Algoritmi in podatkovne strukture 2: Zapiski predavanj do izrednih ukrepov naše preljube vlade

Pradavatelj: Borut Robič

Napisal: Janez Justin

2. semester 2019/2020

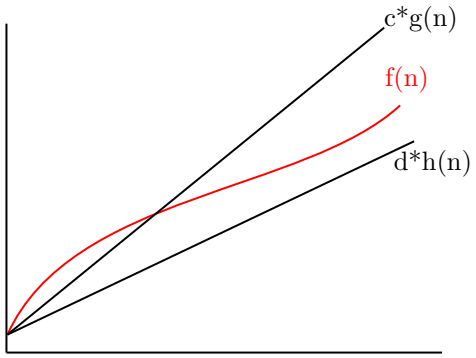
Literatura

- Cormen,...: Introduction to Algorithms
- Segewick,...: Algorithms
- Sahni,...: Data structures, Algorithms and Applications
- Skiena,...: The Algorithms Design Manual
- Knuth, Patashnik, Greene: Concrete Mathematics

Chapter 1

Ocenjevanje porabe računskega virov

Poraba računskega vira narašča, ko narašča velikost primerkov računskega problema.



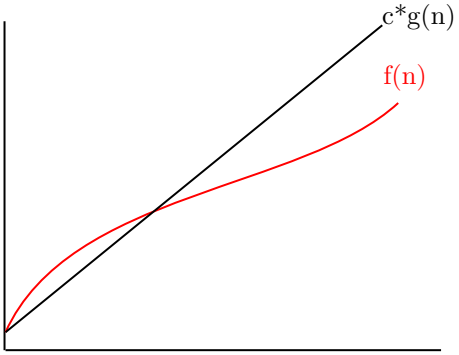
Zadovoljimo se, če uspemo f(n) čimbolje omejiti navzgor in navzdol z dvema funkcijama, g(n) in h(n).

1.1 Asimptotična rast funkcij

Funckija f(n) ja **asimptotčno**:

- **Omejena navzgor** s funkcijo g(n), kadar:

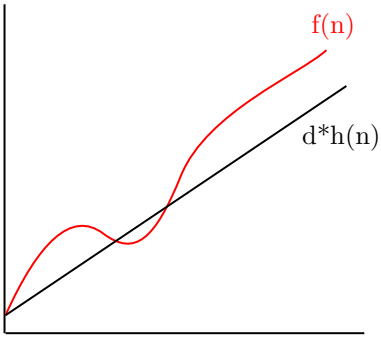
$$\exists c > 0, n_0 > 0 : (\forall n \geq n_0 : f(n) \leq c \cdot g(n))$$



To zapišemo kot: $f(n) = O(g(n))$

- **Omejena navzdol** s funkcijo h(n), kadar:

$$\exists d > 0, n_1 > 0 : (\forall n \geq n_1 : d \cdot h(n) \leq f(n))$$



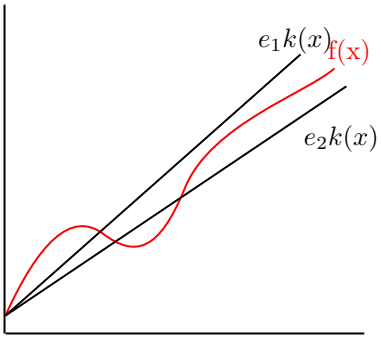
To zapišemo kot: $f(n) = \Omega(h(n))$

- **Enaka** funkciji k(n), kadar:

$$f(n) = O(k(n)) \wedge f(n) = \Omega(k(n))$$

$$oz.$$

$$\exists e_1, e_2 > 0, n_2 > 0 : (\forall n \geq n_2 : e_1 \cdot k(n) \leq f(n) \leq e_2 \cdot k(n))$$



To zapišemo kot: $f(n) = \Theta(h(n))$

Primeri:

- $f(n) = \sum_{k=1}^{n} k$
  $f(n) = \frac{1}{2}n(n+1) = \frac{1}{2}n^2 + \frac{1}{2}n = \frac{1}{2}n^2(1+\frac{1}{n}) \to \frac{1}{2}n^2$
  $f(n) = \Theta(n^2)$

- $f(n) = a_n x^n + ... + a_1 x + a_0$
  Prepuščeno bralcu (D.N.)

- $f(n) = n!$
  $n! \doteq \sqrt{2\pi n}(\frac{n}{e})^n = \begin{cases} O(n^{n+\frac{1}{2}}) \\ \Theta(\frac{n^{n+\frac{1}{2}}}{3^n}) \end{cases}$

- f(n) $= \sum_{i=1}^{n} log(i) =$
  $log1 + log2 + ... + logn = log(1 \cdot 2 \cdot 3 \cdot ... \cdot n) = /stirling/ = log(\sqrt{2\pi n}(\frac{n}{e})^n) =$

  $log(\sqrt{2\pi}) + \frac{1}{2}logn + n \cdot logn - n \cdot loge = nlogn[\frac{log\sqrt{2\pi}}{nlogn} + \frac{1}{2n} + 1 - \frac{loge}{logn}] \doteq$
  $k \cdot nlogn = \Theta(nlogn)$

- f(n) $= c \cdot a^{log_b n} = c \cdot (b^{log_b a})^{log_b n} = c \cdot (b^{log_b n})^{log_b a} = c \cdot b^{log_b a} = \Theta(n^k)$

- $f(n) = \sum_{k=1}^{\infty} \frac{k}{2^k} = 2$

Chapter 2

Urejanje

Dani so:

- $a_1, a_2, ..., a_n$
- Relacija $\preceq$ popolne urejenost

Cilj: Poiskati razporeditev $a_{i_1}, a_{i_2}, ..., a_{i_n}$, da bo $a_{i_1} \preceq a_{i_2} \preceq ... \preceq a_{i_n}$

Če je n takšno, da so vsi a_i v glavnem pomnilniku v neki tabeli, bomo imeni **notranje urejanje**.
Če pa je n prevelik, so na zunanjem pomnilniku v neki datoteki. Nekateri bodo dosegljivi preko datotečnega okna. Urejanje bo **zunanje**.

Algoritmi za notranje urejanje:

- navadni ... preprosti in počasnejši
- izboljšani ... zapleteni in hitrejši

2.1 Notranje urejanje

Urejanje tabel velikosti n.

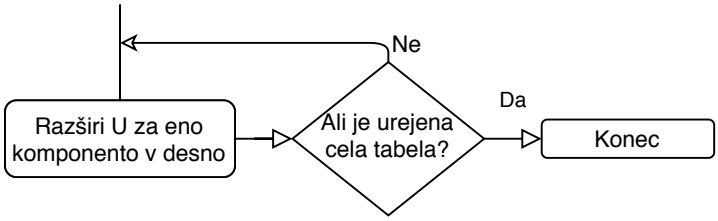
2.1.1 Navadni algoritmi

Imajo podobno zgradbo:

0	1	...	$i-1$	i	...	$n-1$

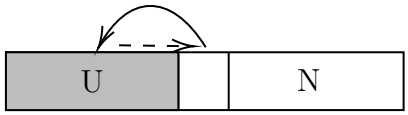
U — Urejeni del; N — Neurejeni del

Shema algoritma:

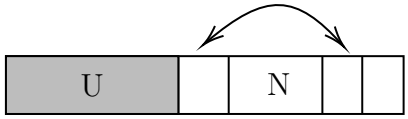


Navadni algoritmi se razlikujejo, kako razširijo U:

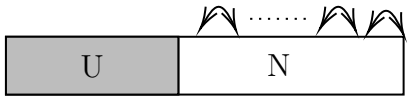
- **Insertion sort**:
  Prvi elementv N pravi(vrini) na ustrezno mesto v U



- **Selection sort**:
  prvi element v N zamenjaj z najmanšim elementom v N



- **BubbleSort**:
  Sprehodi se po N od leve proti desni. Vsakokrat primerjaj soseda in ju po potrebi zamenjaj



R(i) - časovna zahtevnost razširjanja U v i-ti ponovitvi
Časovna zahtevnost enostavnih algoritmov $\mathbf{T(n)} = \sum_{i=1}^{n} R(i) =$ /se izkaže (D.N.) / $= \Theta(n^2)$

Pojavi se vprašanje: Ali se da hitreje?
Odgovor: DA.

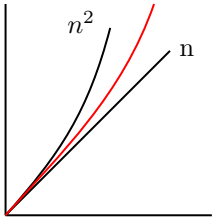
Izboljšani alogritmi

- Shellovo Urejanje ... $O(n^\alpha)$, $\alpha > 2$
- Heapsort (urejanje s kopico) ... $\Theta(nlogn)$
- QuickSort (hitro urejanje) ... $\Theta(nlogn)$ v povprečju

2.1.2 Spodnja meja za časovno zahtevnost notranjega urejanja

Navadni algoritmi imajo $\Theta(n^2)$ operacij primerjanja in zamenjav.

Vprašamo se: Ali se da (asimptotično) hitreje?
Npr. V času $\Theta(n)$? Ali celo v $\Theta(\sqrt{n})$? $\Theta(logn)$?



Za branje vseh podatkov porabimo $O(n)$ časa, zato je spodnja meja funkcija n. Zanima nas, če obstaja funkcija $f(n)$, da bo $O(n) < O(f(n)) < O(n^2)$.

Pri urejanju uporabljamo dve operaciji:

- Primerjanje
- Zamenjava

Mi ocenjujemo le najmanj koliko primerjanj je potrebnih.

Vprašanje: Kako bi uredili n števil, če bi imeli le operacijo primerjavo?

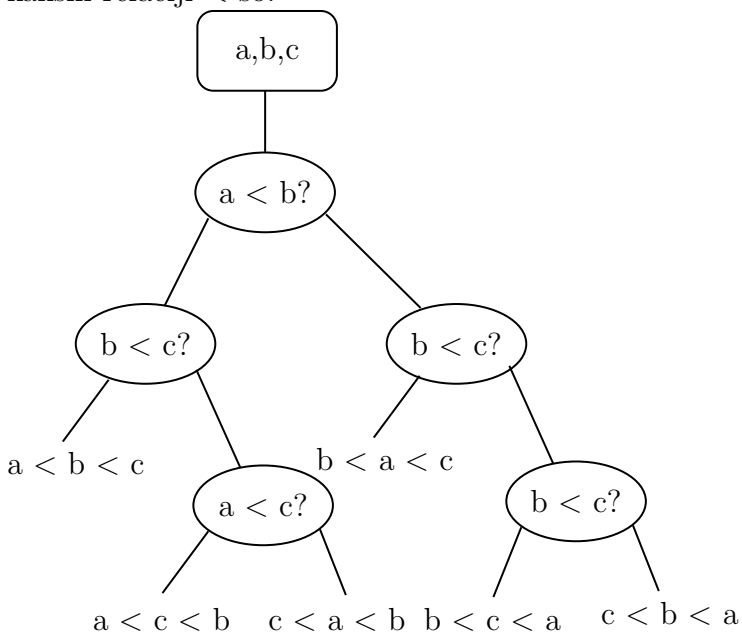
a	b	c
3	-2	16

$\Longrightarrow b < a < c \Longrightarrow$

b	c	c
-2	3	16

Primer:

- Dana so 3 paroma različna števila a, b, c.
  Ugotovi, v kakšni relaciji $<$ so.



Ugotovimo: Ko pridemo iz korena do lista, izvemo v kakšni relaciji $<$ so a, b, c.
Algoritem je dvojiško drevo, ki ima v notrajih vozliščih **operacije primerjave**, v listih pa vse relaciji med danimi a, b, c, tj. **vse permutacije treh elementov**.

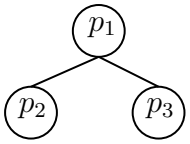
Posplošimo ta primer na n števil a_1,...,a_n.
Zgradimo podobno odločitveno drevo(kot za primer a,b,c), ki bo imelo v notranjosti vozlišč primerjave, v listih pa vse permutacije.
Analiza: Opisano drevo je **dvojiško** in ima **N = n! listov**
DEF: Višina v drevesu = število notranjih vozlišč na najdaljši poti iz korena do listov.
VEMO: Vsako dvojiško drevo z N listi ima višino h, kjer je $h \geq \lceil log_2 N \rceil$
Torej: pri ugotavljanju, katero od teh relacij velja, bo potrebnih
$h(n) \geq \lceil log_2 n! \rceil$
Računamo: $h(n) \geq =$ /Stirling/ $\geq k \cdot logn$.
$\Longrightarrow$ Vsako urejanje n števil, ki uporabljajo operacijo primerjanja zahteva $k \cdot n \cdot logn = \Omega(n \cdot logn)$ operacij primerjanja(za ureditev vsaj enega primerka a_1,...,a_n).

Za vsak algoritem notranjega urejanja obstaja vhod a_1,...,a_n za katerega ureditev je potrebnih $\geq k \cdot n \cdot logn$ primerjav.
Zato je časovna zahtevnost notranjega urejanja $\underline{\Omega(n \cdot logn)}$.

2.1.3 Heapsort(Urejanje s kopico)

Def: Kopica(heap) je dvojiško drevo(vozlišča imajo podatke), ki je:

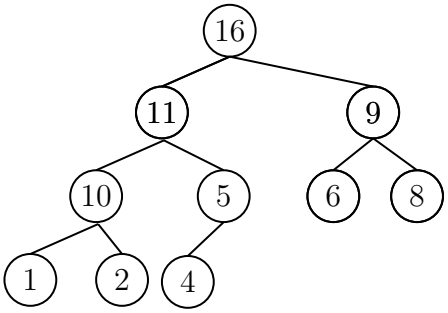
- urejeno:
  ∀ vozlišče ima podatek, ki je večji od podatkov v sinovih tega vozlišča.



$$\mathrm{p}_1 \geq p_2 \wedge p_1 \geq p_3$$

- levo poravnano:
  - Vse veje(pot iz korena do lista) so dolge d ali d-1(d je $\mathbb{N}$)
  - Vse daljše veje so levo od krajših vej

Primer:



Opazimo:

- Podatki vzdolž veje padajo
- Koren ima največji podatek
- Levo in desno poddrevo kopice sta tudi kopici.

Zamisel algoritma:

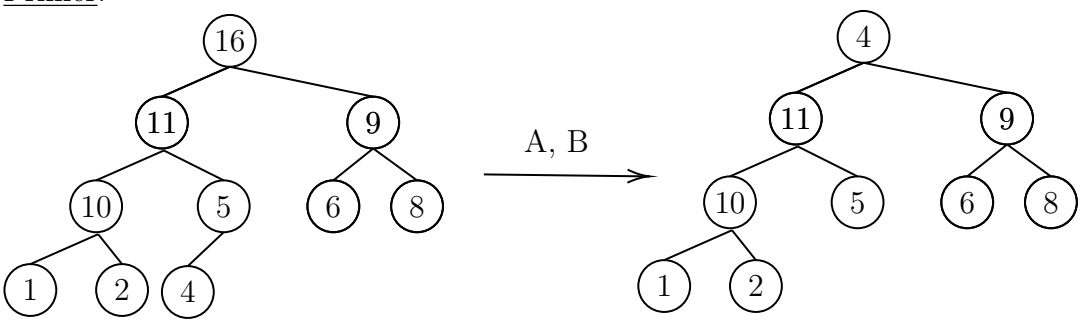
A. Izloči koren iz drevesa

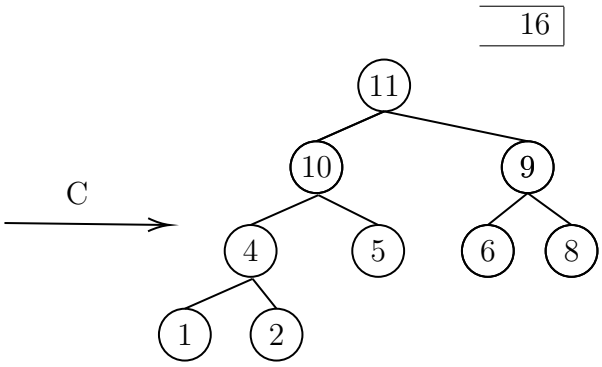
B. Na mesto izločenega korena prestavi zadnji list

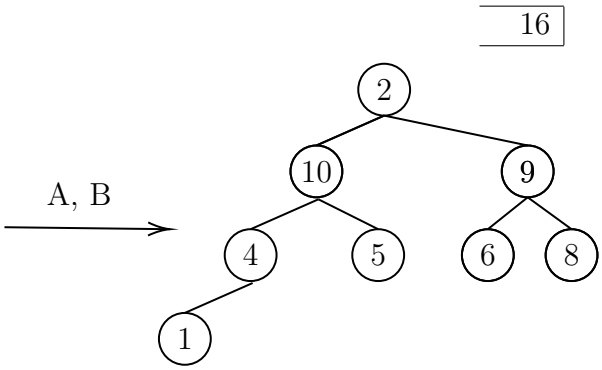
C. Popravi dobljeno drevo v kopico

D. Ali smo konec? Če ne, pojdi na A

Primer:







11 16

In tako dalje ponavaljamo, dokler heap ni prazen.

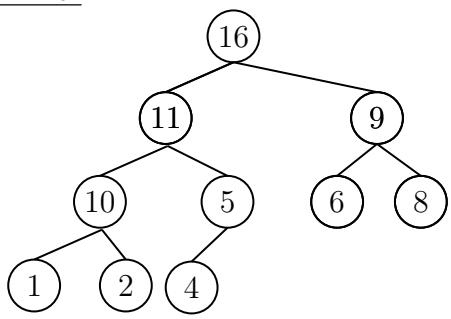
Implementacija

Podatkovna struktura:

$a:$	1	2	...	i	...	$2i$	$2i+1$	...	n

- Koren kopice v a[1]
- Če je vozliše v a[i], sta sinova v a[2i] (levi sin) in a[2i+1] (desni sin).

