načelo optimalnosti Vsako podzaporedje optimalnega zaporedja odločitev, je tudi optimalno.

15 Problem 0-1 nahrbtnika

Kako nahrbtnik z omejeno nosilnostjo napolniti s čimvrednejšim plenom?

Definicija Dana je množica $X = 1, 2, ..., n$ predmetov, za vsak predmet pa je podana še njegova teža $t_i \in \mathbb{Z}^+$ in vrednost $c_i \in \mathbb{Z}^+$ ter nosilnost $b \in \mathbb{N}$ nahrbtnika.

Poiskati je treba podmnožico $Y \subseteq X$... plen, za katero je:

- $\sum_{i \in Y} t_i \leq b$ ni pretežek
- $\sum_{i \in Y} c_i$ pa največja možna (plen je najvrednejši)

Opomba Problem lahko zapišemo kot problem linearnega programiranja (celoštevilskega). Poiskati je treba števila $x_1, x_2, ..., x_n$, kjer je $x_i \in 0, 1$, ki:

- izpolnjujejo pogoj $\sum_{i=1}^{n} t_i x_i \leq b$
- maksimizira ciljno funkcijo $\sum_{i=1}^{n} c_i x_i$

Mi pa bomo ta problem rešili z dinamičnim programiranjem.

Zamisel Naj bo $C = \sum_{i=1}^{n} c_i$ skupna vrednost vseh izdelkov. Naj bo $c \in [0, c]$. Naj bo $x_i = 1, 2, ..., i$. Vsaka podmnožica od x_i ima svojo težo in vrednost. Opazujemo le tiste podmnožice od x_i, ki so težke kvečjemu $\leq b$. Med temi podmnožicami včasih nobena ni vredna natanko c, včasih pa je lahko celo več takih. Tedaj je ena med njimi **najlažja**. To podmnožico označimo s $S_i(c)$.

Definicija

$$S_i(c) = \begin{cases} \text{najlažja med podmnožicami od } x_i, \text{ ki so vredne natanko } c \text{ in} \\ \quad \text{težke kvečjemu } b \text{ (kadar taka obstaja)} \\ \uparrow \text{ (kadar take podmnožice ni)} \end{cases}$$

Zasnova algoritma Računaj množice $S_n(c), S_n(c-1), ...$ in končaj pri prvi, npr. $S_n(c*)$, ki je definirana. Ta množica je rešitev Y problema nahrbtnika.

Iskanje $S_n(c*)$ bomo realizirali z dinamičnim programiranjem ("od spodaj navzgor").

Definicija Če je $S_i(c)$ definirana ($\uparrow$), naj bo $T_i(c)$ njena **teža**.

Kako izračunati $S_i(c)$ in $T_i(c)$?

- $i = 1 : S_1(0) = \emptyset$, ker je $\emptyset$ edina podmnožica od množice $X_1 = 1$ z vrednostjo 0. $S_1(c_1) = 1$ $S_1(c) \uparrow$, če je $c > 0 \wedge c \neq c_1$, ker $X_1 = 1$ nima podmnožice s tako ceno c.

  Njihove teže so:

  $T_1(0) = 0$
  $T_1(c_1) = t_1$
  $T_1(c) =$ /po dogovoru/ $= 1 + \sum_1^n t_i$ (ker $S_1(c)$ ni definirana)

- $i \geq 2$: Recimo, da je $S_i(c)$ že izračunana. Tedaj bodisi **vsebuje** element i ali pa ga ne. Podrobneje:

  $i \in S_i(c)$. Tedaj $S_i(c)$ sestavljata i ter najlažja podmnožica od X_{i-1}, ki je vredna $c - c_i$. Torej: $S_i(c) = i \cup S_{i-a}(c - c_i)$ s težo $T_i(c) = t_i + T_{i-1}(c - c_i)$. Velja pod pogojem, da:

  $$c - c_i \geq 0 S_{i-1}(c - c_i) \downarrow t_i + T_{i-1}(c - c_i) \leq b$$

  $i \notin S_i(c)$. Tedaj: $S_i(c) = S_{i-a}(c)$ s težo $T_i(c) = T_{i-1}(c)$.

Toda za $S_i(c)$ je morala biti izbrana **lažja** od obeh možnosti. Zato je teža množice $S_i(c)$:

$$T_i(c) = min T_{i-1}(c), t_i + T_{i-1}(c - c_i).$$

15.1 Dinamična implementacija izračuna množice $S_i(c)$ od spodaj navzgor

Najprej izračunamo "manjše"množice in njihove teže, npr. $S_{i-1}(c - c_i)$ in $S_{i-1}(c)$ in $T_{i-1}(c - c_i)$ in $T_{i-1}(c)$, ki jih nato sestavimo po zgornjih pravilih v "večje"množice, tj. v $S_i(c)$ in $T_i(c)$. Tako napreduje po naraščajočih i, od 1 do n.

15.2 Algoritem

Vhod: $X, n, t_i, c_i, bza i = 1, 2, ..., n$
Izhod: Y, c^* (plen in njegova vrednost)

```
1  begin
2      C := ∑_{i=1}^{n} c_i;
3      for c:= 0 to C do
4          S_1(c) :=↑;
5          T_1(c) := 1 + ∑_{i=1}^{n} t_i;
6      end
7      S_1(0) = ∅; T_1(0) = 0
8      S_1(c_1) = 1, T_1(c_1) = t_1;
9      for i=2 to n do
10         for c=0 to C do
11             if c − c_i ≥ 0 in S_{i−1}(c − c_i) ↓ in
12               t_i + T_{i−1}(c − c_i) ≤ b in
13               t_i + T_{i−1}(c − c_i) ≤ T_{i−1}(c)
14             then S_i(c) = i ∪ S_{i−1}(c − c_i);
15                 T_i(c) = t_i + T_{i−1}(c − c_i);
16             else
17                 S_i(c) = S_{i−1}(c); T_i(c) = T_{i−1}(c);
18             end
19         end
20     end
21
22     c* = največji c, pri katerem S_n(c) ↓;
23     Y = S_n(c*);
24 end
```

15.3 Časovna zahtevnost

Zaradi dvojne zanke for je časovna zahtevnost $\Theta(nC)$, kjer je $C = \sum_{i=1}^{n} c_i$.

Koliko je C?
Naj bo vsaka c_i shranjena v d-bitni besedi. Torej $0 \leq c_i \leq 2^d - 1$.

V najboljšem primeru, ko je $vsak \forall c_i = 1$, je $C = n = \Theta(n)$.

V najslabšem primeru, ko je $vsak c_i = 2^d - 1$, pa je $C = n(2^d - 1) = \Theta(n2^d) = \Theta(n2^{\frac{D}{n}})$, kjer je D=nd skupna dolžina vseh podatkov $c_i, i = 1...n$. V najslabšem primeru je C eksponentno odvisen od dolžine vhodnih podatkov (D).

Pravimo, da je naš algoritem **psevdopolinomski**, ker je sicer polinomsko odvisen od C, ne pa od njegove dejanske dolžine v pomnilniku.

To je **aproksimativni algoritem** (vprašaj profesorja kaj je mislil z razlago :-S).

16 Problem najcenejših poti

To je tipičen optimizacijski problem, ki ga rešujemo s pomočjo dinamičnega programiranja. Ta problem je pomemben, ker obstaja veliko drugih optimizacijskih problemov lahko prevedemo nanj.

Definicija Dan je obtežen in usmerjen graf G(V, E, c):

- V - množica vozlišč
- E podmnožica V x V usmerjenih povezav
- $c : E \to Real$, cena

Dana je pot iz vozlišča i_z v i_k je zaporedje $i_1, i_2, ...i_n$, kjer: $i_1 = i_z, i_n = i_k, (i_j, i_{j+1}) \in E$ za $j = 1, 2, ...n-1$.

Dana je cena poti: $u_{i_z} = \sum_{j=1}^{n-1} c(i_j, i_{j+1})$. Cikel je pot, ki se konča v začetni točki.

VSTAVI SLIKO

17 Problem najcenejših poti

- med začetnim in vsemi ostalimi
- med vsemi potmi

17.1 Najcenejše poti med začetnim in vsemi ostalimi

Zahtevamo 2 pogoja:

1. začetno vozlišče je **povezano z vsemi ostalimi**
2. graf G **nima negativnih ciklov**

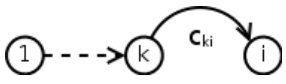
Razlog: $1 \wedge 2 \Rightarrow$ iz začetnega $\exists$ do $\forall$ vozlišča pot, ki ima končno enolično ceno

Trditev ("Načelo optimalnosti") Naj bo $i_p...i_q$ pot, ki je del najcenejše poti iz i_z do i_k. VSTAVI SLIKO

Tedaj je $i_p...i_q$ najcenejša pot iz i_p do i_q.

17.1.1 Razmišljajmo

- G(V,E,c) in veljata 1 in 2
- po definiciji naj bo u_i = cena najcenejše poti iz 1(=začetna točka) do točke i
- če je $i = 1 \Rightarrow u_1 = 0$
- če pa $i \neq 1$, potem pride najcenejša pot v i iz nekega k ($\neq i$)



Po načelu optimalnosti je to del najcenejše poti do k (z dolžino u_k). Kateri je k? Tisti za katerega je $u_k + c_{ki}$ najmanjše! Torej:

$$u_i = min_{k \neq i} u_k + c_{ki}$$

Ugotovili smo, da za rešitve $u_1, u_2, ...u_n$ veljajo enačbe:

$$u_i\{ \begin{array}{l} 0, i = 1 \\ min_{k \neq i} u_k + c_{ki}, i \neq 1 \end{array} \tag{12}$$

To so Bellmanove enačbe.

Izrek: Naj bo G(V,E,c) in veljata 1 in 2. Tedaj velja: če so $u_1, u_2, ...u_n$ rešitev problema najcenejših poti iz 1 do ostalih potem $u_1, u_2, ...u_n$ zadoščajo 12.