Primer:



$\Longrightarrow$

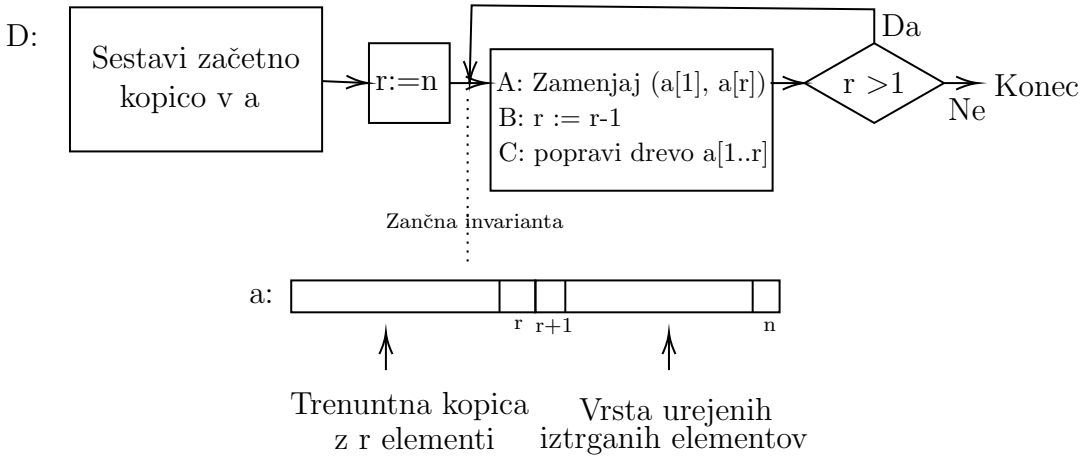
16	11	9	10	5	6	8	1	2	4

Torej:

- Če je $s > 1$, potem je v a[s] sin od nekoga. Čigav? Njegov oče je v a[$\lfloor s/2 \rfloor$]. Če je s sodo, potem je s levi sin, sicer desni.
- Zadnji oče je v komponenti a[$\lfloor \frac{n}{2} \rfloor$]

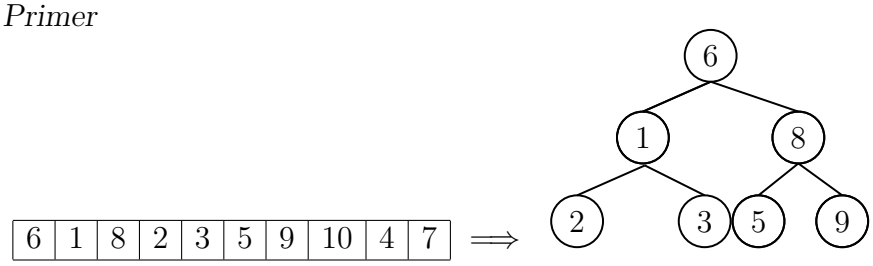
Algoritem(Podrobneje)

Naloga D bo vzpostavila začetno kopico.



Predpostavimo, da že imamo razvito proceduro *Popravi_v_kopico(i,j)*, ki zna popraviti drevo v kopico, če sta poddrevesi vozlišča i že kopici.
Potem bi bila naloga C le klic *Popravi_v_kopico(1,j)*. Toda to proceduro lahko uporabimo tudi za razvoj naloge D.

Vprašanje: Kako izvesti D?
Recimo, da so števila v drevesu.
Primer



Od zadnjega očeta po nivojih od spodaj navzgor in od desne proti levi na vsakem nivoju pri vsakem obiskanem vozlišču pokličemo *Popravi_v_kopico*.
D je zato lahko naslednja:

```
procedure sestavi_zacetno_kopico:
    var i:Integer
    begin for i:=n div2 downto 1 do
        popravi_v_kopico(i, n)
    end

procedure  popravi_v_kopico(var i, j:Integer):
    var s:Integer
    begin
    if i <= (j div2) then // Ali je nek oče v a[i]
        begin
        s := 2*i // v a[s] je levi sin
        if s+1 <= j then // če obstaja tudi desni sin
            if a[s] < a[s+1] then s:=s+1
            if a[i] < a[s] then  // če je ta sin večji od očeta
                begin zamenjaj(a[i], a[s])
                // drevo kamor je prišel oče popravi (če je treba)
                popravi v kopico(s, j)
            end
        end
    end
```

Z definiranima *popravi_v_kopico* in *sestavi_zacetno_kopico* lahko implementiramo heapsort(a[])

```
heapsort(var a[1..n])
    var r:Integer
    begin
        sestavi_zacetno_kopico // D
        r := n
        while r > 1
        begin
            zamenjaj(a[1], a[r]) // A, B
            r := r - 1
            popravi_v_kopico(1,r) // C
        end
    end
```

Časovna zahtevnost Heapsorta

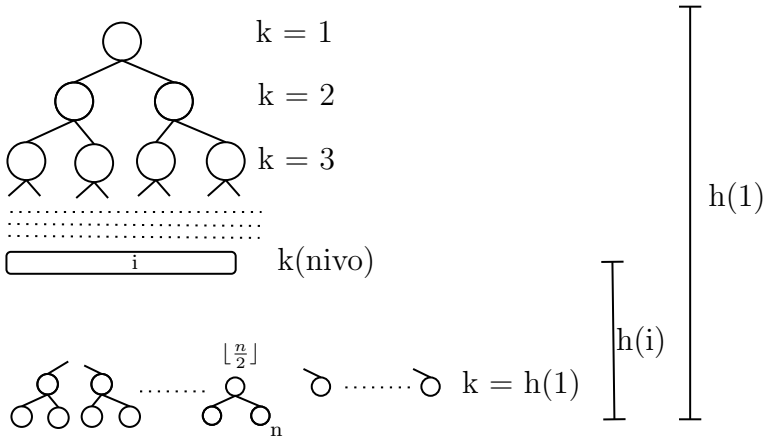
V drevesu najvec n elementov in drevo je dvojiško. Torej je višina največ $logn$.
Torej *popravi_v_kopico(i, j)* opravi kvečjemu $log_2 n$ primerjav in zamenjav. Naj bo to število P(i).
Torej: $P(i) \leq log_2 n$

sestavi_zacetno_kopico kliče *popravi_v_kopico* v zanki po $i = \lfloor \frac{n}{2} \rfloor, \lfloor \frac{n}{2} \rfloor - 1, ..., 1$. Zato sledi:

$$Z(n) = \sum_{i=\lfloor \frac{n}{2} \rfloor}^{1} P(i) \leq \sum_{i=\lfloor \frac{n}{2} \rfloor}^{1} log_2 n \leq \lfloor \frac{n}{2} \rfloor log_2 n = O(n \cdot logn)$$

Vprašanje: Kako hitro narašča Z(n)?
Trditev: $Z(n) = \Theta(n)$

Dokaz:



Dejstva:

a. $\forall$ nivo k=1,...,h(1) je pol in ima 2^{k-1} vozlišč

b. Predzadnji nivo ($k = h(1)$) vsebuje med 1 in $2^{h(1)-1}$ očetov (ostali so sinovi)

c. Zadnji nivo (k=h(1)+1) vsebuje med 1 in $2^{h(1)} - 1$ listov (in nič ocetov)

d. Opazimo: $\sum_{k=1}^{h(1)} 2^{k-1} = 1+2+...2^{h(1)-1} = 2^{h(1)} - 1 < n$, zato $2^h(1) \leq n$.

e. Če je vozlišče i oče in je na nivoju k, je na višini $h(i)h(1) + 1 - k$.

f. Spomnimo se:
Vrsta $\sum_{j=1}^{\infty} \alpha_j$ ($\alpha_1 > 0$) konvergira, če je $\lim_{j\to\infty} \frac{\alpha_{j+1}}{\alpha_j} < q < 1$.
Naj bo $\alpha_j = \frac{j}{2^j}$. Kdaj vrsta $\sum_{j=1}^{\infty}$ konvergira?
Kadar: $\frac{(j+1)\cdot 2^j}{2^{j+1}\cdot j} = ... = 1/2 < 1$

Računamo:

$$Z(n) = \sum_{i=\lfloor \frac{n}{2} \rfloor}^{1} \leq^{\text{preko b}} \sum_{k=h(1)}^{1} 2^{k-1}(h(1) + 1 - k)$$

2^{k-1} dobimo iz tocke a. $(h(1) + 1 - k)$ iz točke e).
Uvedemo $j := h(1) + 1 - k$

$$= \sum_{j=1}^{h(1)} 2^{h(1)-j} \cdot j = 2^{h(1)} \cdot \sum_{j=1}^{h(1)} \frac{j}{2^j} \leq^{\text{preko d, f}} n \cdot konst = \Theta(n) \quad \square$$

Celotni heapsort:
D ... $\Theta(n)$
Glavna zanka:

$$\sum_{r=n}^{2}(1+P(r)) \leq n-1+\sum_{r=n}^{2} log_2 n \leq n-1+(n-1)\cdot log_2 n = \underline{\Theta(n \cdot logn)}$$

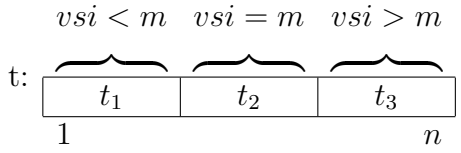
2.1.4 Quicksort

Dana tabela z n števili:

a : | 1 | 2 | ... | n |

Zamisel:

1. Izberi neko število m(pivot)
   Tabelo t prevedi tako, da dobimo



2. na enak način(kot 1.) prevedimo t_1 in t_3

Algoritem:

```
procedure QuickSort(var t)
begin
    if t "ima <= 1 element" then return(t)
    else begin
        m := določi_pivot(t)
        prerazporedi(t, 1, m, m) // definiramo kasneje
        return [QuickSort(t1); t2; QuickSort(t3)]
        // Ali pa: return QuickSort(t1);QuickSort(t2 unija t3)
    end
end
```

Opombe:

- Včasih je t_2 prazna (če je m izračunan in $\neq$ od $\forall$ elementa tabele)
- Včasih je t_1 prazna (če je m $\leq$ najmanjši a[i])

- Včasih je t_3 prazna (če je m $\geq$ največji a[i])

Kako m določiti hitro?

- m := a[1] (lahko slabo, če je t že delno urejena)
- m := a[n] (lahko slabo, če je t že delno urejena)
- m := a[n div2]
- m := naključno izbran
- m := $\lfloor \frac{1}{3}(\text{prvi} + \text{srednji} + \text{zadnji}) \rfloor$
- m := mediana elementov v t

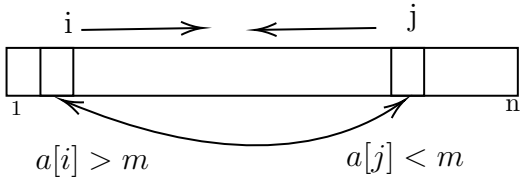
Kako razporediti t?
Zamisel: Najprej iz t sestavimo

t_1	$t_2 \cup t_3$

in nato

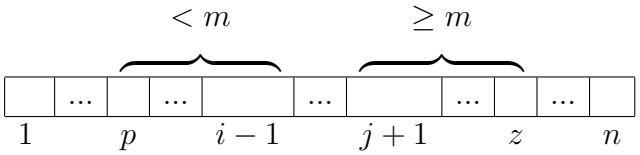
t_1	t_2	t_3

Izberemo le 1. korak:
Kako to izvedemo?
Zamisel:



Algoritem:

Imamo sledeče stanje:




```
procedure prerazporedi(t,p,z,m)
    begin
    i := p; j := z;
    while i < j do
        begin
        while t[i] < m && i < z do i++;
        while t[j] >= m && j >= p do j--;
        if i < j then
            begin
            zamenjaj(t[i], t[j])
                i++;j--;
            end
        end
    end
```

Ko se konča je $t_1 = t[p..i]$ in $t_2 \cup t_3 = t[j+1..z]$

Časovna zahtevnost

- Najslabša časovna zahtevnost (analiza najslabšga primera)
- Najboljša časovna zahtevnost (analiza najboljšga primera)
- Povprečna časovna zahtevnost

APS2

(povzetki predavanj od 15.3. do konca ukrepov 2019/20)

Borut Robič

4. junij 2020

Gradivo nadomešča predavanja iz predmeta APS2 v šolskem letu 2019/20 v času koronavirusa, ko so bila predavanja nadomeščena z delom na daljavo. Snov v gradivu (in snov z običajnih predavanj) bo zadoščala za „teoretični" del izpita.

Kazalo

1 Quicksort

Algoritem Quicksort smo razvili na predavanjih. Nadaljujemo z analizo algoritma.

Analiza časovne zahtevnosti algoritma Quicksort

Zapišimo še enkrat algoritem Quicksort v psevdokodi:

```
procedure Quicksort(t);
begin
  if t ima kvecjemu en element then return(t) else
    begin
      m := izberi med elementi tabele t ali izra"cunaj;
      razdeli t v t_1 in t_2 U t_3;
      return(Quicksort(t_1);Quicksort(t_2 U t_3))
    end
end
```

Naj bo $T(|t|)$ časovna zahtevnost urejanja tabele t (velikosti $|t|$) s proceduro Quicksort. Tedaj iz psevdokode vidimo, da velja:

$$T(|t|) = T(|t_1|) + T(|t_2 \cup t_3|) + \Theta(|t|) + \Theta(1). \qquad (*)$$

Na desni strani izraza je prvi člen časovna zahtevnost urejanja tabele t_1, drugi časovna zahtevnost urejanja tablele $t_2 \cup t_3$, tretji časovna zahtevnost prerazporejanja elementov tabele t in četrti časovna zahtevnost določanja pivota m.

Rešitev T enačbe $(*)$ je odvisna od tega, kako podobni sta dolžini tabel t_1 in $t_2 \cup t_3$. To bo seveda odvisno od vsakokratne tabele t. Zato časovno analizo izvedemo za *najslabši* (najbolj neugodni), *najboljši* (najbolj ugodni) in *povprečni* primer tabele t.

Analiza za najslabši primer

Tabela t bo *najbolj neugodna* tedaj, ko bo vsaka prerazporeditev med izvajanjem Quicksorta vrnila karseda neuravnoteženi podtabeli, tj., ko bo t_1 vsakokrat vseboval en element, $t_2 \cup t_3$ pa vse ostale elemente tabele. Takrat bo enačba $(*)$ prešla v

$$T(|t|) = T(1) + T(|t| - 1) + \Theta(|t|) + \Theta(1)$$

Če pišemo $|t| = n$ in upoštevamo, da je $T(1) + \Theta(|t|) + \Theta(1)$ reda $\Theta(|t|)$, dobimo

$$T(n) = T(n-1) + \Theta(n). \tag{1}$$

Kakšna funkcija $T(n)$ je rešitev te rekurzivne enačbe?

Trditev. Rešitev enačbe (1) je funkcija $T(n)$ z lastnostjo $T(n) = \Theta(n^2)$.

Dokaz. Domača naloga. Namig: razvijte desni del (1) v vsoto členov $c_k k$. □

Sklep: če imamo smolo, bo urejanje tabele t s Quicksortom trajalo $\Theta(|t|^2)$ časa.

Analiza za najboljši primer

Tabela t bo *najbolj ugodna*, če bo vsaka prerazporeditev med izvajanjem Quicksorta vrnila karseda uravnoteženi podtabeli, tj., ko bosta t_1 in $t_2 \cup t_3$ vsakokrat enako dolgi. To je možno le, če je število elementov v t potenca števila 2. Pa predpostavimo, da je $|t| = n = 2^m$ za nek $m > 0$. Tedaj se bo enačba $(*)$ glasila takole:

$$T(n) = T(n/2) + T(n/2) + \Theta(n) + \Theta(1)$$

oziroma

$$T(n) = 2T(n/2) + \Theta(n). \tag{2}$$

Kakšna funkcija $T(n)$ je rešitev te rekurzivne enačbe?

Trditev. Rešitev enačbe (2) je funkcija $T(n)$ z lastnostjo $T(n) = \Theta(n \log n)$.

Dokaz. Domača naloga. □

Sklep: če imamo srečo, bo urejanje tabele t s Quicksortom trajalo $\Theta(|t| \log |t|)$ časa.

Analiza za povprečni primer

V praksi vhodna tabela t ne bo niti najbolj neugodna niti najbolj ugodna, pač pa „povprečna." Analiza za povprečni primer je običajno bolj zapletena, ker moramo upoštevati verjetnostno porazdelitev vhodnih podatkov. Ta ni vedno znana; zato takrat predpostavimo, da so vsi vhodni podatki enako verjetni. To bomo predpostavili tudi v našem primeru.

Predpostavka 1. Naj bodo v t paroma različni elementi. (To poenostavi analizo.)

Naj bo $T(n)$ *povprečni čas urejanja tabele* t z n elementi z algoritmom Quicksort. Če je t prazna ali ima en element, bo čas urejanja konstanten: $T(0) = T(1) = b \in \mathbb{R}^+$.

Naj bo m delilni element. Če bi vedeli, da je m i-ti najmanjši element v t, kjer je i znano število, bi enačbo $(*)$ lahko preoblikovali v

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

Toda m je v resnici lahko katerikoli po velikosti, tj. i je lahko katerikoli od $1,\ldots,n$. Kako to upoštevati? Naj bo $p(i)$ verjetnost, da je m i-ti najmanjši element v t. Tedaj je

$$T(n) = \sum_{i=1}^{n} p(i)\big[T(i-1) + T(n-i) + \Theta(n)\big]. \tag{3}$$

Če želimo nadaljevati, moramo poznati verjetnost $p(i)$.

Predpostavka 2. Naj za vsak $i = 1,\ldots,n$ velja $p(i) = 1/n$.

Zdaj enačbo (3) lahko preoblikujemo v (domača naloga)

$$T(n) = cn + \frac{2}{n}\sum_{i=0}^{n-1} T(i). \tag{4}$$

To je torej povprečni čas urejanja tabele t z n elementi s algoritmom Quicksort. Toda nas zanima asimptotično vedenje funkcije $T(n)$, izraz (4) pa o tem ne govori. Odgovor nam prinese naslednja trditev:

Trditev. Rešitev enačbe (4) je funkcija $T(n)$ z lastnostjo $T(n) \leqslant 2(b+c)\, n \ln n$.

Opomba. Torej za $T(n)$ velja $T(n) = \Theta(n \log n)$.

Dokaz. V teh zapiskih dokaz izpustim. Omenim pa, da trditev dokažemo z indukcijo po $n \geqslant 2$. Pri indukcijskem koraku pa je treba uporabiti neenakost $\sum_{i=2}^{n-1} i \ln i \leqslant \int_2^n x \ln x\, dx$ in izračunati vrednost integrala. □

Sklep: V povprečju bo urejanje tabele t s Quicksortom trajalo $\Theta(|t| \log |t|)$ časa.

2 Metoda deli in vladaj

Pri razvoju algoritmov pogosto uporabljamo metodo, imenovano *Deli in vladaj*. Zamisel metode je tale: Nalogo N dane velikosti razdeli na manjše podnaloge, te reši in iz njihovih rešitev sestavi rešitev naloge N.

Metodo smo že srečali pri razvoju algoritma Quicksort. Tam smo osnovno nalogo, tj. urejanje tabele t razdelili na dve nalogi enake vrste, tj. urejanje podtabele t_1 in urejanje podtabele $t_2 \cup t_3$. Algoritem Quicksort za urejanje tabele t smo nato rekurzivno izrazili z dvema klicema istega algoritma Quicksort, tokrat za urejanje vsake od podtabel. Za *pripravo* podtabel t_1 in $t_2 \cup t_3$ smo uporabili proceduro Razdeli. Iz urejenih t_1 in $t_2 \cup t_3$ smo nato *sestavili* urejeno t (kar je bilo trivialno).

Zdaj pa to posplošimo. Naj bo dan problem P in njegov primerek, tj. naloga $N \in P$ velikosti n. Denimo, da nalogo N lahko razdelimo na $p \geqslant 2$ podnalog $N_1, \ldots, N_p$ velikosti $n_1, \ldots, n_p$ $(0 \leqslant n_i < n)$, ki so vse primerki problema P. Nekateri problemi so take narave, da rešitev naloge lahko sestavimo iz rešitev nekaterih njenih podnalog. Pa denimo, da je tudi problem P tak in da je treba rešiti a $(1 \leqslant a \leqslant p)$ podnalog N_i. Ker so te podnaloge primerki istega problema P kot naloga N, lahko za njihovo reševanje uporabimo kar algoritem A za reševanje naloge N. Ogrodje algoritma A je torej:

```
procedure A(N);
begin
  if n<2 then resi neposredno else
   begin
     R(N);    // delitev N na podnaloge N_1,..,N_p
     A(N_1);  // resevanje a podnalog
     ...                 ...
     A(N_a);             ...
     S(N)     // sestavljanje delnih resitev v koncno
   end
end
```

Tu je $R(N)$ procedura, ki pripravi podnaloge $N_1, \ldots, N_p$, procedura $S(N)$ pa iz rešitev a podnalog sestavi rešitev naloge N. Vidimo, da je algoritem A rekurziven.

Primer. Pri Quicksortu smo imeli P = urejanje tabele, $N = t$, $n = |t|$, $p = 2$, $N_1 = t_1$, $N_2 = t_2 \cup t_3$ in $a = 2$, R = prerazporedi, S trivialno.

Časovna zahtevnost algoritma A

Kaj lahko povemo o časovni zahtevnosti tako splošnega algoritma A? Označimo s $T(n)$, $r(n)$ in $s(n)$ časovne zahtevnosti postopkov $A(N)$, $R(N)$ in $S(N)$. Iz zapisa algoritma A sledi

$$T(n) = T(n_1) + \ldots + T(n_a) + r(n) + s(n). \tag{1}$$

Če odnosi med velikostmi n_i niso znani, je enačbo (1) težko izboljšati. Včasih pa nam uspe pripraviti podnaloge N_i, ki so (približno) enakih velikosti; takrat velja $n_i \approx \frac{n}{c}$, za neki $c \geqslant 2$. Enačba (1) tedaj preide v

$$T(n) = aT(\frac{n}{c}) + r(n) + s(n). \tag{2}$$

Če je $n = 1$, je naloga N trivialna in zato rešljiva v konstantnem času $b = O(1)$. Zdaj lahko zapišemo rekurzivno enačbo za časovno zahtevnost $T(n)$ splošnega algoritma A tudi z začetnim pogojem:

$$T(n) = \begin{cases} b & \text{če } n = 1; \\ aT(\frac{n}{c}) + r(n) + s(n) & \text{če } n > 0, \end{cases} \tag{3}$$

kjer so $a \geqslant 1$, $b > 0$ in $c \geqslant 2$.

Analiza časovne zahtevnosti algoritma A

Kako izračunati funkcijo $T(n)$ iz enačbe (3)? Za začetek pišimo $f(n) = r(n) + s(n)$. Funkcija $f(n)$ je *skupen čas*, potreben za *razdelitev* naloge N v podnaloge *in sestavljanje* njihovih rešitev v rešitev naloge N. Seveda nas zanimajo funkcije $f(n)$, ki imajo *kvečjemu polinomsko rast*. V nadaljevanju najprej izračunamo $T(n)$ za splošno $f(n)$, potem pa še za $f(n)$, ki je polinom.

Računanje $T(n)$ za splošno $f(n)$

Izpeljavo bom izpustil, navajam le glavne korake in rezultat. Najprej predpostavimo, da je $n = c^m$ za neki $m \geqslant 0$. Ta predpostavka omogoči uporabo nekaj običajnih prijemov pri reševanju rekurzivnih enačb (zamenjava spremenljivk, seštevanje rekruzivnih enačb pri zaporednih argumetih), tako da iz (3) izpeljemo izraz za $T(n)$:

$$T(n) = n^{\log_c a}\left[b + \sum_{k=1}^{\log_c n} \frac{f(c^k)}{a^k}\right], \quad \text{kjer je } n > 1 \text{ in } T(1) = b. \tag{4}$$

Tega izraza si ni treba zapomniti; potrebovali ga bomo le pri naslednjem koraku.

Računanje $T(n)$ polinomsko $f(n)$

Zdaj pa v enačbo (4) vstavimo polinomsko funkcijo $f(n)$, tj. $f(n) = bn^d$, $d \geqslant 0$. Tedaj iz (4) z malo običajne telovadbe dobimo

$$T(n) = bn^{\log_c a}\left[1+\frac{c^d}{a}+\left(\frac{c^d}{a}\right)^2+\ldots+\left(\frac{c^d}{a}\right)^m\right], \quad \text{kjer je } n > 1 \text{ in } m = \log_c n. \quad (5)$$

To je sicer rešitev enačbe (3) za polinomsko funkcijo $f(n)$, vendar je nerodna v tem smislu, da ni očitno, kako hitro narašča $T(n)$. Pričakujemo lahko, da na to hitrost vplivajo parametri a, b, c, d. Kako pa vplivajo? Opazimo, da je v oglatih oklepajih geometrijska vrsta. Ker vemo, da je njena vsota odvisna od tega ali je $\frac{c^d}{a}$ več od 1, enako 1 ali manj od 1, analiziramo te alternative vsako posebej (izpeljave nimajo posebnih trikov, zato jih izpuščam):

1. Denimo, da je $\frac{c^d}{a} > 1$. Tedaj lahko izpeljemo, da je $T(n) = \Theta(n^d)$.
2. Denimo, da je $\frac{c^d}{a} = 1$. Tedaj lahko izpeljemo, da je $T(n) = \Theta(n^d \log n)$.
3. Denimo, da je $\frac{c^d}{a} < 1$. Tedaj lahko izpeljemo, da je $T(n) = \Theta(n^{\log_c a})$.

Prišli smo do ugotovitve, ki jo strnemo v izrek:

Izrek. (*Glavni izrek metode Deli in vladaj*) Za rešitev $T(n)$ enačbe

$$T(n) = \begin{cases} b & \text{če } n = 1; \\ aT(\frac{n}{c}) + bn^d & \text{če } n > 0, \end{cases}$$

kjer so $a \geqslant 1$, $b > 0$, $c \geqslant 2$ in $d \geqslant 0$, velja naslednje:

$$T(n) = \begin{cases} \Theta(n^d) & \text{če } \frac{c^d}{a} > 1; \\ \Theta(n^d \log n) & \text{če } \frac{c^d}{a} = 1; \\ \Theta(n^{\log_c a}) & \text{če } \frac{c^d}{a} < 1. \end{cases}$$

Primer

Preizkusimo izrek na algoritmu Quicksort. Če delilni element m razdeli tabelo vedno v dve enako veliki podtabeli, potem je algoritem Quicksort oblike A za $c = 2$. Ostali parametri so ne glede na m naslednji: $a = 2$ (ker je treba urediti obe podtabeli); b konstanta brez bistvenega vpliva; in $d = 1$ (ker ima razdelitev tabele v podtabeli zahtevnost $\Theta(n)$, sestavljanje končne urejene tabele iz urejenih podtabel pa $\Theta(1)$. Vidimo, da je $\frac{c^d}{a} = \frac{2^1}{2} = 1$. Po zgornjem izreku je $T(n) = \Theta(n^d \log n) = \Theta(n \log n)$, kar se sklada z analizo Quicksorta za najboljši primer.

Primer

Preizkusimo izrek na algoritmu BinSearch za dvojiško iskanje v urejeni tabeli. Algoritem poznamo: če je srednji element s tabele enak iskanemu x, je iskanje uspešno, sicer se iskanje ponovi v podtabeli levo oz. desno od s (če $x < s$ oz. $s < x$). BinSearch je torej oblike A. Očitno je $p = 2$ (vedno sta dve podtabeli); $c = 2$ (podtabeli sta enako veliki); $a = 1$ (iskanje se nadaljuje v eni podtabeli); $d = 0$ (razdelitev tabele na dve je trivialna; prav tako tudi sestavljanje končnega odgovora). Torej imamo $\frac{c^d}{a} = \frac{2^0}{1} = 1$, zato je po zgornjem izreku časovna zahtevnost algoritma BinSearch $T(n) = \Theta(n^d \log n) = \Theta(n^0 \log n) = \Theta(\log n)$, kar nam je že znano.

V nadaljevanju si bomo ogledali še nekaj računskih problemov, za katere bomo razvili algoritme z metodo Deli in vladaj.

3 Matrično množenje

Če sta A in B matriki reda $n \times n$, je njun produkt $C = A \cdot B$ matrika reda $n \times n$ s komponentami $C_{ij} = \sum_{k=1}^{n} A_{ik}B_{kj}$. Običajni algoritem za izračun matrike C je enostaven: po vrsti izračuna vseh n^2 komponent, vsako po definiciji C_{ij}:

```
procedure MatProdukt(A,B,C);
begin
  C := 0;
  for i = 1 to n do
    for j = 1 to n do
      for k = 1 to n do
        C[i,j] := C[i,j] + A[i,k]*B[k,j]
end.
```

Izračun C torej zahteva n^3 skalarnih množenj in $n^2(n-1)$ skalarnih seštevanj. Časovna zahtevnost algoritma je torej $\Theta(n^3)$. Aditivne operacije (seštevanje, odštevanje) so hitrejše od multiplikativnih (množenje, deljenje); lahko jih zanemarimo, ne da bi to bistveno vplivalo na asimptotično časovno zahtevnost algoritma.

Leta 1967 je Shmuel Winograd sestavil algoritem, ki za izračun $C = AB$ porabi $\frac{1}{2}n^3+n^2$ operacij skalarnega, kar je pri velikih n skoraj pol manj kot pri MatProdukt. Toda Winogradov algoritem ima še vedno *asimptotično* zahtevnost $\Theta(n^3)$.

Problem. Ali se da matrike množiti v času, asimptotično manjšem od $\Theta(n^3)$?

Ker je treba izračunati n^2 komponent matrike C, mora biti ta čas vsaj $\Theta(n^2)$.

Poskus z metodo Deli in vladaj

Poskusimo razviti hitrejši algoritem za matrično množenje po metodi Deli in vladaj. *Predpostavka*: n je potenca števila 2. (Tako bo možno deliti matriko na podmatrike.)

Vsako od matrik A, B si mislimo razdeljeno v štiri bločne podmatrike reda $\frac{n}{2} \times \frac{n}{2}$. Tedaj matrični produkt AB lahko izrazimo z matričnimi produkti teh podmatrik:

$$C = AB = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix} = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix}$$

kjer so

$$\begin{aligned} C_{11} &= A_{11}B_{11} + A_{12}B_{21} \\ C_{12} &= A_{11}B_{12} + A_{12}B_{22} \\ C_{21} &= A_{21}B_{11} + A_{22}B_{21} \\ C_{22} &= A_{21}B_{12} + A_{22}B_{22} \end{aligned}$$

Nalogo AB smo razdelili na osem podnalog $A_{ij}B_{kl}$ enake vrste (matrično množenje). Zato bo naš algoritem za računanje AB osemkrat klical samega sebe, dobljene rešitve podnalog pa sestavil tako, kot opisujejo zgornje enačbe za komponente C_{ij}.

Kakšna je časovna $T(n)$ zahtevnost tega algoritma? Čas $T(n)$ za izračun matričnega produkta $C = AB$ je sestavljen iz časa $8T(\frac{n}{2})$, ki je potreben za izračun osmih matričnih produktov $A_{ij}B_{kl}$, in časa $4\frac{n}{2}\frac{n}{2}$, ki je potreben za izračun štirih matričnih vsot (vsaka zahteva $\frac{n}{2}\frac{n}{2}$ sklarnih vsot). Torej je $T(n) = 8T\left(\frac{n}{2}\right) + 4\frac{n}{2}\frac{n}{2}$ oz.

$$T(n) = 8T\left(\frac{n}{2}\right) + n^2. \tag{1}$$

Ali je časovna zahtevnost $T(n)$ tako razvitega algoritma manjša od $\Theta(n^3)$? Glavni izrek metode Deli in vladaj nam za funkcijo $T(n)$ pove naslednje: ker so v enačbi (1) parametri $p = 8, a = 8, b = 1, c = 2$ in $d = 2$, je $\frac{c^d}{a} = \frac{2^2}{8} = \frac{1}{2} < 1$, je

$$T(n) = \Theta(n^{\log_c a}) = \Theta(n^{\log_2 8}) = \Theta(n^3).$$

To je razočaranje! Asimptotična časovna zahtevnost našega, z metodo Deli in vladaj razvitega algoritma *ni manjša* od časovne zahtevnosti algoritma `MatProdukt`.

Da bi dosegli asimptotično boljši $T(n)$, bi morali najti take a, c, d, ki bi izpolnili zahtevo $\frac{c^d}{a} < 1 \wedge \log_c a < 3$. Taka števila je lahko najti: npr., če ohranimo $c = 2$ in $d = 2$, zahtevo izpolni vsak $a \leqslant 7$. Težje pa je odkriti algoritem vrste Deli in vladaj, ki bi imel parametre $a = 7, c = 2, d = 2$. Kajti to pomeni, da bi moral tak algoritem sestaviti C (štiri podmatrike C_{ij}) s pomočjo le *sedmih* produktov $A_{ij}B_{kl}$. *Je tak algoritem sploh možen?*

Strassenovo matrično množenje

Volker Strassen je 1969 odgovoril na zagornje vprašanje pritrdilno. Bistvo njegovega algoritma je drugačno računanje podmatrik C_{ij}. Tule je njegovo odkritje:

Najprej izračunamo matrične produkte

$$\begin{aligned} P_{11} &= (A_{11} + A_{22})(B_{11} + B_{22}) & P_{21} &= (A_{22} + A_{11})(B_{22} + B_{11}) \\ P_{12} &= (A_{12} - A_{22})(B_{21} + B_{22}) & P_{22} &= (A_{21} - A_{11})(B_{12} + B_{11}) \\ P_{13} &= (A_{11} + A_{12})B_{22} & P_{23} &= (A_{22} + A_{21})B_{11} \\ P_{14} &= A_{22}(B_{11} - B_{21}) & P_{24} &= A_{11}(B_{22} - B_{12}) \end{aligned}$$

nato pa iz njih podmatrike

$$\begin{aligned} C_{11} &= P_{11} + P_{12} - P_{13} - P_{14} \\ C_{12} &= P_{13} - P_{24} \\ C_{21} &= P_{23} - P_{14} \\ C_{22} &= P_{21} + P_{22} - P_{23} - P_{24} \end{aligned}$$

Toda Strassenova definicija teh izrazov je taka, da sta produkta P_{11} in P_{21} *enaka*! Ker bo treba računati le enega od njiju, bo matričnih množenj podmatrik *sedem*! Strassenov algoritem ima naslednjo strukturo:

```
procedure Strassen(A,B) return C;
begin
  if n=1 then C := AB else
   begin
     P_11 := Strassen(A_11 + A_22, B_11 + B_22)
     ...
     C_11 := P_11 + P_12 - P_13 - P14
     ...
     C := (C_11,C_12,C_21,C_22)
   end
end.
```

Poglejmo, kako je s časovno zahtevnostjo Strassenovega algoritma. Za izračun vseh podmatrik C_{ij} je potrebnih 7 matričnih množenj reda $\frac{n}{2}$ (pri računanju P_{ij}) in 18 matričnih vsot/razlik (od tega 10 pri računanju P_{ij} in 8 pri računanju C_{ij}). Torej je $T(n) = 7\,T(\frac{n}{2}) + 18\frac{n}{2}\frac{n}{2}$, oz.

$$T(n) = 7\,T\Big(\frac{n}{2}\Big) + 4.5\,n^2$$

Zdaj so parametri te enačbe $a = 7, b = 4.5, c = 2$ in $d = 2$. Ker je $\frac{c^d}{a} = \frac{2^2}{7} = \frac{4}{7} < 1$, je

$$T(n) = \Theta(n^{\log_c a}) = \Theta(n^{\log_2 7}) = \Theta(n^{2.80735}).$$

Strassenovo množenje matrik ima asimptotično časovno zahtevnost manjšo od $\Theta(n^3)$.

Kaj pravi praksa? Zaradi skritih koeficientov se Strassenov algoritem splača uporabiti šele pri $n \geqslant 45$ in gostih matrikah. Algoritem pa ima velik teoretični pomen, saj je bil prvi algoritem za množenje matrik asimptotično hitrjši od $\Theta(n^3)$. Razvoj je šel dalje; danes vemo, da se da matrike množiti v času $\Theta(n^{2.373})$, a največja spodnja meja za eksponent še ni znana.

4 k-ti največji element

Problem: V neurejeni tabeli t z n elementi poišči k-ti največji element ($1 \leqslant k \leqslant n$).

Poskusimo najti kako spodnjo mejo in kako zgornjo mejo za časovna zahtevnost $T(n)$ tega problema. Meji nam bosta pomagali, da ne bomo iskali algoritma, ki sploh ne obstaja, pa tudi ne algoritma, ki je pretirano počasen. Najbolje bi bilo, če bi našli *tesni* meji, tj. *največjo spodnjo* in *najmanjšo zgornjo* mejo za $T(n)$. Zavedati pa se moramo tudi, da je iskanje tesnih mej lahko zelo zahtevna naloga. Pa začnimo.

Na rešitev našega problema lahko vpliva vsak element tabele. Zato vsak algoritem za ta problem *mora* prebrati vseh n elementov. Časovna zahtevnost vsakega algoritma bo torej *vsaj* linearna, tj. $T(n) = \Omega(n)$. Ker je to tesna spodnja meja za $T(n)$, ne bomo iskali algoritmov z manjšo časovno zahtevnostjo od $\Theta(n)$.

Po drugi strani pa lahko hitro sestavimo algoritem, ki reši naš problem: algoritem uredi tabelo t v času $\Theta(n \log n)$ (npr. s Heapsortom) in vrne element na njenem k-tem mestu. Torej bomo iskali algoritem s časovno zahtevnostjo kvečjemu $\Theta(n \log n)$, tj. $T(n) = O(n \log n)$.