Kaj pa obratno?
Recimo, da so dane Bellmanove enačbe za G(V,E,c) in naj bodo $u_1, u_2, ...u_n$ njihova rešitev.

Vprašanja

1. **Kaj sploh pomenijo vrednosti** $u_1, u_2, ...u_n$**?**
2. **Ali so v kakšni zvezi s problemom najcenejših poti v G?**
3. **Ali je rešitev** $u_1, u_2, ...u_n$ **edina rešitev Bellmanovih enačb?**

Odgovori

1. S pomočjo rešitve $u_1, u_2, ...u_n$ Bellmanovih enačb lahko vozlišča grafa G uredimo v drevo. **Kako?** Vzemimo nek u_i. Poiščimo tisti j, za katerega je $u_i = u_j + c_{ji}$. Točki i in j grafa G povežimo z relacijo $j \prec i$. To storimo za vse $i \in V$. Rezultat: drevo s korenom v 1.
2. Vrednost u_i (rešitve $u_1, ...u_n$ Bellmanovih enačb) je cena neke poti iz 1 do i v grafu G.
3. Rešitev $u_1, ...u_n$ Bellmanovih enačb je edina.

Izrek Naj bo dan $G(V, E, c)$ in veljata 1 in 2. Tedaj velja: rešitev $u_1, ...u_n$ Bellmanovih enačb je enolična in je rešitev problema najcenejših poti iz 1 do ostalih v G.

Sklep Rešitev problema najcenejšh poti iz 1 do ostalih vozlišč v grafu G **je natanko** rešitev Bellmanovih enačb.

Torej Lahko se začnemo ukvarjati z reševanjem Bellmanovih enačb. Vidimo zakaj je primerna **metoda dinamičnega programiranja**. Pri računanju vrednosti u_i potrebujemo **vse ostale** u_k.

$$u_i = min_{k \neq i}\{u_k + c_{ki}\} \tag{13}$$

V nadaljevanju obravnavamo posebne primere grafov, kjer ta pogoj ($k \neq i$) **poostrimo**.

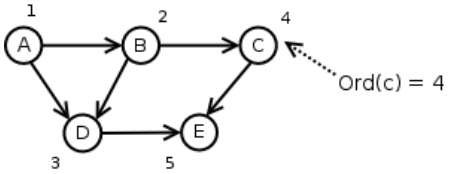
17.2 Topološko urejanje grafov

Naj bo dan usmerjen graf G(V,E)

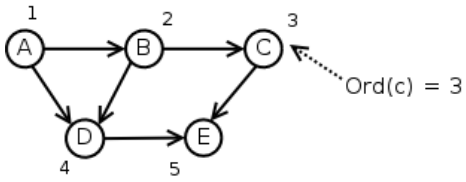
1. **Ali je mogoče njegova vozlišča označiti tako, da vsaka povezava poteka iz vozlišča z nižjo oznako v vozlišče z višjo oznako?**
   Če je to možno, je graf G topološko urejen.

Definicija Usmerjen graf $G(V, E)$ je **topološko urejen**, če $\exists$ funkcija $Ord : V \to \{1, 2, ...|V|\}$, tako da velja: $(i, j) \in E \Rightarrow Ord(i) < Ord(j)$.

Primer



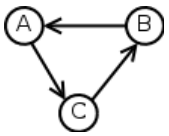
1. **Ali ima lahko graf več topoloških ureditev?**
   Lahko.
   Primer:



2. **Ali je možno vsak (usmerjen) graf topološko urediti?**
   Ne.
   Primer:

Toka	Ord
A	a
B	b
C	c

$\underline{\mathbf{a}} < c < b < \underline{\mathbf{a}}$



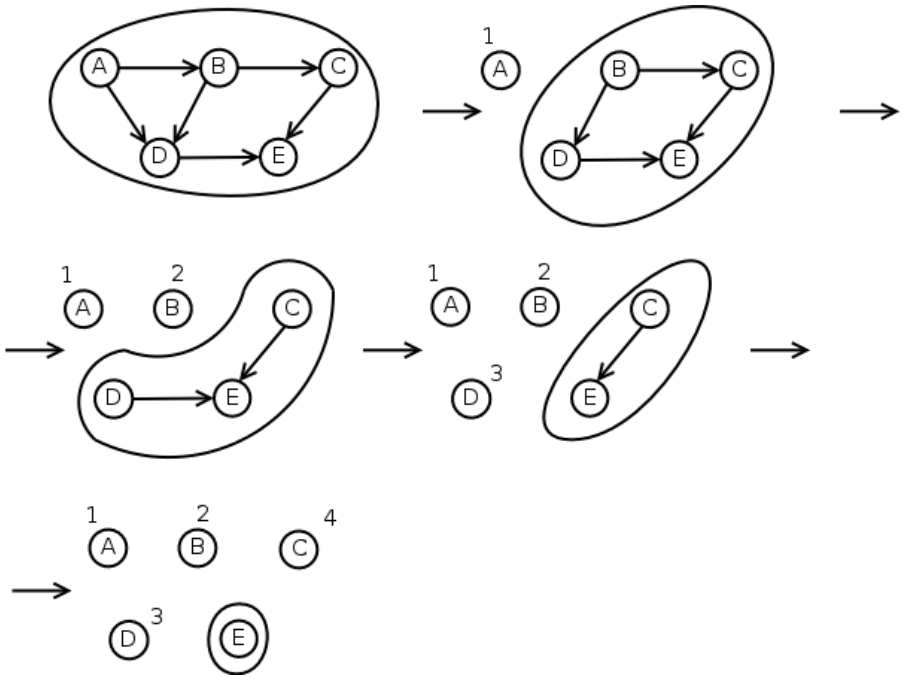
Izrek Graf G je možno **topološko urediti** natanko tedaj, ko je G **acikličen**.