Metoda Deli in vladaj nam za tak algoritem da naslednjo zamisel:

- Izberimo število m (*delilni element*) in razdelimo tabelo t na tri podtabele t_1, t_2, t_3, kjer bodo v t_1 vsi elementi $< m$, v t_2 vsi $= m$, v t_3 pa vsi $> m$.
- Iz velikosti prvih dveh podtabel, tj. $|t_1|$ in $|t_2|$, se da ugotoviti, v kateri od t_1, t_2, t_3 je k-ti element po velikosti. Kako? Če je $k \leqslant |t_1|$, potem je v t_1; če je $|t_1| \leqslant k \leqslant |t_1| + |t_2|$, potem je v t_2; in če $k > |t_1| + |t_2|$, potem je v t_3.
- Ko izvemo, v kateri od t_1, t_2, t_3 je iskani element, nadaljujemo iskanje v njej. Toda pozor: če je v t_1, nadaljujemo z iskanjem k-tega v t_1; če je v t_2, potem je to m; in če je v t_3, potem nadaljujemo z iskanjem $k - |t_1| - |t_2|$-tega v t_3.

Torej algoritem nadomesti reševenje naloge z reševanjem ene od treh podnalog.

Algoritem je zapisan v spodnji psevdokodi. Dodali smo še uvodni del, ki poskrbi za reševanje trivialnih nalog (tj. tabel manjših od neke, še neznane velikosti $n_{\min}$.)

```
procedure Isci(k,t) returns int;
begin
  if |t| < n_min then
    begin
      Uredi(t);
      return(t[k])
    end
  else
    begin
      m := Izberi(t);
      Razdeli(t,t_1,t_2,t_3);
      if k <= |t_1| then Isci(k,t_1)
      else if k <= |t_1|+|t_2| then return(m)
      else Isci(k-|t_1|-|t_2|,t_3)
    end
end.
```

Procedura `Uredi` je lahko npr. Heapsort. Procedura `Razdeli` je podobna tisti pri Quicksortu, le da v času $\Theta(n)$ vrne tri podtabele. Bralec naj jo razvije za vajo. Dolžni pa smo še odgovoriti na troje:

Koliko je n_{min}? Kako izbrati m? Kakšna je časovna zahtevnost takega algoritma?

Naivni algoritem

Poskusimo priti do odgovora po najlažji poti: naj bosta $n_{\min} = 1$ in m naključno izbran element tabele t. Kakšno časovno zahtevnost $T(n)$ ima tedaj ta algoritem?

Ocenimo $T(n)$ za *najslabši primer*. V pesimističnem scenariju je m največji element v t, zato dobimo v podtabeli t_1 $n-1$ elementov, v t_2 le m, t_3 pa je prazna. Po pesimističnem scenariju se bo iskanje nadaljevalo v t_1, kjer se bo zgodilo podobno. Torej bo v najslabšem primeru za $T(n)$ veljalo

$$T(n) = T(n-1) + \Theta(n).$$

To rekurzivno enačbo smo srečali pri analizi Quicksorta za najslabši primer, zato vemo, da za njeno rešitev velja

$$T(n) = \Theta(n^2).$$

Toda mi že vemo, da s Heapsortom k-ti element vedno najdemo v času $\Theta(n \log n)$.

Izboljšani (Blum-Floyd-Pratt-Rivest-Tarjanov) algoritem

Pri naivnem algoritmu nas moti, da ima v najslabšem primeru *kvadratno* časovno zahtevnost. To bi radi izboljšali. Videti je, da je vzrok za to v naključnem izbiranju delilnega elementa m. Toda tudi če bi m izbirali po kakem drugem enostavnem pravilu (npr. m naj bo prvi, zadnji ali srednji element v t), bi v najslabšem primeru dobili kvadratno časovno zahtevnost. Delilni element m bi bilo treba izbirati tako, da nobena od podtabel ne bi bila „prevelika" (in zato druga „premajhna"). Skratka, tako kot smo videli pri opisu metode Deli in vladaj, bi morali biti t_1 in t_3 (vsaj približno) enako veliki, saj bi bilo potem vseeno, v kateri bi se nadaljevalo iskanje. Kaj pa tabela t_2? Ta nas ne skrbi, saj se „iskanje" po t_2 takoj konča z rezultatom m. Vprašanje je torej:

Kako zagotoviti, da bosta podtabeli t_1 in t_3 vedno približno enaki.

Izberimo delilni element m z novo proceduro `Izberi` takole:

1. Tabelo t si mislimo razdeljeno v peterke zaporednih elementov.
2. V vsaki peterki poiščimo njeno mediano (tretji največji element).
3. Naj bo M tabela vseh dobljenih median. Potem naj bo m mediana tabele M.

Tako izračunani m uporabimo v proceduri `Razdeli`, ki razdeli t v podtabele t_1, t_2, t_3. Prednosti tako izračunanega m razkrije naslednja trditev:

Trditev. Po razdelitvi tabele t je $|t_1| \leqslant \frac{3}{4}|t|$ in $|t_3| \leqslant \frac{3}{4}|t|$.

Dokaz. Dokaz je delno grafičen a ne zelo težak. Vseeno ga v teh zapiskih izpuščam. □

Torej, če je m izračunan z novo procediro `Izberi`, nobena od t_1 in t_3 ni „premajhena", saj se zaradi zgornje trditve nobena ne more pretirano povečati v škodo druge.

Preden lahko analiziramo časovno zahtevnost $T(n)$ izboljšanega algoritma moramo pojasniti, kako naj procedura `Izberi` izračuna m. Mediano posamične peterke (točka 2.) lahko določimo v konstantnem času $O(1)$ z nekaj primerjanji petih elementov (domača naloga). Ker je vseh peterk $\lceil \frac{n}{5} \rceil$, priprava tabele zahteva $\text{konst} \cdot \lceil \frac{n}{5} \rceil = O(n)$ časa. Kako pa določiti mediano $\lceil \frac{n}{5} \rceil$ elementov tabele M? Po definiciji je mediana element, od katerega je manjših polovica elementov tabele. Mediana m je zato $\lceil \frac{|M|}{2} \rceil$-ti največji element tabele M, kjer je $|M| = \lceil \frac{n}{5} \rceil$. Torej bo morala procedura `Izberi` rekurzivno poklicati `Isci(|M|/2,M)`, da ji ta izračuna m.

Zdaj pa analizirajmo časovno zahtevnost $T(n)$ za najslabši primer. Zakaj ravno za *najslabši* primer? Razlog je ta, da želimo ugotoviti ali novo izbiranje m izboljša najslabšo časovno zahtevnost $\Theta(n^2)$ naivnega algoritma.

Torej naj bo $T(n)$ najslabša časovna zahtevnost izboljšanega algoritma za iskanje k-tega elementa v tabeli velikosti n. Skladno z algoritmom `Isci` obravnavamo dve situaciji: prvo, ko je $n \geqslant n_{\min}$ in drugo, ko je $n < n_{\min}$:

- $[n \geqslant n_{\min}]$ V tem primeru časovno zahtevnost $T(n)$ sestavlja več prispevkov:

  $$\begin{aligned} T(n) &= \Theta(n) && //\text{čas, potreben za izračun } M \\ &\quad + T(\tfrac{n}{5}) && //\text{čas, potreben za izračun } m \\ &\quad + \Theta(n) && //\text{čas, potreben za razdelitev } t \text{ v } t_1, t_2, t_3 \\ &\quad + T(\max\{|t_1|, |t_3|\}) && //\text{čas, potreben za iskanje po večji od } t_1, t_3 \end{aligned}$$

  Če upoštevamo še zgornji izrek, dobimo $T(n) \leqslant T(\frac{n}{5}) + c\,n + T(\frac{3}{4}n)$, kjer je $c > 0$ konstanta.

- $[n < n_{\min}]$ Če je $n_{\min}$ tista mejna velikost, do katere je urejanje tabele t z $n < n_{\min}$ elementi hitrejše od cn, potem je v tem primeru $T(n) \leqslant cn$. (Število $n_{\min}$ najdemo eksperimentalno; literatura navaja $n_{\min} = 50$.)

Zdaj iz obeh situacij sestavimo kombinirano najslabšo časovno zahtevnost:

$$T(n) \leqslant \begin{cases} cn & \text{če } n < n_{\min}; \\ T(\frac{n}{5}) + c\,n + T(\frac{3}{4}n) & \text{če } n \geqslant n_{\min}. \end{cases}$$

Kako hitro narašča ta funkcija $T(n)$? Presenečenje nam pripravi naslednja trditev:

Trditev. $T(n) \leqslant 20cn$.

Dokaz. Najprej dokažemo, da trditev velja za vse $n \leqslant 50$. Nato za osnovo indukcije vzamemo trditev pri $n = 50$. Veljavnost indukcijskega koraka z n na $n + 1$ preverimo pri $n \geqslant 50$. (Podrobnosti tukaj opustim in jih prepustim bralcu za vajo.) □

Na začetku razdelka smo videli, da iskanje k-tega elementa v tabeli velikosti n zahteva vsaj $\Omega(n)$ časa. Pravkar pa smo dokazali, da to iskanje zahteva največ $O(n)$ časa. Ker sta spodnja in zgornja meja časovne zahtevnosti tega problema enaki, ima ta problem časovno zahtevnost $\Theta(n)$. Ker ima izboljšani algoritem to časovno zahtevnost, je *asimpotično optimalen.*

Opomba. Problem iskanja k-tega največjega elementa smo rešili z metodo Deli in vladaj. V nadaljevanju si bomo ogledali še en problem, ki je rešljiv s to metodo, pri katerem so vse faze metode (razdelitev na podprobleme in sestavljanje delnih rešitev) netrivialne.

5 Diskretna Fourierova transformacija

Diskretna Fourierova transformacija ima veliko vlogo v uporabi. Algoritem za njen izračun bomo razvili z metodo Deli in vladaj. Pred tem pa bralca motiviramo.

Motivacija

Problem: Koliko operacij množenja zahteva izračun produkta $r(x) = p(x)q(x)$, kjer sta $p(x)$ in $q(x)$ dana polinoma stopenj n in m.

Ocenimo število množenj, ki jih zahteva izračun $r(x)$ po običajni poti. Pišimo

$$p(x) = \sum_{i=0}^{n} a_i x^i \qquad \text{in} \qquad q(x) = \sum_{i=0}^{m} b_i x^i.$$

Tedaj je

$$r(x) = p(x)q(x) = \left(\sum_{i=0}^{n} a_i x^i\right)\left(\sum_{i=0}^{m} b_i x^i\right).$$

Produkt vsot je polinom spremenljivke x in stopnje $n + m$,

$$r(x) = \sum_{i=0}^{n+m} c_i x^i,$$

s koeficienti c_i, ki jih moramo dobiti iz koeficientov polinomov $p(x)$ in $q(x)$. Kako? Potenca x^i (ob c_i) lahko nastane le iz produktov potenc x^k in x^{i-k} (ob a_k in b_{i-k}), kjer je $k = 0, \ldots, i$. Zato za koeficiente c_i, kjer je $i = 0, \ldots, n + m$, velja

$$c_i = \sum_{k=0}^{i} a_k b_{i-k},$$

kar zahteva $i + 1$ množenj. Izračun vseh $n + m + 1$ koeficientov c_i zato zahteva $\sum_{i=0}^{n+m}(i+1)$ množenj, kar je reda $\Theta((n+m)^2)$. (Izpeljavo prepuščam bralcu za vajo).

Sklep: *Izračun produkta $r(x) = p(x)q(x)$ zahteva kvečjemu $O((n + m)^2)$ množenj.*

Vprašanje: Ali se da produkt $r(x)$ izračunati z asimptotično manj množenji?

Zamisel

Polinom $p(x) = \sum_{i=0}^{n} a_i x^i$ lahko predstavimo v obliki $(a_0, \ldots, a_n)$. Tej obliki bomo rekli *koeficientna predstavitev* (k.p.) polinoma $p(x)$. Koeficientna predstavitev polinoma $q(x)$ je torej $(b_0, \ldots, b_m)$.

Polinome pa lahko podamo tudi z njihovimi vrednostmi v izbranih točkah. Na primer, $p(x)$ bi bil lahko dan v obliki $(p_0, \ldots, p_n)$, kjer so p_i njegove vrednosti v $n+1$ paroma različnih točkah $t_0, \ldots, t_n$. Temu bomo rekli *vrednostna predstavitev* (v.p.) polinoma $p(x)$. Tudi $q(x)$ bi bil lahko podan s svojo vrednostno predstavitvijo $(q_0, \ldots, q_m)$ v $m+1$ točkah $t'_0, \ldots, t'_m$ (ki so lahko tudi $t_0, \ldots, t_m$).

Primer. Naj bo $p(x) = 3x^4 + 5x^2 - 2x + 1$. Tedaj je $(1, -2, 5, 0, 3)$ njegova k.p., njegova v.p. v točkah $(2, 3, 4, 5, 6)$ pa $(65, \ldots, \ldots, \ldots, \ldots)$. □

Predpostavimo, da sta polinoma $p(x)$ in $q(x)$ dana z vrednostnima predstavitvama

$$p(x) = (p_0, \ldots, p_n) \qquad \text{in} \qquad q(x) = (q_0, \ldots, q_m),$$

kjer je $p_i = p(t_i)$ za $i = 0, \ldots, n$ in $q_j = q(t_j)$ za $j = 0, \ldots, m$.

Kako zdaj izračunamo produkt $r(x) = p(x)q(x)$? Koliko množenj je za to potrebnih?

Ker sta $p(x)$ in $q(x)$ vrednostno predstavljena, lahko domnevamo, da bomo zadovoljni tudi z vrednostno predstavljenim produktom $r(x)$. Torej iščemo

$$r(x) = (r_0, \ldots, r_s),$$

kjer je $r_i = r(t_i) = p(t_i)q(t_i) = p_i q_i$ za $i = 0, \ldots, s$. Koliko je s, stopnja polinoma $r(x)$? Stopnji polinomov $p(x)$ in $q(x)$ sta n in m (ker imata njuni vrednostni predstavitvi $n+1$ in $m+1$ elementov), zato je stopnja s polinoma $r(x)$ enaka

$$s = n + m.$$

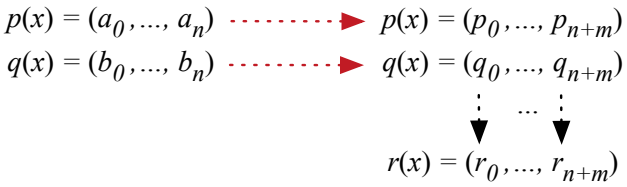
Torej moramo izračunati $n + m$ vrednosti $r_0, \ldots, r_{n+m}$.

Ker je $r_i = p_i q_i$, moramo množiti istoležne vrednosti iz $(p_0, \ldots, p_n)$ in $(q_0, \ldots, q_m)$. Zdaj opazimo težavo: n in m sta lahko različna in vsak od njiju je manjši od $n + m$. Za izračun $n + m + 1$ vrednosti r_i bi morali vrednostni predstavitvi polinomov $p(x)$ in $q(x)$ vsebovati po $n + m + 1$ vrednosti teh polinomov (ne pa le $n + 1$ in $m + 1$). Zato vsako vrednostno predstavitev dopolnimo z vrednostmi njenega polinoma v dodatnih točkah. Dobimo $p(x) = (p_0, \ldots, p_{n+m})$ in $q(x) = (q_0, \ldots, q_{n+m})$. Zdaj lahko izračunamo $r(x) = (r_0, \ldots, r_{n+m})$ z $n+m+1$ množenji: $r_i := p_i q_i$, $i = 0, \ldots, m+n$.

Sklep: *Z uporabo v.p. lahko izračunamo $r(x) = p(x)q(x)$ s $\Theta(n + m)$ množenji!*

Zmanjšanje števila množenj s $\Theta((n+m)^2)$ na $\Theta(n+m)$ smo dosegli z uporabo drugačne predstavitve polinomov $p(x), q(x)$ in $r(x)$.

Kaj pa če $p(x)$ in $q(x)$ ne bi bila dana v vrednostni temveč koeficientni predstavitvi? Potem bi morali njuni koeficientni predstavitvi najprej *pretvoriti* v vrednostni.



Slika 5.1 .

Da pa bi se to izplačalo, bi morala pretvorba k.p. obeh polinomov zahtevati *asimptotično manj* kot $\Theta((n+m)^2)$ množenj, sicer bi lahko izračunali $r(x) = p(x)q(x)$ kar s postopkom iz razdelka z motivacijo.

Ocenimo, koliko množenj bi morala zahtevati pretvorba k.p. *enega* polinoma, da bi se pretvarjanje obeh izplačalo. Kadar imata polinoma isto stopnjo $s = n = m$, zahteva postopek, ki smo ga opisali pri motivaciji, $\Theta((n+m)^2) = \Theta(4s^2) = \Theta(s^2)$ množenj. Če bi pretvorba predstavitve enega polinoma zahtevala $\Theta(s^2)$ množenj, bi naša zamisel zahtevala skupaj $\Theta(s^2) + \Theta(s^2) + \Theta(2s)$ množenj, kar je tudi reda $\Theta(s^2)$. Sledi, da potrebujemo postopek, ki bo k.p. polinoma stopnje s pretvoril v v.p. z *asimptotično manj kot* $\Theta(s^2)$ *množenji.* Torej moramo rešiti nov problem:

Kako pretvoriti k.p. polinoma stopnje s v v.p. z uporabo manj kot $\Theta(s^2)$ množenj?

Za računanje vrednosti polinomov že imamo na voljo znani *Hornerjev postopek.* Spomnimo se ga na polinomu iz prejšnjega primera.

Primer. Naj bo $p(x) = 3x^4 + 5x^2 - 2x + 1$ in $t_0 = 2$. Po Hornerju izračunamo $p(t_0)$ takole: $p(t_0) = (((3t_0 + 0)t_0 + 5)t_0 - 2)t_0 + 1 = (((3 \cdot 2 + 0) \cdot 2 + 5) \cdot 2 - 2) \cdot 2 + 1 = 65$. □

V splošnem dobimo vrednost $p(x) = a_n x^n + a_{n-1}x^{n-1} + a_{n-2}x^{n-2} + \ldots + a_1 x + a_0$ pri $x = t$ takole: $p(t) = (\ldots((a_n \cdot t + a_{n-1}) \cdot t + a_{n-2}) \cdot t + \ldots a_1) \cdot t + a_0$. Torej izračun $p(t)$ zahteva n množenj. Ker pa je za pretvorbo k.p. $p(x)$ v v.p. treba izračunati $n+1$ vrednosti $p(t_0), p(t_1), \ldots, p(t_n)$, bi pretvorba zahtevala $n(n+1) = \Theta(n^2)$ množenj. Ugotovili smo, da *uporaba Hornerjevega postopka ne reši naše naloge.*

Najti moramo algoritem za konstrukcijo vrednostne predstavitve polinoma, ki bo asimptotično hitrejši od pravkar opisanega. Ali tak algoritem sploh obstaja? Da. V zgodbo vstopi *Diskretna Fourierova transformacija* (DFT), ki bo skupaj z metodo *Deli in vladaj* omogočila razvoj takega algoritma. Algoritem se bo imenoval *Hitra Fourierova transformacija* (FFT), njegova časovna zahtevnost pa bo $\Theta(n \log n)$.

Diskretna Fourierova transformacija (DFT) ...

Najprej opišimo DFT v splošnem. Pri DFT igra pomembno vlogo matematična struktura imenovana vektorski prostor. Predpostavljamo, da se je bralec z njim že srečal, zato se ga bomo le na hitro spomnili.

Vektorski prostor V nad obsegom $\mathbb{C}$ sestavljajo Abelova grupa $(V, +)$ vektorjev, obseg $(\mathbb{C}, +, *)$ skalarjev in operacija $\cdot : \mathbb{C} \times V \to V$ množenja vektorjev s skalarji, ki je uglašena z operacijami v V in $\mathbb{C}$. Dimenzijo prostora V označimo z $d = \dim V$. Obseg $\mathbb{C}$ ima *enoto* 1, tj. skalar z lastnostjo, da je $1 * \alpha = \alpha * 1 = \alpha$ za vsak skalar α.

Linearna transformacija vektorskega prostora V vase je preslikava $T : V \to V$ z lastnostjo $T(\alpha \cdot u + \beta \cdot v) = \alpha \cdot T(u) + \beta \cdot T(v)$, za poljubna vektorja u, v in poljubna skalarja α, β. Z besedami je transformacija linearne kombinacije vektorjev enaka linerani kombinaciji transformacij vektorjev. Transformacijo končnodimenzionalnega vektorskega prostora lahko predstavimo s končnodimenzionalno matriko. Kadar obstaja njena inverzna matrika, obstaja tudi inverzna transformacija.

Transformacij vektorskega prostora je veliko, nas pa bo zanimala transformacija, imenovana DFT. Ključno vlogo pri DFT bo odigral *primitivni koren enote*. To je skalar ω, ki ga definirata dve lastnosti:

1. $\omega^d = 1$ in // ω je *koren* enote
2. $\omega^k \neq 1$ za $k = 1, \ldots, d-1$. // ω je *primitiven* koren enote

V obsegu $\mathbb{C}$ je

$$\omega = e^{\iota \frac{2\pi}{d}}. \qquad (\iota \text{ je imaginarna enota})$$

Diskretna Fourierova transformacija (DFT) je linearna transformacija prostora V, ki je predstavljena z matriko F reda $d \times d$, katere komponente so

$$F_{ij} = \omega^{i*j} \quad , \quad 0 \leqslant i, j \leqslant d-1.$$

Inverzna DFT je predstavljena z matriko F^{-1} reda $d \times d$, katere komponente so

$$F^{-1}_{ij} = \frac{1}{d}\omega^{-i*j} \quad , \quad 0 \leqslant i, j \leqslant d-1.$$

Dokaz. Da je F^{-1} res inverz matrike F dokažemo tako, da preverimo, ali je $FF^{-1} = I$. □

Primer

Izračunajmo matriko F za $d = 2$. Naprej izračunamo $\omega = e^{\iota \frac{2\pi}{d}} = e^{\iota \frac{2\pi}{2}} = e^{\iota\pi} = -1$. Zato sta $\omega^0 = 1$ in $\omega^1 = -1$. Nato izračunamo vse vrednosti ω^{i*j} pri $0 \leqslant i, j \leqslant d-1$. Dobimo $\omega^{0*0} = 1; \omega^{0*1} = 1; \omega^{1*0} = 1; \omega^{1*1} = -1$. Torej je matrika $F = \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}$. Izračunajmo še inverzno matriko F^{-1}. Njene komponente so $F^{-1}_{ij} = \frac{1}{d}\omega^{-i*j}$. Ker so $\omega^{-0*0} = 1; \omega^{-0*1} = 1; \omega^{-1*0} = 1; \omega^{-1*1} = -1$, je $F^{-1} = \frac{1}{2}\begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}$. □

... in njena uporaba

Vrnimo se k problemu pretvorbe koeficientne predstavitve $(a_0, \ldots, a_n)$ polinoma $p(x)$ v vrednostno predstavitev $(p_0, \ldots, p_n)$ v točkah $(t_0, \ldots, t_n)$ ter pretvorbe k.p. $(b_0, \ldots, b_m)$ polinoma $q(x)$ v v.p. $(q_0, \ldots, q_m)$ v točkah $(t_0, \ldots, t_m)$.

Naj bo V vektorski prostor nad $\mathbb{C}$, ki vsebuje vse vektorje z $n+1$ komponentami. Torej je $\dim V = d = n+1$. Tudi k.p. $(a_0, \ldots, a_n)$ je tak vektor in zato element V. Izračunajmo, v kaj ga preslika DFT reda d?

$$F((a_0, \ldots, a_n)) = \begin{pmatrix} \omega^{0*0} & \cdots & \omega^{0*j} & \cdots & \omega^{0*n} \\ \vdots & & \vdots & & \vdots \\ \omega^{i*0} & \cdots & \omega^{i*j} & \cdots & \omega^{i*n} \\ \vdots & & \vdots & & \vdots \\ \omega^{n*0} & \cdots & \omega^{n*j} & \cdots & \omega^{n*n} \end{pmatrix} \begin{pmatrix} a_0 \\ \vdots \\ a_j \\ \vdots \\ a_n \end{pmatrix} = \begin{pmatrix} \sum_{j=0}^{n} a_j \omega^{0*j} \\ \vdots \\ \sum_{j=0}^{n} a_j \omega^{i*j} \\ \vdots \\ \sum_{j=0}^{n} a_j \omega^{n*j} \end{pmatrix} =$$

$$= \begin{pmatrix} \sum_{j=0}^{n} a_j (\omega^0)^j \\ \vdots \\ \sum_{j=0}^{n} a_j (\omega^i)^j \\ \vdots \\ \sum_{j=0}^{n} a_j (\omega^n)^j \end{pmatrix} = \begin{pmatrix} p(\omega^0) \\ \vdots \\ p(\omega^i) \\ \vdots \\ p(\omega^n) \end{pmatrix}$$

Ugotovili smo, da DFT reda $n+1$ preslika koeficientno predstavitev polinoma $p(x)$ stopnje n v njegovo vrednostno predstavitev na točkah $\omega^0, \ldots \omega^n$. Na enak način bi ugotovili, da DFT reda $m+1$ preslika koeficientno predstavitev polinoma $q(x)$ stopnje m v njegovo vrednostno predstavitev na točkah $\omega^0, \ldots \omega^m$. V splošnem torej DFT pretvori k.p. polinoma v njegovo v.p. v kompleksnih točkah $t_i = \omega^i$. Zdaj lahko dopolnimo Sliko 5.1 v Sliko 5.2.

Slika 5.2 .

Primer

Naj bo $p(x) = 2x^3 + 2x^2 + x + 1$. Njegova koeficientna predstavitev je (1,1,2,2). Kakšna je vrednostna predstavitev polinoma $p(x)$, ki jo dobimo z DFT? Računajmo. Stopnja polinoma je $n = 3$, zato bo DFT reda $d = n+1 = 4$. Primitivni koren enote za $d = 4$ je $\omega = e^{\iota\frac{2\pi}{d}} = e^{\iota\frac{2\pi}{4}} = e^{\iota\frac{\pi}{2}} = \iota$. Fourierova matrika reda $d \times d = 4 \times 4$ je

$$F = \begin{pmatrix} \omega^{0*0} & \omega^{0*1} & \omega^{0*2} & \omega^{0*3} \\ \omega^{1*0} & \omega^{1*1} & \omega^{1*2} & \omega^{1*3} \\ \omega^{2*0} & \omega^{2*1} & \omega^{2*2} & \omega^{2*3} \\ \omega^{3*0} & \omega^{3*1} & \omega^{3*2} & \omega^{3*3} \end{pmatrix} = \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & \iota & -1 & -\iota \\ 1 & -1 & 1 & -1 \\ 1 & -\iota & -1 & \iota \end{pmatrix}$$

Koeficientna predstavitev (1,1,2,2) polinoma $p(x)$ se preslika v vrednostno predstavitev polinoma v točkah $(t_0, t_1, t_2, t_3) = (\omega^0, \omega^1, \omega^2, \omega^3) = (1, \iota, -1, -\iota)$ takole:

$$F\Big((1,1,2,2)\Big) = \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & \iota & -1 & -\iota \\ 1 & -1 & 1 & -1 \\ 1 & -\iota & -1 & \iota \end{pmatrix} \begin{pmatrix} 1 \\ 1 \\ 2 \\ 2 \end{pmatrix} = \begin{pmatrix} 6 \\ -1-\iota \\ 0 \\ -1+\iota \end{pmatrix} = \begin{pmatrix} p(1) \\ p(\iota) \\ p(-1) \\ p(-\iota) \end{pmatrix}$$

Zdaj pa preizkusimo še inverzno DFT. Ta bi morala preslikati pravkar izračunano vrednostno predstavitev nazaj v koeficientno predstavitev. Inverzna matrika matrike F je po definiciji

$$F^{-1} = \frac{1}{4}\begin{pmatrix} \omega^{-0*0} & \omega^{-0*1} & \omega^{-0*2} & \omega^{-0*3} \\ \omega^{-1*0} & \omega^{-1*1} & \omega^{-1*2} & \omega^{-1*3} \\ \omega^{-2*0} & \omega^{-2*1} & \omega^{-2*2} & \omega^{-2*3} \\ \omega^{-3*0} & \omega^{-3*1} & \omega^{-3*2} & \omega^{-3*3} \end{pmatrix} = \frac{1}{4}\begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & -\iota & -1 & \iota \\ 1 & -1 & 1 & -1 \\ 1 & \iota & -1 & -\iota \end{pmatrix}$$

Če z F^{-1} transformiramo vrednostno predstavitev $(6, -1-\iota, 0, -1+\iota)$, dobimo

$$F^{-1}\Big((6, -1-\iota, 0, -1+\iota)\Big) = \frac{1}{4}\begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & -\iota & -1 & \iota \\ 1 & -1 & 1 & -1 \\ 1 & \iota & -1 & -\iota \end{pmatrix} \begin{pmatrix} 6 \\ -1-\iota \\ 0 \\ -1+\iota \end{pmatrix} = \begin{pmatrix} 1 \\ 1 \\ 2 \\ 2 \end{pmatrix}$$

kar je koeficientna predstavitev (1,1,2,2) polinoma $p(x)$.

Algoritem Hitra Fourierova transformacija (FFT)

Izračun DFT z običajnim množenjem matrike in vektorja bi zahteval $\Theta(n^2)$ množenj. To je preveč, zato se vprašamo, ali se da zmnožiti matriko F in vektor hitreje? Odgovor je *da*. Če izkoristimo lastnosti števila ω, lahko z metodo Deli in vladaj razvijemo algoritem FFT, ki ima časovno zahtevnost $\Theta(n \log n)$. Poglejmo kako.

Algoritem FFT bomo opisali na polinomu $p(x)$.[1] Pri tem bomo predpostavili, da je v njegovi koeficientni predstavitvi $(a_0, \ldots, a_n)$ *sodo* mnogo komponent.[2] *Predpostavka.* $d = n + 1 = 2r$, kjer $r \in \mathbb{N}$.

Najprej problem poimenujmo. Če bomo v njem odkrili podprobleme iste vrste, se bomo nanje sklicevali z enakim imenom, in to bo vodilo do rekurzivnega algoritma.

Problem $DFT(a, d)$: Preslikaj vektor $a = (a_0, \ldots, a_n)$ z DFT reda $d = n + 1$.

Algoritem FFT za $DFT(\mathbf{a}, d)$ bomo razvili v dveh delih: prvi del bo *razdelil* $DFT(\mathbf{a}, d)$ na dva podproblema, drugi pa *sestavil* končno rešitev iz delnih rešitev.

I. Razdelitev problema na dva podproblema

Problem $DFT(a, d)$ razdelimo na dva podproblema v štirih korakih:

1. V $p(x)$ so koeficienti s sodimi indeksi in lihimi indeksi. „Sodi" koeficienti so $a_0, a_2, a_4, \ldots, a_{n-1}$, „lihi" pa $a_1, a_3, a_5, \ldots, a_n$. Vsakih koeficientov je r. Iz „sodih" lahko definiramo nov polinom $p_S(x)$, iz „lihih" pa nov polinom $p_L(x)$:

$$p_S(x) \stackrel{\text{def}}{=} a_0 + a_2 x + a_4 x^2 + \cdots + a_{n-1} x^{r-1} \qquad a_S \stackrel{\text{def}}{=} (a_0, a_2, a_4, \ldots, a_{n-1})$$
$$p_L(x) \stackrel{\text{def}}{=} a_1 + a_3 x + a_5 x^2 + \cdots + a_n x^{r-1} \qquad a_L \stackrel{\text{def}}{=} (a_1, a_3, a_5, \ldots, a_n)$$

   Njuni koeficientni predstavitvi smo označili z a_S in a_L.

2. Polinom $p(x)$ lahko izrazimo z novima polinomoma takole (preveri):

$$p(x) = p_S(x^2) + x\, p_L(x^2).$$

3. To nam na pove, da lahko izračunamo vrednosti $p(x)$ v dveh korakih:

   a) Izračunamo $p_S(x^2)$ in $p_L(x^2)$ v d točkah $x^2 = (\omega^0)^2, (\omega^1)^2, \ldots, (\omega^n)^2$.
   b) S temi vrednostmi izračunamo $p(x)$ v d točkah $x = \omega^0, \omega^1, \ldots, \omega^n$.

[1] Za $q(x)$ je algoritem enak, le n in $(a_0, \ldots, a_n)$ moramo zamenjati z m in $(b_0, \ldots, b_m)$.

[2] Kadar to ne drži, prištejemo polinomu $p(x)$ navidezni člen $0x^{n+1}$. Koeficientna predstavitev $(a_0, \ldots, a_n, 0)$ „novega" $p(x)$ ima zdaj sodo število komponent.

4. Naš namen je bil razdeliti problem $DFT(a, d)$ v dva podproblema *iste vrste* kot $DFT(a, d)$. Toda računanji vrednosti $p_S(x^2)$ in $p_L(x^2)$ v koraku 3a nista videti problema iste vrste kot $DFT(a, d)$. Zakaj? Oba sprašujeta po vrednostih polinoma pri *kvadratu* argumenta, medtem ko $DFT(a, d)$ pri argumentu; poleg tega nobeden od podproblemov ne zahteva računanja na *manjši* množici argumentov kot $DFT(a, d)$. *Ali torej nismo uspeli razdeliti problema $DFT(a, d)$ v dva manjša istorodna podproblema?* Smo. V naslednjih štirih korakih pojasnimo, kakšen je pravilen pogled na oba podproblema:

   a) Velja naslednje: $(\omega^{k+r})^2 = \omega^{2k}$.
   Dokaz. $(\omega^{k+r})^2 = \omega^{2k}\omega^{2r} = \omega^{2k}\omega^d = \omega^{2k} \cdot 1 = \omega^{2k}$, ker je ω d-ti koren enote.□

   b) Zato sta med d točkami $x^2 = (\omega^0)^2, (\omega^1)^2, \ldots, (\omega^n)^2$ po dve enaki:

$$(\omega^0)^2, \ldots, \underline{(\omega^k)^2}, \ldots, (\omega^{r-1})^2, (\omega^r)^2, \ldots, \underline{(\omega^{k+r})^2}, \ldots, (\omega^n)^2.$$

   Enake točke so $(\omega^k)^2$ in $(\omega^{k+r})^2$, kjer je $k = 0, \ldots, r-1$. Ker je ω d-ti *primitivni* koren enote, so pari pri različnih vrednostih k različni. Sledi, da je med d točkami $x^2 = (\omega^0)^2, (\omega^1)^2, \ldots, (\omega^n)^2$ le $d/2 = r$ paroma različnih. To pa pomeni, da bo treba v koraku 3a izračunati $p_0(x^2)$ in $p_1(x^2)$ le v r točkah $x^2 = (\omega^0)^2, \ldots, (\omega^k)^2, \ldots, (\omega^{r-1})^2$. Ker lahko eksponenta nad ω zamenjamo, so to točke

$$(\omega^2)^0, \ldots, (\omega^2)^k, \ldots, (\omega^2)^{r-1}.$$

   c) Če je ω d-ti primitivni koren enote, je ω^2 r-ti primitivni koren enote.
   Dokaz. (koren) $\omega^2 = (e^{\frac{2\pi}{d}})^2 = e^{\frac{2\pi}{d/2}} = e^{\frac{2\pi}{r}}$. (primitiven) Prepuščam za vajo. □

   d) Pišimo $\psi = \omega^2$. Zaradi 4b in 4c lahko korak 3a zapišemo v novi obliki:

   Izračunamo $p_S(x)$ in $p_L(x)$ v r točkah $x = \psi^0, \psi^1, \ldots, \psi^{r-1}$.

   Zdaj vidimo, da sta računanji vrednosti $p_S(x)$ in $p_L(x)$ podproblema, ki sta iste vrste kot problem računanja vrednosti $p(x)$, le da se nanašata na druga polinoma in zahtevata dvakrat manj argumentov. Zato lahko oba poimenujemo z istim imenom kot osnovni problem, a drugimi parametri: $DFT(a_S, r)$ in $DFT(a_L, r)$.

Sklep: Korak 3a sestavljata dva podproblema, $DFT(a_S, r)$ in $DFT(a_L, r)$.

Zdaj, ko smo uspeli razdeliti osnovni problem $DFT(a, d)$ v dva podproblema $DFT(a_S, r)$ in $DFT(a_L, r)$ iste vrste kot osnovni problem, je opravljen prvi del razvoja algoritma FFT z metodo Deli in vladaj. Kot že vemo pa bo učinkovitost algoritma FFT odvisna tudi od učinkovitosti, s katero bo sestavljal rešitvi podproblemov $DFT(a_S, r)$ in $DFT(a_L, r)$ v rešitev osnovnega problema $DFT(a, d)$. Ta del algoritma FFT opisuje naslednji razdelek.

II. Sestavljanje delnih rešitev v končno

V koraku 3b moramo iz vrednosti $p_S(\cdot)$ in $p_L(\cdot)$, ki jih izračuna korak 4d, sestaviti vse vrednosti

$$\underbrace{p(\omega^0), p(\omega^1), \ldots, p(\omega^{r-1})}_{\text{prva polovica}}, \underbrace{p(\omega^r), p(\omega^{r+1}), \ldots, p(\omega^n)}_{\text{druga polovica}}.$$

V naslednjih dveh korakih jih sestavimo takole:

6. Računanje vrednosti v prvi polovici, tj. vrednosti $p(\omega^k)$, $k = 0, \ldots, r-1$ gre po enačbi iz koraka 2:

$$p(\omega^k) = p_S(\omega^{2k}) + \omega^k p_L(\omega^{2k}) = \underbrace{p_S(\psi^k)}_{A} + \omega^k \underbrace{p_L(\psi^k)}_{B}.$$

   Toda v trenutku, ko se bo računala vrednost $p(\omega^k)$, bosta A in B že na voljo (kot rešitvi dveh podproblemov), zato bo za izračun $p(\omega^k)$ zadiščalo le eno množenje in eno seštevanje. Za izračun vseh r vrednosti $p(\omega^k)$ iz prve polovice pa bo potrebnih r množenj in r seštevanj.