Dokaz

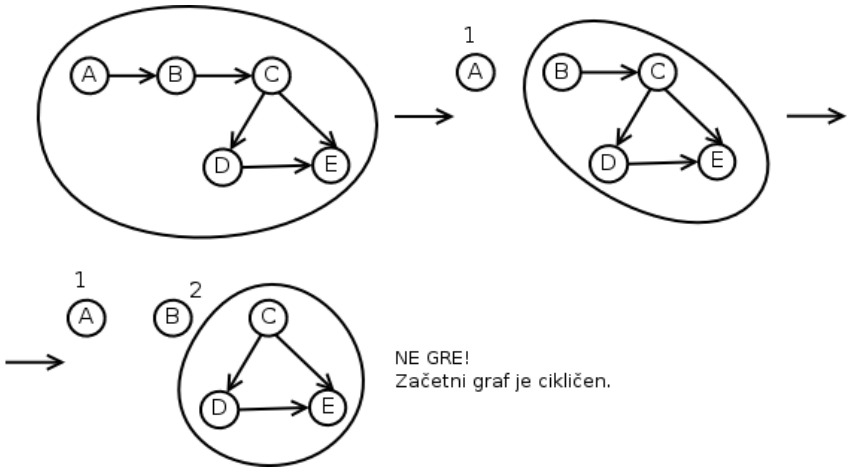
($\Rightarrow$) Če je G topološko urejen, bi obstoj cikla zahteval oznako nekega vozlišča z lastnostjo $a < a$. Torej cikla ne more biti.

($\Leftarrow$) Dokaz z indukcijo po $|V|$.
$|V| = 1$. Trivialno
Velja $|V| = k : \forall G$ z $|V| = k$ točkami lahko topološko uredimo.
$|V| = k + 1$. Ker je G acikličen, obstaja točka z vhodno stopnjo O (točka v). Točko v označimo z 1 in jo odstranimo iz G (in vse povezave, ki jo zapuščajo). Graf $G - v$ lahko topološko uredimo po indukcijski predpostavki (z oznakami $2, 3, ...|V|$).
Qed.
Na dokazu zgradimo algoritem za topološko ureidtev grafa.

Primer 1



Primer 2



17.2.1 Algoritem (Topološko urejanje grafa)

Vhod: usmerjeni graf G(V,E)

Izhod:

- DA ($\equiv$ G je acikličen) in ureditev $Ord : V \to \{1, 2, ...|V|\}$
- NE ($\equiv$ V je cikličen)

Postopek

```
1  G_t = G; ..... krcil se bo G_t
2  s = 0; ..... stevec
3  while G_t vsebue vsaj eno vozlisce z vhodno stopnjo 0 do
4          v = izberi vozlisce z vhodno stopnjo 0 v G_t;
5          s = s + 1;
6          Ord(v) = s;
7          G_t = G_t - v;
8  end
9  if G_t = prazna mnozica then
10         return DA in Ord(...);
11 else
12         NE;
13 end
```

17.2.2 Časovna zahtevnost

$\Theta(|V|^2)$:

- ker se telo while izvede $\leq |V|$-krat
- izbiranje točke v zahteva $\leq |V|$ primerjav
- ostale operacije so reda $\Theta(1)$

17.3 Najcenejše poti med začetnim vozliščem in vsemi ostalimi v acikličnih grafih

Razmišljamo

- dan imamo graf G(V,E,c)
- Vzemimo dve poljubni točki i in j. Če i $\rightarrow$ j (tj. iz i ni poti do j), potem najcenejša pot do j ni odvisna od najcenejše poti do i. Zato u_j **ni odvisna** od u_i, zato u_j smemo izračunati pred $u_i (j \prec i)$.

Če je G acikličen, za vsak i,j velja $i \nrightarrow j$ in $j \rightarrow i$, tj. $\forall i, j$ velja $j \prec i \wedge i \prec j$, tj. $\forall i, j$: enega od u_i in u_j smemo izračunati pred drugim.

Katerega?
Tistega, ki ima manjšo oznako v topološki ureditvi.