7. Računanje vrednosti v drugi polovici, tj. vrednosti $p(\omega^{r+k})$, $k = 0, \ldots, r-1$, nas preseneti, ker zahteva le odštevanja. Poglejmo zakaj:

$$\begin{aligned}
p(\omega^{r+k}) &= p_S(\omega^{2(r+k)}) + \omega^k p_L(\omega^{2(r+k)}) = && \text{/zaradi 4a/} \\
&= p_S(\omega^{2k}) \quad + \omega^{r+k} p_L(\omega^{2k}) && = \quad \text{/zaradi 4c/} \\
&= p_S(\psi^k) \quad + \omega^{r+k} p_L(\psi^k) && = \quad \text{/ker } \omega^{r+k} = -\omega^k. \text{ Dokaži./} \\
&= p_S(\psi^k) \quad - \underbrace{\omega^k p_L(\psi^k)}_{C}.
\end{aligned}$$

   Ko se bo računala vrednost $p(\omega^{r+k})$, bo produkt C že izračunan, saj se je izračunal pri koraku 6. Zato bo izračun $p(\omega^{r+k})$ zahteval le eno odštevanje. Izračun vseh r vrednosti $p(\omega^{r+k})$ iz druge polovice pa bo zahteval r odštevanj.

Sklep: Korak 3b zahteva $r = \frac{d}{2}$ množenj in $2r = d$ aditivnih operacij.

Algoritem FFT v psevdokodi

Zdaj lahko zapišemo osnovno strukturo algoritma FFT.

```
procedure FFT(a,d);
begin
  if d=1 then return a else
    begin
      Pripravi \omega;
      Razdeli(a,a_S,a_L);
      p_S := FFT(a_S,d/2);
      p_L := FFT(a_L,d/2);
      p := Sestavi(p_S,p_L);
    end
end.
```

Časovna zahtevnost algoritma FFT

Naj bo $T(d)$ časovna zahtevnost algoritma `FFT(a,d)`. Tedaj je časovna zahtevnost vsakega od rekurzivnih klicev `FFT(a_S,d/2)` in `FFT(a_L,d/2)` enaka $T(\frac{d}{2})$. Koraki 1, 3a, 4d ter 6, 7 kažejo, da je skupna časovna zahtevnost procedur `Pripravi`, `Razdeli` in `Sestavi` enaka $\Theta(d)$. Zato za časovno zahtevnost algoritma `FFT(a,d)` velja enačba

$$T(d) = 2T\left(\frac{d}{2}\right) + \Theta(d).$$

Glavni izrek metode Deli in vladaj nam pove, da za rešitev $T(d)$ te enakosti velja

$$T(d) = \Theta(d \log d).$$

Opomba. Časovno zahtevnost algoritma FFT smo izrazili z d, številom komponent v koeficientni predstavitvi polinoma. Med d in stopnjo n polinoma je zveza $d = n+1$, zato tudi za časovno zahtevnost FFT, izraženo z n, velja $T(n) = \Theta(n \log n)$.

Zaključne opombe

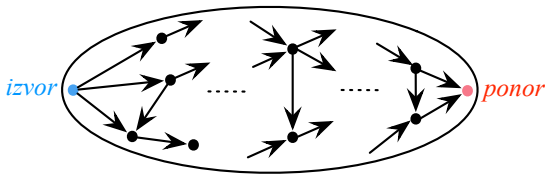
Razvili smo algoritem FFT, ki v času $\Theta(n \log n)$ pretvori koeficientno predstavitev polinoma stopnje n v njegovo vrednostno predstavitev na točkah $\omega^0, \ldots, \omega^n$, kjer je ω $(n+1)$-vi primitivni koren enote. Zdaj lahko FFT uporabimo pri množenju polinomov $p(x)$ in $q(x)$ tako, da njuni k.p. pretvorimo v v.p. na omenjenih točkah. Ko dodamo manjkajoče vrednosti vsaki od v.p., množenje istoležnih komponent da v.p. produkta $p(x)q(x)$. Slednjo pretvorimo v k.p. z inverzno DFT oz. trivialno spremenjenim FFT. Sklep: polinoma stopnje n lahko zmnožimo v času $\Theta(n \log n)$.

6 Največji pretok

To je *optimizacijski problem*, ker sprašuje po rešitvi, ki je najboljša glede na dani kriterij. Problem bomo rešili s posebnim algoritmom, ki je pomemben, ker na njem temelji kar nekaj novejših, izboljšanih algoritmov. Najprej pa nekaj za motivacijo.

Motivacija

Dano je *omrežje*, ki ga sestavljajo vozlišča in povezave med njimi. V omrežju sta dve odlikovani vozlišči, *izvor* in *ponor*. Iz izvora doteka v omrežje neka *dobrina*, ki se brez izgub razporedi in pretaka po povezavah omrežja, in ki vsa odteka skozi ponor. Če se dotok dobrine (količina dobrine v sekundi) poveča, se spremeni njen pretok po povezavah in končno poveča tudi njen odtok v ponor – a le do neke meje, saj ima vsaka povezava svojo *kapaciteto*, največji pretok, ki ga zmore. Zato obstaja neki *največji pretok* dobrine od izvora do ponora takega omrežja. Problem, ki nas bo zanimal, je za dano omrežje izračunati največji pretok.



Slika 6.1 Omrežje z izvorom in ponorom.

Primerov iz vsakdanjega življenja ni malo: vodovodno, cestno in podatkovno omrežje so trije primeri. Čeprav našteta omrežja v splošnem nimajo le enega izvora in le enega ponora, to lahko enostavno popravimo. Dodati moramo *navidezni izvor*, povezan z dejanskimi izvori, in *navidezni ponor*, povezan z dejanskimi ponori, ter razglasiti kapacitete dodanih *navideznih povezav* za neomejene. Po tem popravku je največji pretok iz dejanskih izvorov do dejanskih ponorov enak največjemu pretoku iz navideznega izvora do navideznega ponora v popravljenem omrežju.

Zdaj pa definirajmo problem natančneje.

Definicija problema

Omrežje je označen usmerjen graf $G(V, A, c)$. $V = \{1, 2, \ldots, n\}$ množica vozlišč, $A \subseteq V \times V$ je množica usmerjenih povezav med vozlišči in $c : A \to \mathbb{R}_0^+$ funkcija, ki vsaki povezavi $(i, j) \in A$ priredi njeno kapaciteto $c_{i,j} \geqslant 0$. Vozlišče 1 je izvor, vozlišče n pa ponor omrežja. Iz izvora priteka dobrina s hitrostjo v $[\frac{\text{enota dobrine}}{\text{sek}}]$, ki se razporedi po povezavah omrežja in odteka v ponor. Če označimo z $v_{i,j}$ pretok dobrine po povezavi (i, j), mora veljati

(1) $0 \leqslant v_{i,j} \leqslant c_{i,j}$ ... pretok čez povezavo je med 0 in kapaciteto povezave;

(2) $\sum_i v_{i,k} - \sum_j v_{k,j} = \begin{cases} -v & \text{če } k = 1; \quad // \text{ v 1 doteka 0 in odteka } v \ [\frac{\text{enota dobrine}}{\text{sek}}]; \\ 0 & \text{če } k \neq 1, n; \quad // \text{ kolikor v } k \text{ doteka, iz } k \text{ tudi odteka;} \\ v & \text{če } k = n; \quad // \text{ v } n \text{ doteka } v \text{ in odteka 0 } [\frac{\text{enota dobrine}}{\text{sek}}]. \end{cases}$

Količina v je *trenutni pretok* skozi omrežje, množica $\{v_{i,j}\}$ pa *razporeditev* trenutnega pretoka v po povezavah omrežja.

Velikost $v > 0$ dotoka dobrine v omrežje lahko zmanjšamo na velikost $v' < v$. Zmanjšani dotok se bo samodejno prerazporedil po povezavah, tako da bo nova razporeditev $\{v'_{i,j}\}$ izpolnjevala zgornji zahtevi (1) in (2). *Kaj pa obrnjeno?* Ali lahko najdemo razporeditev $\{v''_{i,j}\}$, ki bo izpolnila (1) in (2), in bo $v'' > v$? Torej, ali lahko povečamo trenutni pretok v skozi omrežje G? Opazimo, da pretok ne more biti večji od $\sum_j c_{1,j}$, zato je to neka zgornja meja za velikost pretoka skozi G. To pa pomeni, da obstaja tudi *najmajša zgornja meja* za velikost pretoka skozi G.

Problem: Za omrežje $G(V, A, c)$ izračunaj najmanjšo zgornjo mejo v^* velikosti pretoka skozi omrežje in razporeditev $\{v^*_{i,j}\}$ pretoka v^* po povezavah omrežja.

Naivni algoritem

Najprej se lotimo reševanja problema s pojmom *prereza* grafa, ki ga je koristno poznati. Izkazalo se bo, da nas to vodi do algoritma z eksponentno časovno zahtevnostjo, kar je preveč. Zato lahko bralec ta razdelek preleti ali preskoči.

Definirajmo pojme, ki jih bomo rabili. Paru (S, T) rečemo $(1, n)$-*prerez* omrežja $G(V, A, c)$, če je $S \cup T = V$ in $S \cap T = \emptyset$ ter $1 \in S$ in $n \in T$. Torej množico V *razdelimo* v disjunktni množici S in T tako, da je izvor v S, ponor pa v T. *Kapaciteta* $c(S, T)$ tega prereza je vsota kapacitet vseh povezav, ki vodijo iz S v T,

$$c(S, T) := \sum_{i \in S, j \in T} c_{i,j}.$$

Intuitivno: iz S v T se lahko pretaka *kvečjemu* $c(S, T)$ enot dobrine na sekundo. To pomeni, da lahko izvor pošilja dobrino v omrežje s hitrostjo v, kjer je $v \leqslant c(S, T)$.

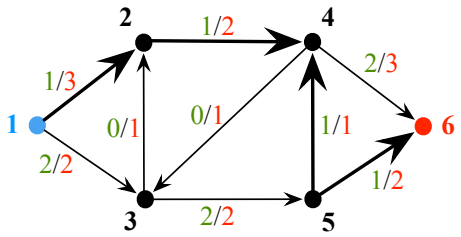
V splošnem ima G več $(1, n)$-prerezov, zato bo tisti, ki ima najmanjšo kapaciteto, določal največji pretok v^* iz izvora v ponor. Ugotovili smo, kako dobimo rešitev v^*:

$$v^* = \min_{(S,T)} c(S, T).$$

Kakšna je *časovna zahtevnost* računanja v^* iz zgornjega izraza? Izračunati je treba kapacitete vseh $(1, n)$-prerezov grafa G in izbrati najmanjšo med njimi. Vseh $(1, n)$-prerezov (S, T) grafa G je toliko kot podmnožic $S \subseteq V$, ki vsebujejo 1 ne pa n. Teh podmnožic je 2^{n-2}. Zato je časovna zahtevnost računanja v^* reda $\Theta(2^n)$.

Ford-Fulkersonov algoritem

Poskus, da bi rešili problem s pomočjo prerezov grafa G, se ni končal uspešno. Problema se zdaj lotimo na način, ki sta ga odkrila Ford in Fulkerson. Bistvo njune zamisli opišimo najprej na primeru omrežja na sliki 6.2.



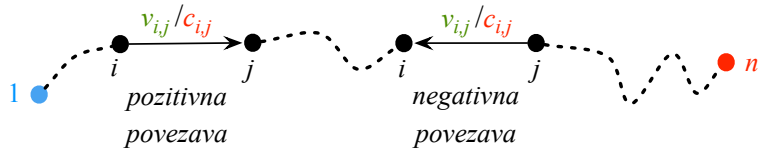
Slika 6.2 Primer omrežja. Vsaka povezava (i, j) je označena s parom $v_{i,j}/c_{i,j}$. Odebeljene povezave sestavljajo izbrano temeljno pot iz 1 v 6.

Izberimo *temeljno pot*[1] P, na primer $P \equiv 1 \overset{1/3}{\to} 2 \overset{1/2}{\to} 4 \overset{1/1}{\leftarrow} 5 \overset{1/2}{\to} 6$, in poskušajmo povečati pretok čez P, pri tem pa pozabimo na vsa vozlišča in povezave izven P. Povečajmo pretok $v_{1,2} = 1$ na 2. (To se da, saj je $c_{1,2} = 3$.) Ker zdaj v 2 doteka za 1 večji tok, bi moral iz 2 tudi odtekati za 1 večji tok, da bi zahteva (2) na str. 28 ostala izpolnjena. Zato moramo povečati $v_{2,4} = 1$ na 2. (To se da, ker je $c_{2,4} = 2$.) Zdaj v 4 doteka za 1 večji tok, zato bi moral iz 4 proti 5 odtekati za 1 večji tok. Toda trenutni tok med 4 in 5 je usmerjen proti 4. Zato zahtevo (2) za vozlišče 4 izpolnimo tako, da pretok $v_{5,4} = 1$ *zmanjšamo* za 1 na 0. (To smemo, saj je zaradi (1) na str. 28 pretok lahko tudi 0.) Ker zdaj iz 5 proti 4 teče za 1 manjši tok, moramo (zaradi zahteve (2)) za 1 povečati tok iz 5 proti 6, tj. povečati $v_{5,6} = 1$ na 2. (Tudi to se da, saj je $c_{5,6} = 2$.) Ker je 6 ponor, nam je uspelo povečati pretok čez P za 1. Z novimi pretoki v povezavah je $P \equiv 1 \overset{2/3}{\to} 2 \overset{2/2}{\to} 4 \overset{0/1}{\leftarrow} 5 \overset{2/2}{\to} 6$.

[1] *Temeljna pot* v $G(V, A, c)$ je zaporedje vozlišč $i_1, i_2, \ldots, i_k$, kjer se vsak i_j pojavi le enkrat, $i_1 = 1, i_k = n$ ter $(i_j, i_{j+1}) \in A$ ali $(i_{j+1}, i_j) \in A$. Po domače: temeljna pot je „pot" iz izvora v ponor, ki nima ciklov in na kateri zanemarimo smeri povezav.

Povzemimo: povečanje pretoka po prvi povezavi izbrane temeljne poti P je zahtevalo spremembe pretokov na vseh naslednjih povezavah na P. Vse te zahteve so izvirale iz pogoja (2) na str. 28, da mora iz vozlišča odtekati toliko dobrine kolikor je vanj doteka. Zato je bilo treba povečati odtekanje iz vozlišča (a ne preko kapacitete povezave) ali pa – če je bila povezava usmerjena nasprotno od ponora – zmanjšati dotekanje v vozlišče (a ne pod 0). Ker smo lahko vse spremembe uresničili, smo pretok čez P uspeli povečati.

V nadaljevanju to zamisel Forda in Fulkersona opišimo bolj natančno in splošno. Predpostavimo, da je v omrežju $G(V, A, c)$ vsako vozlišče na neki temeljni poti (tako je v resničnih omrežjih) in da so vsi $c_{i,j} \in \mathbb{N}$ (to prepreči patološke primere). Naj bo P temeljna pot v omrežju $G(V, A, c)$. Povezava (i, j) na P je *pozitivna*, če na P kaže v smeri od izvora proti ponoru; sicer je (i, j) na P *negativna*. (Slika 6.3). Množico vseh pozitivnih povezav na P označimo s P^+, množico vseh negativnih povezav na P pa s P^-. Povezava $(i, j) \in P^+$ je *zasičena*, če je $v_{i,j} = c_{i,j}$; povezava $(i, j) \in P^-$ pa je *zasičena*, če je $v_{i,j} = 0$. Temeljna pot P je *zasičena*, če vsebuje vsaj eno zasičeno povezavo.



Slika 6.3 Temeljna pot v omrežju. Vsaka povezava na njej je pozitivna ali negativna. Pozitivna povezava je zasičena, če je pretok čez njo dosegel njeno kapaciteto; negativna povezava pa je zasičena, če je pretok čez njo padel na 0. Cela pot je zasičena, če je na njej vsaj ena zasičena povezava.

Iz definicij sledi, da pretoka čez zasičeno P ne moremo povečati, saj je pretok v vsaj eni pozitivni povezavi na P dosegel kapaciteto povezave, ali pa je pretok v vsaj eni negativni povezavi na P padel na 0. Iz definicij tudi sledi, da P *ni zasičena*, če za *vsako* $(i, j) \in P^+$ velja $v_{i,j} < c_{i,j}$ *in* če za *vsako* $(i, j) \in P^-$ velja $v_{i,j} > 0$.

Zdaj se nam utrne zamisel algoritma: *po vrsti zasiti vse nezasičene temeljne poti.* V $G(V, A, c)$ je končno mnogo nezasičenih temeljnih poti. Ker so kapacitete povezav naravna števila, moramo pretok čez nezasičeno temeljno pot povečati za neko *naravno število*, da jo zasitimo. Sledi, da bo algoritem zasitil vse in se ustavil.[2] Zamisel pa takoj sproži tudi nekaj praktičnih vprašanj:

1. Ali je pretok čez omrežje $G(V, A, v)$ največji, če so vse temeljne poti zasičene?
2. Kako v $G(V, A, v)$ najdemo nezasičeno temeljno pot?
3. Kako nezasičeno temeljno pot zasitimo?
4. Za koliko se poveča pretok čez nezasičeno temeljno pot, če pot zasitimo?

[2]Ford in Fulkerson sta pokazala, da če so kapacitete poljubna realna števila, obstajajo „patološka" omrežja, kjer se algoritem ne ustavi, pač pa infinitezimalno povečuje pretok v nedogled.

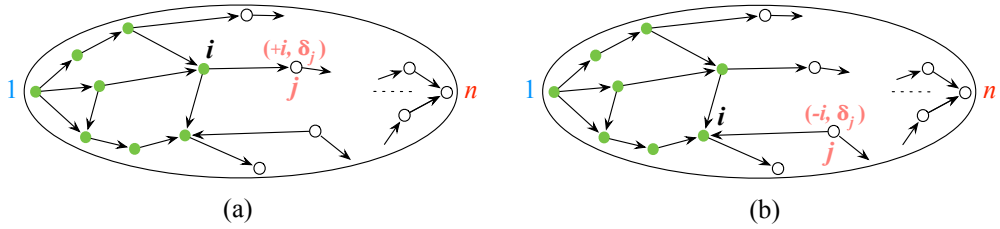
Začnimo z odgovorom na vprašanje 4. Na povečanje pretoka čez nezasičeno pot P vplivajo njene pozitivne in negativne povezave. Pretok v $(i,j) \in P^+$ se lahko poveča za največ $c_{i,j} - v_{i,j}$, zato je lahko povečanje pretoka čez P po zaslugi P^+ največ $\min_{(i,j)\in P^+}\{c_{i,j} - v_{i,j}\}$. Pretok v $(i,j) \in P^-$ pa se lahko zmanjša za kvečjemu $v_{i,j}$, zato je lahko povečanje pretoka čez P po zaslugi P^- kvečjemu $\min_{(i,j)\in P^-}\{v_{i,j}\}$. Sledi, da temeljno pot P zasitimo natanko tedaj, ko povečamo pretok čez njo za

$$\min\Big\{\min_{(i,j)\in P^+}\{c_{i,j} - v_{i,j}\}, \min_{(i,j)\in P^-}\{v_{i,j}\}\Big\}. \qquad (*)$$

V nadaljevanju odgovorimo hkrati na vprašanji 2 in 3. Zamislimo si naslednjo metodo, s katero bomo odkrili in zasitili neko nezasičeno temeljno pot v $G(V, A, c)$. Metoda naj začne v izvoru omrežja. Od tam naj postopno prodira v omrežje tako, da obišče čedalje več vozlišč, dokler ne doseže ponora. Pri tem naj vsako obiskano vozlišče ustrezno označi. Ključna zamisel „ustreznega" označevanja je naslednja: *oznake obiskanih vozlišč naj bodo take, da bo* (*po tem, ko bo dosežen ponor*) *možno* (*i*) *iz njih izslediti neko nezasičeno temeljno pot* P *in* (*ii*) P *zasititi.*

Izvedbo teh zamisli opisujejo naslednje točke:

- *Oznaka vozlišča.* Najprej opišimo oznake, ki naj jih nosijo označena vozlišča. Naj bo i označeno vozlišče, j pa neoznačen sosed. Kakšno oznako naj dobi j? Oznaka naj bo odvisna od smeri povezave med i in j: če je $(i,j) \in A$, naj j dobi oznako $(+i, \delta_j)$, če pa je $(j,i) \in A$, naj j dobi oznako $(-i, \delta_j)$. (Slika 6.4.)



Slika 6.4 Označena vozlišča (zelena) in neoznačena vozlišča (bela). Dodelitev oznake neoznačenemu vozlišču j, ki je sosed nepregledanega vozlišča i: (a) če gre povezava iz i v j; (b) če gre povezava iz j v i.

- *Pomen in namen oznake.* Kakšen pomen in namen ima oznaka vozlišča j? Prva komponeneta oznake vozlišča j bo omogočila, da izsledimo predhodnika vozlišča j (torej točko i) na nezasičeni temeljni poti po tem, ko bo pri prodiranju skozi G dosežen ponor in iz katerega se bo pri vračanju proti izvoru rekonstruirala ta pot. Druga komponenta oznake vozlišča j bo povedala, da se dá pretok čez pot $1 \to \cdots \to i \to j$ povečati za največ δ_j. Zato to komponento definiramo kot

$$\delta_j = \begin{cases} \min\{\delta_i, c_{i,j} - v_{i,j}\} & \text{če } (i,j) \in A; \\ \min\{\delta_i, v_{j,i}\} & \text{če } (j,i) \in A. \end{cases}$$

S tako definiranim δ_j lahko med prodiranjem skozi G sproti računamo vrednost izraza $(*)$ s prejšnje strani. (Zdaj je jasno, da ima izvor oznako $(-, \infty)$.)

- *Potek označevanja.* Med prodiranjem skozi omrežje bo v vsakem trenutku nekaj vozlišč že *označenih* in nekaj še *neoznačenih.* Med označenimi bodo nekatera imela vse svoje sosede tudi že označene, pri drugih pa bo vsaj en sosed še neoznačen. Prvim bomo rekli *pregledana*, drugim pa *nepregledana.* Na začetku bo izvor označen a nepregledan, ostala vozlišča pa neoznačena. Med prodiranjem skozi omrežje bomo ponavljali naslednje:
  (a) izberi nepregledano vozlišče i;
  (b) označi kakega neoznačenega soseda j (kot je opisano zgoraj);
  (c) razglasi j za označenega in nepregledanega;
  (d) razglasi i za pregledanega, če je bil j njegov zadnji neoznačeni sosed.

- *Izsleditev poti.* Ko je pri prodiranju skozi omrežje $G(V, A, c)$ dosežen in označen ponor n, je njegova oznaka $(+j, \delta_n)$ ali $(-j, \delta_n)$, kjer je $j \in V$. Če je $\delta_n > 0$, to pomeni, da *obstaja* nezasičena temeljna pot P, ki se jo da zasititi, če pretok čez njo povečamo za δ_n. Katera pa je ta temeljna pot P, čez katera vozlišča poteka? Iz oznake ponora n vidimo, da je predzadnje vozlišče na njej vozlišče j. To nam pove, da je $P \equiv 1, \cdots, j, n$. (Če je $(j, n) \in A$, je $P \equiv 1 \rightsquigarrow j \to n$, če pa je $(n, j) \in A$, je $P \equiv 1 \rightsquigarrow j \leftarrow n$.) Tudi vozlišče j ima oznako, ki je bodisi $(+i, \delta_n)$ ali $(-i, \delta_n)$, kjer je $i \in V$. Zato je $P \equiv 1, \cdots, i, j, n$. Tako enega za drugim izsledimo še vsa ostala vozlišča na poti P in tudi smeri povezav med njimi. Skratka, če je $\delta_n > 0$, lahko rekonstruiramo temeljno pot, ki jo lahko zasitimo s povečanjem pretoka za δ_n.

- *Zasičenje poti.* Kako zasitimo temeljno pot P, ki je bila odkrita v prejšji točki? Enostavno: vsaki pozitivni povezavi na poti P povečamo njen pretok za δ_n, vsaki negativni povezavi na P pa zmanjšamo njen pretok za δ_n. (To bi lahko sproti počeli že v prejšnji točki po vsakem odkritju prejšnjega vozlišča.)

- *Ponovitev.* Ko zasitimo rekonstruirano temeljno pot P, se pretok čez omrežje poveča za δ_n. Kljub temu lahko obstaja v omrežju še kaka nezasičena temeljna pot. Da jo odkrijemo, moramo ponoviti cel postopek označevanja. Seveda moramo pred tem izbrisati oznake vseh vozlišč razen izvora.

- *Konec.* Algoritem se konča, ko pri nekem pretoku v^* čez $G(V, A, c)$ v omrežju ni več nezasičenih temeljnih poti, tj. tedaj, ko je $\delta_n = 0$.

Časovna zahtevnost

Naj bo v^* največji pretok čez $G(V, A, c)$. Da Ford-Fulkersonov algoritem izračuna v^*, mora zasititi kvečjemu v^* temeljnih poti, saj vsako zasičenje temeljne poti poveča pretok čez $G(V, A, c)$ za vsaj 1. Vsako temeljno pot sestavlja kvečjemu $|A|$ povezav, zato izgradnja, izsleditev in zasičenje temeljne poti zahtevajo $O(|A|)$ časa. Sledi, da je časovna zahtevnost Ford-Fulkersonovega algoritma reda $O(v^*|A|)$.

Toda ni nam všeč, da je časovna zahtevnost računanja rezultata v^* odvisna prav od tega rezultata; poleg tega je lahko v^* zelo velik. Edmonds in Karp sta to slabost odpravila z naslednjim enostavnim dopolnilom v poteku označevanja: *vozlišča naj postanejo pregledana v istem vrstnem redu kot so postala označena*. Tako popravljeni algoritem ima časovno zahtevnost $O(|V|\cdot|A|^2)$.

Izboljšani algoritmi

Problem največjega pretoka je pomemben, zato je bil in je še predmet mnogih raziskav. Tule je seznam izboljšav Ford-Fulkersonovega algoritma in drugih algoritmov, skupaj z njihovimi časovnimi zahtevnostmi v odvisnosti od $n = |V|$ in $m = |A|$:

- $O(v^*m)$ ..... Ford, Fulkerson;
- $O(m^2n)$ ..... Edmonds, Karp;
- $O(mn^2)$ ..... Dinitz;
- $O(n^3)$ ..... Karzanov;
- $O(\sqrt{m}n^2)$ ..... Cherkassky;
- $O(n^3)$ ..... Malhotra et al.;
- $O(m^{2/3}n^{5/3})$ ..... Galil;
- $O(mn\log^2 n)$ ..... Galil, Naamad;
- $O(mn\log n)$ ..... Sleator, Tarjan;
- $O(mn\log(n^2/m))$ ... Goldberg, Tarjan;
- $O(mn\log_{\frac{m}{n\log n}} n)$ ... King,Rao,Tarjan;
- $O(mn)$ ..... Orlin

7 Metoda dinamičnega programiranja

Pri razvoju algoritmov pogosto uporabljamo tudi metodo, imenovano *Dinamično programiranje*. Zamisel metode je naslednja: Nalogo N dane velikosti reši pri trivialni velikosti, potem pa za vsako naslednjo velikost sestavi rešitve nalog te velikosti iz rešitev nalog manjših velikosti. Reševanje torej poteka „od spodaj navzgor", od manjših nalog proti večjim. Lahko se zgodi, da sestavljanje rešitev večjih nalog zahteva rešitve dosti manjših nalog, zato je te treba pomniti dalj časa.

Ko z dinamičnim programiranjem rešujemo optimizacijke probleme, se dostikrat opremo na t.i. *načelo optimalnosti*. Naj bo $D_1, D_2, \ldots, D_m$ zaporednje odločitev, ki jih sprejmemo med računanjem rešitve naloge N. Pravimo, da je to zaporedje *optimalno*, če nas privede do optimalne rešitve naloge N. *Načelo optimalnosti* pa pravi takole: *Vsako podzaporedje optimalnega zaporedja odločitev je tudi optimalno.*

Primer. Če najkrajšo (tj. optimalno) pot $A \rightsquigarrow B$ iz kraja A v kraj B razdelimo na dve etapi, $A \rightsquigarrow C$ in $C \rightsquigarrow B$, mora biti vsaka od etap tudi najkrajša (optimalna) – prva najkrajša iz A v C, druga pa najkrajša iz C v B. (Dokaz. Če to ne bi bilo res, bi bila najkrajša neka druga pot $A \leftrightarrow C$. Toda tedaj $A \rightsquigarrow B$ ne bi bila najkrajša, kar bi bilo v nasprotju z našo predpostavko!)

Primer. Načelo optimalnosti smo v resnici že srečali na strani 39 pri problemu NAHRBTNIK, ki smo ga tudi rešili z metodo dinamičnega programiranja.

8 Nahrbtnik

Tudi to je *optimizacijski problem*, ker sprašuje po najboljši rešitvi glede na dani kriterij. Problem je NP-težek, algoritem zanj pa bomo razvili z metodo dinamičnega programiranja. Najprej pa nekaj za motivacijo.

Motivacija

Neko nedeljo ob polni luni se Jaka kdo ve zakaj znajde v trgovini, oprtan s praznim, scefranim nahrbtnikom. Jaka je bister dečko, vé, da bi nahrbtnik zdržal težo 31kg, kaj več pa ne: dno bi počilo in vse bi se vsulo na tla za njegovo da ne rečem kaj. Jaki se zasvetijo oči, ko vidi vse pisano reči, ki bi se dale zbasati v nahrbtnik. „Vsaka reč ima svojo vrednost", si pravi. A tudi težo! „Ta moj nahrbtnik pa zdrži le piškavih 31kg." Jaka, kot rečeno odprte glave, razmišlja, kaj naj vtakne v nahrbtnik in česa ne, da se bo nabral najvrednejši plen, ki se ne bo vsul na tla že v trgovini.

Definicija problema

Definirajmo problem, s katerim se je soočil Jaka.

Problem: Dana je množica $R = \{1, 2, \ldots, n\}$ z elementi, ki predstavljajo neke *reči*, ter funkciji $v : R \to \mathbb{N}$ in $t : R \to \mathbb{N}$, ki vsaki reči i priredita *vrednost* $v(i)$ in *težo* $t(i)$. Dano je tudi število $b \in \mathbb{N}$, ki predstavlja *nosilnost nahrbtnika*. Naloga je naslednja: Poišči podmnožico $P \subseteq R$, imenovano *plen*, za katero bo veljalo

$$\sum_{i \in P} t(i) \leqslant b \quad (\textit{plen ni pretežek})$$

in ki bo maksimirala vsoto

$$\sum_{i \in P} v(i). \quad (\textit{plen je najvrednejši})$$

Ta problem se imenuje *Problem nahrbtnika* (krajše NAHRBTNIK) in je NP-težek. Zato verjetno ne bi uspeli zasnovati algoritma, ki bi problem rešil v *polinomsko* omejenem času glede na velikost primerka problema. Nič pa nam ne brani iskati algoritma, ki bo vedno vrnil rešitev ne glede na potreben čas.

Zamisel algoritma

Naj bo $V = \sum_{i \in R} v(i)$ skupna vrednost vseh reči v množici R. Izberimo poljuben $v \in \{0, 1, \ldots, V\}$. Izbrani v je vrednost, ki je smiselna za pomnožice $P \subseteq R$. (Števila, ki so manjša od 0 ali večja od V ali necela, ne morejo biti vrednosti kake $P \subseteq R$, saj so vrednosti reči v P naravna števila.)

Naj bo $i \in R$ poljubna reč. Definirajmo R_i kot množico prvih i reči množice R, tj. $R_i = \{1, \ldots, i\}$.

Množica R_i ima seveda svoje podmnožice, in vsaka od njih ima svojo vrednost in težo. Osredotočimo se le na tiste podmnožice množice R_i, ki so težke kvečjemu b. Zgodi se, da med njimi ni nobene, ki bi bila vredna ravno v (ki je bil izbran zgoraj). Seveda pa se tudi zgodi, da je med njimi *vsaj ena*, ki je vredna ravno v – in tedaj je (vsaj) ena med slednjimi najlažja. To podmnožico označimo z $N_i(v)$. Strogo pa jo definiramo takole:

$$N_i(v) := \begin{cases} \textit{najlažja med podmnožicami množice } R_i, \\ \textit{ki so vredne } v \textit{ in težke kvečjemu } b & \text{če taka obstaja;} \\ \uparrow \ (\textit{nedefinirano}) & \text{če take ni.} \end{cases}$$

Kako naj uporabimo to nenavadno definirano množico pri reševanju problema nahrbtnika? Ideja je tale: po vrsti računaj množice $N_n(V), N_n(V-1), N_n(V-2), \ldots$ in končaj pri prvi, ki je definirana.[1] Če je to množica $N_n(V-m)$, kjer je $0 \leqslant m \leqslant V$, velja $N_n(V)\uparrow, N_n(V-1)\uparrow, N_n(V-2)\uparrow, \ldots, N_n(V-m+1)\uparrow$ in $N_n(V-m)\downarrow$. Intuitivno: algoritem zmanjšuje želeno vrednosti plena, dokler ne najde plena, ki se da odnesti. Očitno je vrednost $V - m$ maksimalna vrednost v^* plena, ki se ga da odnesti, množica $N_n(V-m)$ pa iskana množica P.

Naslednje vprašanje, ki se nam samo postavi je, kako računati množico $N_n(\cdot)$ pri nekem poljubnem argumentu. Smiselno bi bilo, da poskušamo izraziti $N_i(\cdot)$ z eno ali več „manjšimi" množicami enake vrste, tj. množicami $N_{i-1}(\cdot)$. Tako računanje „od zgoraj navzdol" bi nas vodilo k rekurzivnemu algoritmu vrste deli in vladaj za računanje množic $N_n(\cdot)$.

Računanja množic $N_n(\cdot)$ pa bi se lahko lotili tudi od „spodaj navzgor", se pravi, da bi iz že izračunanih množic $N_{i-1}(\cdot)$ računali množice $N_i(\cdot)$. Takemu pristopu pravimo *metoda dinamičnega programiranja.* Uporabili jo bomo v naslednjem razdelku.

[1] Definirano in nedefinirano pogosto označujemo z $\downarrow$ in $\uparrow$. Če puščico pritaknemu matematičnemu objektu, nastane predikat; na primer, $x\downarrow$ pomeni „x je definiran", $x\uparrow$ pa „x je nedefiniran".

Algoritem z dinamičnim programiranjem

Naš cilj v tem razdelku je izraziti množico $N_i(v)$ z eno ali več množicami $N_{i-1}(\cdot)$. Če nam bo uspelo, bomo lahko izrazili težo $T_i(v)$ množice $N_i(v)$ s težami $T_{i-1}(\cdot)$.

Dela se bomo lotili na induktiven način: najprej bomo raziskali, kakšne so množice $N_1(\cdot)$ in njihove teže $T_1(\cdot)$, potem pa bomo razmišljali, kakšna bi utegnila biti zveza med $N_i(v)$ in množicami $N_{i-1}(\cdot)$. Če bomo to zvezo našli, bi nas morala voditi tudi do zveze med $T_i(v)$ in $T_{i-1}(\cdot)$. Da bo manj oklepajev, pišimo v_i namesto $v(i)$ in t_i namesto t(i). Torej začnimo:

$\boxed{i = 1}$

Opravka imamo z množico $R_1 = \{1\}$. Njeni podmnožici sta dve: $\emptyset$ in $\{1\}$. Vrednost prve je 0, druge pa v_1. Torej so množice $N_1(\cdot)$ naslednje:

$N_1(0) = \emptyset$ ker je $\emptyset$ edina podmnožica R_1, vredna 0;
$N_1(v_1) = \{1\}$ ker je $\{1\}$ edina podmnožica R_1, vredna v_1;
za $v \neq 0, v_1$ je $N_1(v)\uparrow$ ker R_1 nima podmnožic, vrednih v.

Njihove teže so

$T_1(0) = 0$;
$T_1(v_1) = t_1$;
za $v \neq 0, v_1$ je $T_1(v)\uparrow$.

$\boxed{i \geqslant 2}$

Zdaj je $R_1 = \{1, \ldots, i\}$. Ko bo $N_i(v)$ izračunana, bo bodisi vsebovala element i ali pa ga ne bo vsebovala. Poglejmo vsako od možnosti podrobneje.

$\boxed{i \in N_i(v)}$

V tem primeru bo $N_i(v)$ sestavljena iz $\{i\}$ in (načelo optimalnosti!) najlažje podmnožice množice R_{i-1}, ki je vredna $v - v_i$; torej bo

$$N_i(v) = \{i\} \cup N_{i-1}(v - v_i). \qquad (*)$$

Teža te množice bo

$$T_i(v) = t_i + T_{i-1}(v - v_i).$$

To bo veljalo, če so bili $v - v_i \geqslant 0$, $N_{i-1}(v - v_i)\downarrow$ in $t_i + T_{i-1}(v - v_i) \leqslant b$.