Razlogi Naj bo G(V,E,c) acikličen. G topološko uredimo. Zaradi enostavnosti vozlišča preimenujemo z njihovimi topološkimi oznakami: $i := Ord(i), \forall i \in V$. Po tem velja: $(i, j) \in E \Rightarrow i < j$. Iz tega sledi, da v vozlišče i lahko pridemo le iz vozlišč k, kjer je $k < i$.
Bellmanove enačbe se zato poenostavijo v:

$$u_i = \begin{cases} 0, i = 1 \text{ (izhodišče)} \\ min_{k<i}\{c_k + c_{ki}\} \end{cases} \tag{14}$$

Računanje rešitve $u_1, u_2, ...u_n$ po naraščajočih i:

$$\begin{aligned} u_1 &= 0 \\ u_2 &= u_1 + c_{12} \\ u_3 &= min\{u_1 + c_{13},\ u_2 + C_{23}\} \\ &\vdots \\ u_i &= min\{u + c_{1i},\ u2 + c_{2i}, ...,\ u_{i-1} + c_{i-1,i}\} \\ &\vdots \\ u_n &= min\{u + c_{1n},\ u2 + c_{2n}, ...,\ u_{n-1} + c_{n-1,n}\} \end{aligned}$$

18 Algoritem po naraščajočih i

Računaj u_i [1]

18.1 Časovna zahtevnost

Pri računanju u_i rabimo:

- i-1 seštevanj
- i-2 primerjav

[1] Tukaj je nekaj nejasnosti glede tega kaj je profesor napisal na predavanjih. Prosim koga, ki ima ta del zapiskov urejen, da pošlje scan ali fotografijo po mailu na alenka [pika] caserman [afna] gmail [pika] com

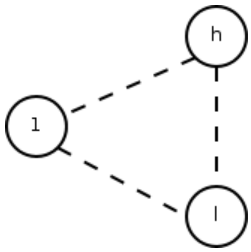
- vseh seštevanj je $\sum_{i=2}^{n}(i-1) = \Theta(n^2)$
- vseh primerjav je $\sum_{i=2}^{n}(i-2) = \Theta(n^2)$

Časovna zahtevnost je torej $\Theta(n^2)$.

19 Najcenejše poti med začetnim in vsemi ostalimi vozlišči v grafih s pozitivnimi cenami (Dijkstra)

Razmišljamo Imamo graf $G(V, E, c)$, kjer $c(e) > 0, e \in E$. Recimo, da velja $u_k < u_l$.

Ali gre najcenejša pot do k lahko preko l?



Če so vse cene pozitivne to ni možno. To lastnost izkorišča Dijkstrov algoritem.

19.1 Dijkstrov algoritem

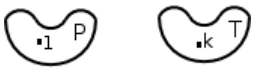
Predpostavimo, da je v nekem trenutku v razbitju P in T (disjunktni mnoici):

$$V = (P \cup T) \wedge (P \cap T = \emptyset)$$

- če $i \in P : u_i \equiv$ cena najcenejše poti iz 1 do i med vsemi potmi v G
- če $i \in T : u_i \equiv$ cena najcenejše poti iz 1 do i med potmi, ki gredo le čez točke iz P (razen zadnje)

Naredimo sledeče:

1. v T poiščemo k z najmanjšim u_k



2. prestavimo k iz T v P

3. ostalim točkam $j \in T$ popravimo u_j takole:

$$u_j = min\{u_j, u_k + c_{kj}\}$$

Trditev Razširjena P in skrčena T sta ohranili lastnosti (zančna invarianta). Dokaz: s protislovjem.

19.1.1 Algoritem

VHOD: $G(V, E, c)$, $c(e) > 0$ za $\forall\, e \in E$
IZHOD: u_i

```
1  begin
2      P = {1};
3      T = {2, 3,..., u};
4      u_1 = 0;
5      za vsak i > 1: u_i = c(1, i), ce je (1,i) v mnozici E
6                                    sicer u_i = neskoncno
7      while T != 0 do
8          poisci k element T, kjer u_k = min{u_i}
9          T = T - {k};
10         P = P unija {k};
11         za v_j v mnozici T: u_j = min{u_j, u_k + c_kj};
12     end
13 end
```

19.2 Časovna zahtevnost

Jedro zanke while se izvede n-krat. Pri i-tem izvajanju zanke vsebuje T n-1 vozlišč. Da najdemo k med njimi in da ostalim popravimo u_j rabimo $\Theta(n-1)$ korakov (primerjav, seštevanj). Časovna zahtevnost je torej $\Theta(n^2)$.

20 Najcenejše poti med začetnim in vsemi ostalimi vozlišči v splošnih grafih (toda brez negativnih ciklov) (Bellman - Fordov algoritem)

Imamo graf $G(V, E, c)$ brez posebnih lastnosti. Veljata splošni Bellmanovi enačbi:

$$u_i = \begin{cases} 0,\ i = 1 \\ min_{k \neq i} \{u_k + c_{ki}\}, \text{ sicer} \end{cases}$$

Razmišljamo Najkrajša pot iz 1 do i gre čez $\leq n-1$ povezav, sicer bi se neko vozlišče ponovilo, dobljeni cikel bi bil pozitiven $\Rightarrow$ pot ne bi bila najkrajša.

Definicija $u_i^{(p)} \equiv$ cena najcenejše poti iz l do i, ki vsebuje kvečjemu p povezav.
Opomba: $... \leq p+1 \Leftrightarrow ... = p+1 \vee ... \leq p$

Torej najcenejša pot iz 1 do i, ki gre čez $\leq p+1$ povezav gre bodisi čez p+1 povezav... kdaj ima ceno $min_{k \neq i} \left\{ u_k^{(p)} + c_{ki} \right\}$ ali pa čez $\leq p$ povezav... kdaj ima ceno $u_i^{(p)}$.

Njena cena bo manjša od obeh cen:

$$u_i^{(p)} = min\left\{u_i^{(p)}, min_{k\neq i}\left\{u_k^{(p)} + c_{ki}\right\}\right\}$$

Nove Bellmanove enačbe so:

$$u_i^{(p)} = \begin{cases} 0, \text{ če je } i = 1 \\ c_{1i}, p = 1 \\ min\left\{u_i^{(p-1)}, min\left\{u_k^{(p-1)} + c_{ki}\right\}\right\}, p = 2, 3, ..., n \end{cases}$$

Kako to izračunati?

p	
1	inicial.
2	
⋮	
m	Izračunati moramo n vrednosti
⋮	$u_1^{(m)}, u_2^{(m)}, ..., u_j^{(m)}, ..., u_n^{(m)}$.
⋮	Pri tem rabimo le prejšnje vrednosti (pri p = m - 1)
⋮	
n-1	

20.1 Algoritem Bellman - Ford

```
1 for p=1 to n-1 do
2     for i=1 to n do
3         u_i^(p) = min{u_i^(p-1), min{u_k^(p-1) + c_ki}}
4     end
5 end
```

Časovna zahtevnost Telo zanke zahteva:

- n-1 seštevanj
- n-1 primerjanj
- $\Rightarrow$ skupaj $\Theta(n)$ operacij

Telo zanke se izvede kvečjemu $n(n-1) = \Theta(n^2)$-krat $\Rightarrow$ časovna zahtevnost je $\Theta(n^3)$.

21 Najcenejše poti med vsemi pari vozlišč

Kako za vsak par $i, j \in V$ izračunati pot?
Pri $\forall i \in V$ sprožimo nek algoritem A za najcenejše poti iz začetnega vozlišča do ostalih vozlišč.

Npr.

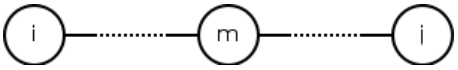
	A	$n \times A$
cikličen graf	$\Theta(n^2)$	$\Theta(n^3)$
pozitivne cene (Dijkstra)	$\Theta(n^2)$	$\Theta(n^3)$
za splošne grafe	$\Theta(n^3)$	$\Theta(n^4)$
Bellman-Fordov algoritem	??	??

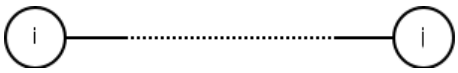
Ali se da $\Theta(n^4)$ izboljšati?
Da!

- posplošeni Bellman-Ford algoritem $\Theta(n^3 logn)$
- Floyd-Warshallov algoritem $\Theta(n^3)$

21.1 Floyd-Warshallov algoritem

Naj bo $G(V, E, c)$ poljuben, brez negativnih ciklov. Spet velja, da je $u_{ij} \equiv$ cena najcenejše poti iz i v j, kjer so vmesna vozlišča kvečjemu m[2].
Opazimo: $u_{ij} = u_{ij}^{(n)}, \forall i, j$.
Velja: najcenejša pot iz i do j, kjer so vmesna vozlišča kvečjemu m:

- bodisi gre čez točko m... 

- ali pa ne gre... 

Zato je njena cena $u_{ij}^{(m)} = min\left\{u_{ij}^{(m-1)}, u_{im}^{(m-1)} + u_{mj}^{(m-1)}\right\}$.
Posebej: $u_{ij}^{(m)} = \begin{cases} c_{ij}, \text{ če } m = 0 \\ min\left\{u_{ij}^{(m-1)}, u_{im}^{(m-1)} + u_{mj}^{(m-1)}\right\}, \text{ če } m = 1, 2, ...n \end{cases}$
Računanje rešitev $u_{ij}^{(m)}$ poteka po naraščajočih m.

m	
0	inicializacija $u_{ij}^{(0)}$
1	
2	
⋮	
k-1	
k	izračunati je treba n^2 vrednosti $u_{ij}^{(k)}, \forall ij = 1, 2, ..., n$. Pri tem rabimo le prejšnje vrednosti $u_{ij}^{(k-1)}$.
⋮	
n	

[2]Ali je to pravilno? Kaj je tu mišljeno kot m?

Algoritem (shema)

```
1  int i, j, m;
2  for(m = 1; m <= n; m++)
3  {
4      for(i = 1; i <= n; i++)
5      {
6          for(j = 1; j <= n; j++)
7          {
8              u[i][j][m] = min(u[i][j][m-1],
9                               u[i][m][m-1] + u[m][j][m-1]);
10         }
11     }
12 }
```

Časovna zahtevnost $\Theta(n^3)$

Prostorska zahtevnost Za izračun k-te vrstice potrebujemo le k-1 vrstico $\Rightarrow$ rabimo $2 \cdot n^2$ prostora (za $2 \cdot n^2$ števil).
Toda, vse lahko izračunamo na prostoru n^2.
Še enkrat tabela:

m	
0	
1	
⋮	
k-1	
k	$\odot$
⋮	
n	

V $k-1$ vrstici je n^2 števil $u_{ij}^{(k-1)}, \forall i,j \in \{1,2,...,n\}$. Predstavljamo si jih v matriki:

$$U = \begin{array}{c} \\ i \end{array} \begin{array}{c} \begin{array}{c} j \end{array} \\ \begin{bmatrix} \vdots \\ \cdots \quad \odot \\ \\ \end{bmatrix} \end{array}$$

Ali za izračun k-te vrstice rabimo prostor za še eno matriko?
Ne. Elemente le te matrike laho izračunamo na mestu (in situ) matrike U:

$$U_{ij} = min\{U_{ij}, U_{ik} + U_{kj}\}$$

Pri izračunu "novega"U_{ij} rabimo 2 stara elementa $\oplus$ in $\otimes$, tako da njuno vsoto primerjamo s starim $\odot$. Za $\forall i, j$ prideta $\oplus$ in $\otimes$ iz k-tega stolpca/vrstice.

$$U = \begin{array}{c} \\ k \\ i \end{array} \begin{array}{c} \begin{array}{cccc} & & k & j \end{array} \\ \left[\begin{array}{c|c|c|c} & & \vdots & \\ \hline & \circ & \otimes & \\ \hline \cdots & \oplus & \odot & \cdots \\ & & \vdots & \end{array}\right] \end{array}$$

Algoritem Floyd-Warshall

VHOD: $C = [c_{ij}]$ matrika cen
IZHOD: $U = [u_{ij}]$ matrika cen najcenejših poti med vsemi pari

Listing 8: Algoritem Floyd-Warshall

```
1  int k, i, j;
2  U = C;
3  for(k = 1; k <= n; k++)
4  {
5      for(i = 1; i <= n; i++)
6      {
7          for(j = 1; j <= n; j++)
8          {
9              U[i][j] = min(U[i][j], U[i][k] + U[k][j]);
10         }
11     }
12 }
```

Časovna zahtevnost n^3 - vse smo opravili na mestu

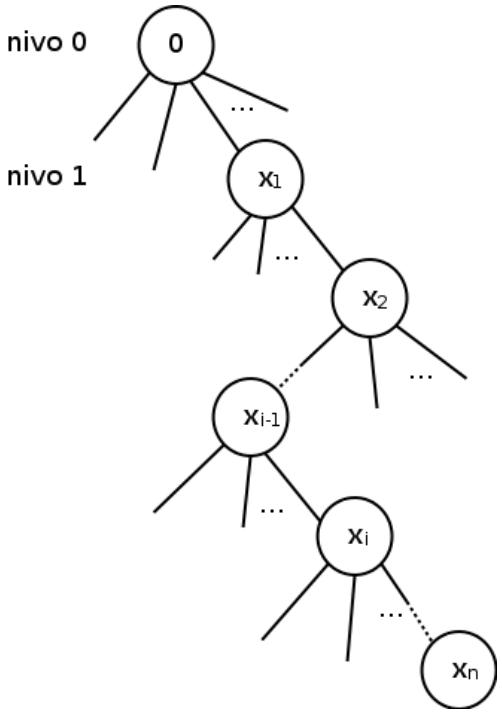
Opomba Graf G vsebuje negativni cikel $\iff u_{ii}^{(m)} < 0$ za nek m in i. $u_{ii}^{(m)}$ je **diagonalna vrednost**.

22 Sestopanje (backtracking)

Metoda, uporabna pri problemih, kjer rešitev sestavljamo postopoma. Rešitev zapišemo kot vektor $x = (x_1, x_2, ..., x_n)$, ki mora zadoščati nekemu pogoju R, pri čemer mora biti $x_i \in K_i$ (K_i je končna množica kandidatov za x_i). Pišemo $m_i = |K_i|$.

Rešitev x sestavljamo korakoma tako, da na i-tem koraku izberemo nek $x_i \in K_i$. Po $i < n$ korakih imamo zaporedje $x_1, x_2, ..., x_i$. Toda to ni nujno začetek rešitve. Da ne bomo zgrešili rešitve (zaradi napačnega izbora x_i) bo treba nekako sistematično pregledati celo množico kandidatov K_i.

Iskalno drevo



Naivni algoritem Z nekim obhodom drevesa sistematično preglej vse poti od korena do listov (≡ vse liste). Toda, drevo ima $\Pi_{i=1}^{n} m_i$ listov (če je izbor x_i neodvisen od prejšnjih izborov), kar pomeni, da je to že v najenostavnejšem primeru ($m_i = 2, \forall i$) 2^n.
Naivni algoritem je treba izpopolniti, če se le da.
Kako?

Zamisel Denimo, da bi imeli nek predikat (pogoj) in funkcijo Obetaven, ki bi na osnovi delne rešitve $x_1, ..., x_{i-1}$, povedala ali nek $x_i \in K_i$ lahko vodi k rešitvi. Tedaj bi pred izborom $x_i \in K_i$ preverili še, če velja $Obetaven(x_i)$. Če bi to veljalo, bi x_i dodali k delni rešitvi, sicer pa bi x_i zavrgli (in s tem celo poddrevo) in namesto njga poskusili z drugim $x_i \in K_i$.

Kaj pa če v K_i ne bi bilo nobenega več?
To bi pomenilo, da tudi x_{i-1} ne vodi k rešitvi - je neobetaven ⇒ torej se moramo vrniti na prejšnji nivo drevesa - **sestopiti**. Rešitev $x_1, x_2, ..., x_{i-2}, x_{i-1}$ "podreti"(zbrisati x_{i-1}) in se spet ukvarjati z izborom x_{i-1}.

Algoritem

VHOD: specifično za problem, ki ga rešujemo
IZHOD: nasli_resitev=TRUE in rešitev $x = (x_1, x_2, ..., x_n)$, za katero velja $R(x)$ ali pa nasli_resitev=FALSE
OPOMBE: x je globalna, tudi nasli_resitev je globalna

```
priprava specificna za resevalni problem
x := ();..... resitev je na zacetku prazna
nasli_resitev := FALSE;
poskusi_korak(1);.... prehod z 0 v x1
```

Listing 9: Primer izboljšane psevdokode za sestopanje

```
PROCEDURE poskusi_korak(i : INTEGER)
    BEGIN
        pripravi Ki;
        REPEAT
            ki = izberi nepregledanega kandidata v Ki;
            IF Obetaven(ki) THEN
                xi := ki;..... dodaj ga v delno resitev
                IF i < n THEN..... x nepopoln
                    poskusi_korak(i + 1);..... rekurzija
                    IF NOT nasli_resitev THEN zbrisi x;
                ELSE
                    nasli_resitev = R(x);
                END
            END
        UNTIL nasli_resitev OR v Ki so vsi pregledani
    END
END
```

Del VIII
Zaključek

23 Kako so zapiski nastali?

Zapiski so nastali ročno pisani obliki med predavanji za predmet Algoritmi in podatkovne strukture 2, ki ga je leta 2006 predaval prof. dr. Borut Robič. Zadnji mesec predavanj sem ugotovila, da so zapiski ponekod nenatančni in slabo urejeni in sem se zato odločila, da jih v celoti pretipkam. Po nekaj dneh vztrajnega tipkanja sem ugotovila, da je zapiskov enostavno preveč, da bi jih v doglednem času zmogla sama pretipkati in urediti, zato sem za pomoč poprosila sošolce. Vsak je pretipkal in uredil vsaj 5 poskeniranih strani zapiskov. Nekateri študenti so sodelovali tudi pri urejanju slik in diagramov, ki se pojavljajo v zapiskih. Kjer so bile v zapiskih kakšne nedoslednosti in napake, so sošolci prispevali njihove verzije zapiskov za uskladitev. Tako smo upam da, odpravili večino napak.

24 Namen zapiskov

Zapiski so kot se za zapiske spodobi namenjeni učenju tako za pisni kot za ustni izpit, lahko pa so tudi odlična referenca za kasneje, saj so opremljeni z

kazalom. Zapiski bodo po koncu tega študijskega leta javno objavljeni in upam da bodo koristili tudi prihodnjim generacijam. Namen teh zapiskov vsekakor ni to, da se študente oskrbi s kompletnim materialom za samostojno študiranje, da jim potem ni potrebno hoditi na predavanja, pač pa naj bodo koristen učni pripomoček, ki omogoča to, da se študent med predavanji bolj posveti temu, kaj profesor razlaga kot pa temu, da čimhitreje in brez napak prepiše snov s table.

Čimveč algoritmov zapisanih v tej skripti, sem zaradi lažje razumljivosti skušala zapisati v Javi. Trenutno se na predavanjih uporablja učbenik prof. Vilfana (Osnovni algoritmi), v katerem pa so primeri algoritmov v jeziku Oberon, ki se ga študenti Fakultete za računalništvo in informatiko vsaj do 2. letnika ne učimo. Ker smo se v 1. in 2. letniku študija spoznavali z jezikoma Java in C, se mi je zdelo bolj logično, da se primeri algoritmov zapišejo v katerem od teh dveh jezikov.

25 Zahvala

Za pomoč pri urejanju zapiskov se iskreno zahvaljujem sošolcem:

- Dolinar Leon
- Urša Levičnik
- Jančič Andrej
- Simčič Matej
- Leposava Knez
- Kovač Marko
- Aleksej Alek
- Aleš Šavli
- Filej Miha
- Pufič Mitja

Literatura

[1] Vilfan Boštjan, *Osnovni algoritmi*, Fakulteta za računalništvo in informatiko, Ljubljana, 2. izdaja, 2002

[2] Knuth Donald, *The Art of Computer Programming, Volume 3: Sorting and Searching*, Addison-Wesley, Massachusetts, Second Edition, 1998