$\boxed{i \notin N_i(v)}$

V tem primeru bo očitno

$$\text{in} \quad \begin{aligned} N_i(v) &= N_{i-1}(v) \\ T_i(v) &= T_{i-1}(v). \end{aligned} \qquad (**)$$

Toda za $N_i(v)$ je morala biti izbrana lažja od alternativnih $N_i(v)$ v $(*)$ in $(**)$, sicer končni $N_i(v)$ ne bi bil skladen s svojo definicijo. Zato je $N_i(v)$ težka

$$T_i(v) = \min\bigl\{t_i + T_{i-1}(v - v_i),\ T_{i-1}(v)\bigr\}.$$

Zdaj lahko zapišemo algoritem za reševanje problema NAHRBTNIK.

```
procedure Nahrbtnik(R,t,v,b) return P;
begin
  Izracunaj V;
  for v := 0 to V do
    N_1(v) := nedefinirano; T_1(v) := nedefinirano;
  endfor;
  N_1(0) := praznamnozica; T_(0) := 0;
  N_1(v_1) := {1}; T_1(v_1) := t_1;
  for i:= 2 to n do
    for v := 0 to V do
      if v-v_i >= 0
         and N_{i-1}(v-v_i) definirana
         and t_i + T_{i-1}(v-v_i) <= b
         and t_i + T_{i-1}(v-v_i) <= T_{i-1}(v)
      then
        begin
          N_i(v) := {i} U N_{i-1}(v-v_i);
          T_i(v) := t_i U T_{i-1}(v-v_i)
        end
      else
        begin
          N_i(v) := N_{i-1}(v);
          T_i(v) := T_{i-1}(v)
        end
    endfor
  endfor;
  v* := najvecji v, pri katerem je N_n(v) definirana;
  P := N_n(v*)
end.
```

Časovna zahtevnost algoritma

Časovno zahtevnost algoritma narekuje dvojna zanka `for i...for v`. Ostali deli algoritma imajo časovno zahtevnost $O(1)$. Telo dvojne zanke zahteva $O(1)$ operacij, izvede pa se $(n-1)(V+1)$-krat. Zato je časovna zahtevnost celega algoritma $O(nV)$.

Časovna zahtevnost algoritma je polinomsko odvisna tako od n kot od V. Ali naš algoritem reši NP-težek problem NAHRBTNIK v polinomskem času? Ne. Časovna zahtevnost je definirana kot funckija *dolžine* vhodnih podatkov (tj. velikosti prostora zanje), ne pa njihove *velikosti* (magnitude). V izrazu $O(nV)$ je V velikost vhodnih podatkov. Če V izrazimo z njihova dolžino $d = \lceil \log V \rceil$, je časovna zahtevnost algoritma $O(n2^d)$, torej *eksponentno* odvisna od dolžine vhodnih podatkov.

9 Najcenejše poti iz izbranega izhodišča

Problem, s katerim se bomo ukvarjali v tem in naslednjem poglavju je tipičen *optimizacijski* problem, ki ga bomo reševali z metodo dinamičnega programiranja. Problem je pomemben, ker je uporaben v praksi, pa tudi zato, ker nanj lahko prevedemo veliko drugih optimizacijskih problemov (npr. NAHRBTNIK).

Definicija problema

Dan je utežen usmerjen graf $G(V, A, c)$, kjer je $V = \{1, 2, \ldots, n\}$ množica vozlišč, $A \subseteq V \times V$ množica usmerjenih povezav med vozlišči in $c : V \times V \to \mathbb{R} \cup \{\infty\}$ funkcija, ki vsakemu paru $(i, j) \in V \times V$ priredi njegovo *ceno* $c_{i,j}$. Za vsak $i \in V$ je $c_{i,i} = 0$, in če $(i, j) \notin A$, je $c_{i,j} = \infty$. Usmerjena pot iz vozlišča $i_z \in V$ v $i_k \in V$ je zaporedje vozlišč $i_1, i_2, \ldots, i_\ell$, $\ell \geqslant 2$, kjer je $i_1 = i_z$ in $i_\ell = i_k$ ter $(i_j, i_{j+1}) \in A$ za $j = 1, \ldots \ell - 1$. Cena u_{i_z,i_k} te poti je vsota cen na njenih povezavah,

$$u_{i_z,i_k} = \sum_{j=1}^{j=\ell-1} c_{i_j,i_{j+1}}.$$

Cikel je usmerjena pot iz i_z v i_k, ki se konča v začetnem vozlišču. Rekli bomo, da je cikel *negativen*, če je njegova cena negativno število. Vozlišču 1 rečemo *izhodišče.*

Problem: Za vsako vozlišče $i \in V$ poišči najcenejšo pot iz 1 v i in njeno ceno $u_{1,i}$.

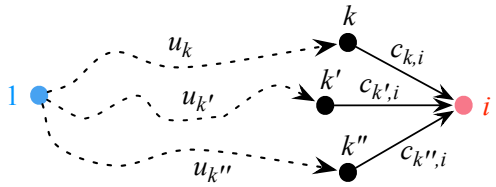
Včasih ta problem ni rešljiv. To se zgodi, če iz izhodišča do nekega vozlišča ni usmerjene poti (zaradi nepovezanosti grafa ali pa zaradi smeri povezav.) Drugi razlog je obstoj negativnih ciklov. Naj bo $i_z \rightsquigarrow i_k$ ($i_k = i_z$) negativen cikel v G in $c_{i_z,i_k} \in \mathbb{R}^-$ njegova cena. Kakšna je najcenejša pot iz 1 v vozlišče i_z? Denimo, da je to pot $1 = i_1, i_2, \ldots, i_z$ s ceno u_{1,i_z}. Če bi to pot podaljšali z negativnim ciklom $i_z \rightsquigarrow i_k (= i_z)$, bi bila cena podaljšane poti $u_{1,i_z} + c_{i_z,i_k} < u_{1,i_z}$, saj je $c_{i_z,i_k} < 0$. Zato $1 = i_1, i_2, \ldots, i_z$ ne bi bila najcenejša pot iz 1 v i_z, čeprav smo predpostavili, da je! Tudi če bi si premislili in rekli, da je podaljšana pot najcenejša, bi tudi njej lahko dodali še en obhod negativnega cikla. Ker bi lahko cikel obhodili poljubno mnogokrat, bi lahko ceno poti iz 1 v i_z v nedogled zmanjševali. Sklenemo lahko, da ne obstaja najcenejša pot iz 1 v i_z.

Zato bomo odslej predpostavljali, da (1) v G *obstaja usmerjena pot do vsakega vozlišča* in (2) G *nima negativnih ciklov.*

Bellmanove enačbe

Pišimo u_i namesto $u_{1,i}$. Kako izračunati u_i za dani i? Če je $i = 1$, je $u_1 = c_{1,1} = 0$. Če je $i \neq 1$, ima najcenejša pot iz 1 v i obliko $1 \rightsquigarrow k \to i$, kjer je $k \neq i$. Po načelu optimalnosti mora biti $1 \rightsquigarrow k$ najcenejša pot iz 1 v k, zato je njena cena u_k. Če temu prištejemo še ceno povezave $k \to i$, dobimo $u_k + c_{k,i}$, kar je cena najcenejše poti iz 1 v i, *ki prečka i-jevega soseda k*. Toda i ima lahko še druge sosede $k', k'', \ldots$ in preko vsakega lahko obstaja najcenejša pot iz 1 v i. Najcenejša med temi potmi mora biti iskana najcenejša pot iz 1 v i (ne glede na to, katerega soseda prečka). Ugotovili smo, da je

$$u_i = \min_{\substack{k \\ (k,i)\in A}} \{u_k + c_{k,i}\}.$$



Slika 9.1 Najcenejša pot iz 1 v i prečka nekega soseda vozlišča i.

To mora veljati za vsak $i = 2, 3, \ldots, n$ posebej; povedano drugače, za $u_1, u_2, u_3, \ldots u_n$ velja naslednji sistem t.i. *Bellmanovih*[1] enačb (BE):

$$\begin{aligned}
u_1 &= 0 \\
u_2 &= \min_{\substack{k \\ (k,2)\in A}} \{u_k + c_{k,2}\} \\
u_3 &= \min_{\substack{k \\ (k,3)\in A}} \{u_k + c_{k,3}\} \\
&\vdots \\
u_i &= \min_{\substack{k \\ (k,i)\in A}} \{u_k + c_{k,i}\} \\
&\vdots \\
u_n &= \min_{\substack{k \\ (k,n)\in A}} \{u_k + c_{k,n}\}
\end{aligned}$$

oz. krajše

$$\boxed{u_i = \begin{cases} 0 & \text{če } i = 1; \\ \min\limits_{\substack{k \\ (k,i)\in A}} \{u_k + c_{k,i}\} & \text{če } i \geqslant 2. \end{cases}}$$

Bellmanove enačbe

[1] Richard E. Bellman, 1920–1984, ameriški matematik.

Ugotovili smo: Če so $u_1, u_2, \ldots, u_n$ cene najcenejših poti iz vozlišča 1 v vozlišča $1, 2, \ldots n$ grafa $G(V, A, c)$, potem so rešitev ustreznega sistema Bellmanovih enačb. Povedano drugače: Če so $u_1, u_2, \ldots, u_n$ rešitev *Problema najcenejših poti iz izhodišča* za dani graf $G(V, A, c)$, potem so tudi rešitev grafu pripadajočega sistema BE.

Kja pa obratno? Če so $u_1, u_2, \ldots, u_n$ rešitev grafu $G(V, A, c)$ pripadajočega sistema BE, ali so tudi rešitev *Problema najcenejših poti iz izhodišča* za $G(V, A, c)$? Odgovor je *da*. Seveda moramo to tudi dokazati.

Ideja dokaza. Ugotoviti moramo, kaj v grafu $G(V, A, c)$ pomenijo $u_1, u_2, \ldots, u_n$, ki so rešitev sistema BE. Do ugotovitve, da „u_i pomeni ceno najcenejše poti iz 1 v i v G" pridemo, ker nam na podlagi lastnosti $u_1, u_2, \ldots, u_n$ uspe (i) povezati vozlišča V grafa G v drevo T s korenom v 1 in povezavami $k \to i$, kjer je k tisti, ki minimizira desni del enačbe $u_i = \min_k\{u_k + c_{k,i}\}$; (ii) dokazati, da je T vpeto drevo v G; (iii) dokazati, da je u_i cena neke poti iz 1 v i v G; in (iv) dokazati, da je $u_1, u_2, \ldots, u_n$ edina rešitev sistema BE. □

Posledica obeh ugotovitev je pomembna:

Rešitev Problema najcenejših poti iz izhodišča v grafu $G(V, A, c)$ je natanko rešitev grafu pripadajočega sistema Bellmanovih enačb.

Problem *Najcenejše poti iz izhodišča* smo prevedli na problem *Reševanje sistema BE.* Zato bomo pozornost usmerili na vprašanje, kako rešiti sistem Bellmanovih enačb.

Reševanje sistema BE Bellmanovih enačb

Oglejmo si spet sistem Bellmanovih enačb

$$\begin{aligned}
u_1 &= 0 \\
u_2 &= \min_{\substack{k \\ (k,2)\in A}} \{u_k + c_{k,2}\} \\
u_3 &= \min_{\substack{k \\ (k,3)\in A}} \{u_k + c_{k,3}\} \\
&\vdots \\
u_i &= \min_{\substack{k \\ (k,i)\in A}} \{u_k + c_{k,i}\} \\
&\vdots \\
u_n &= \min_{\substack{k \\ (k,n)\in A}} \{u_k + c_{k,n}\}
\end{aligned}$$

Kako naj ga začnemo reševati? Denimo, da izberemo enačbo $u_i = \min_k\{u_k + c_{k,i}\}$. Da bi lahko izračunali u_i, moramo prej izračunati u_k za vse k, kjer obstaja $k \to i$. Toda pri vsakem od teh u_k se pojavi enaka zahteva po predhodnem izračunu nekaterih drugih u_j. Torej bi morali začeti s tisto enačbo $u_\ell = \min_k\{u_k + c_{k,\ell}\}$, kjer je $k = 1$ *edini* sosed s povezavo $k \to \ell$, saj je tedaj $u_\ell = \min_k\{u_k + c_{k,\ell}\} = u_1 + c_{1,\ell}$.

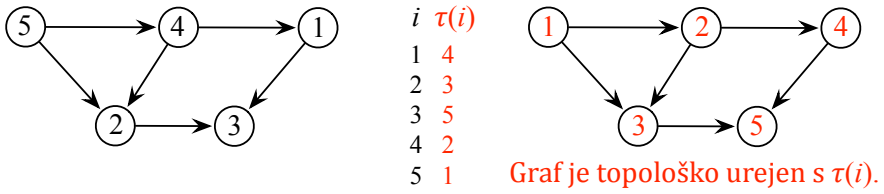
Zdaj, ko bi poznali u_1 in u_ℓ, bi poiskali enačbo, ki zahteva poznavanje le u_1 in u_ℓ. Tako bi nadaljevali z izbiranjem enačb in povečevanjem množice izračunanih cen.

Reševanja sistema pa se lahko lotimo tudi drugače. Ker nam splošna enačba $u_i = \min_{k,(k,i)\in A}\{u_k + c_{k,i}\}$ razkriva, da za izračun vrednosti u_i potrebujemo *nekatere že izračunane* u_k, bi bila za reševanje takega sistema primerna tudi metoda *dinamičnega programiranja.* V nadaljevanju bomo obravnavali posebne družine sistemov BE (oz. pripadajočih grafov $G(V, A, c)$), kjer je ta metoda še posebej primerna.

Prva taka družina sistemov Belmannovih enačb pripada *acikličnim* grafom $G(V, A, c)$. Preden si jo bomo ogledali, moramo na kratko zaviti z glavne poti in spoznati *topološko urejanje* grafov.

Topološko urejanje grafov

Naj bo $G(V, A)$ graf z množico vozlišč $V = \{1, 2, \ldots, n\}$. Vozlišča želimo preimenovati tako, da bo v preimenovanem grafu vsaka povezava tekla iz vozlišča z manjšim imenom v vozlišče z večjim imenom. Preimenovanje vozlišč naj opravi bijektivna funkcija $\tau : V \to V$, ki vsakemu $i \in V$ priredi novo ime $\tau(i) \in V$ tako, da bo veljalo $(i, j) \in A \Rightarrow \tau(i) < \tau(j)$. Če za graf $G(V, A)$ taka funkcija τ obstaja, rečemo, da τ *topološko ureja* $G(V, A)$ oz. da je $G(V, A)$ z njo topološko urejen (Slika 9.2).



Slika 9.2 Topološko urejen graf.

Graf ima lahko več različnih topoloških ureditev. Na primer, graf na Sliki 9.2 topološko ureja tudi funkcija $\tau = \binom{1,\ 2,\ 3,\ 4,\ 5}{3,\ 4,\ 5,\ 2,\ 1}$. Vednar pa obstajajo grafi, ki jih ni mogoče topološko urediti. Tak je, na primer, usmerjeni cikel $3 \to 1 \to 2 \to 3$. Vprašamo: *Kateri grafi se dajo topološko urediti in kateri ne?* Odgovor je tale izrek:

Izrek. *Graf G(V,A) se da topološko urediti natanko tedaj, ko je acikličen.*

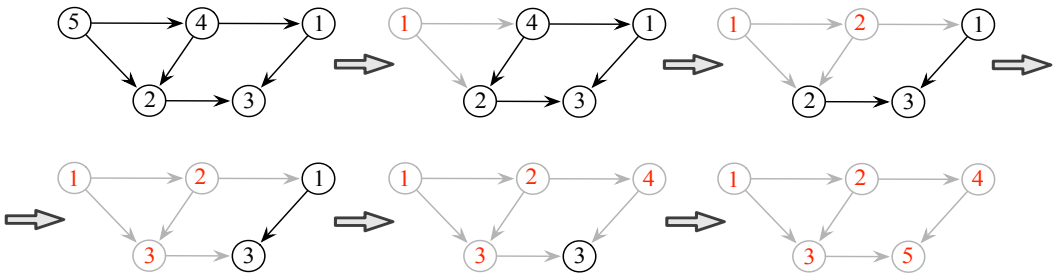
Dokaz. ($\Rightarrow$) Naj bo G topološko urejen. Če bi imel cikel, bi v ciklu obstajalo vozlišče i z lastnostjo $i < i$, kar ni možno. Zato je G acikličen. ($\Leftarrow$) Naj bo G acikličen. Z indukcijo po $|V|$ dokažimo, da se da topološko urediti. Pri $|V| = 1$ je to očitno. Predpostavimo, da trditev velja za vse aciklične grafe z $|V| = n$. Naj bo G poljuben acikličen graf z $n + 1$ vozlišči. Ker je G acikličen, ima vozlišče i z vhodno stopnjo 0. Vozlišče i preimenujmo v 1 in odstranimo skupaj z vsemi njegovimi povezavami iz G. Preostanek $G - i$ je graf z n vozlišči, ki se ga po predpostavki da topološko urediti (z novimi imeni $2, 3, \ldots, n$). □

Drugi del dokaza je konstruktiven, ker opiše algoritem za topološko ureditev poljubnega acikličnega grafa. Zapišimo algoritem v psevdokodi:

```
procedure Topoloska_Ureditev(G);
begin
  G':= G;                                          //G ohrani, G' krci
  s := 0;
  while Ima_vozlisce_z_vh.stopnjo_0(G') do
    i := Izberi_vozlisce_z_vh.stopnjo_0(G');
    s := s+1;
    tau(i) := s;                                   //doloci novo ime za i
    G':= G'-i                                //izloci i in njegove povezave
  endwhile;
  if Prazen_graf(G')
    then return(G_je_aciklicen; topolosko_urejen_s_\tau)
    else return(G_je_ciklicen)
end.
```

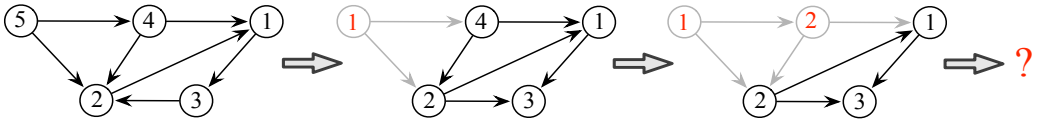
Časovna zahtevnost. Časovno zahtevnost algoritma narekuje zanka `while`. Vsaka izvedba njenega telesa je hitrejša od prejšnje, ker se število vozlišč, ki jih je treba pregledati in med njimi eno izbrati, vsakokrat zmanjša za 1. Ker k-ta izvedba zahteva $O(|V|-k)$ časa, vse zahtevajo $c(|V|+|V|-1+\ldots+2+1) = O(|V|^2)$ časa. Časovna zahtevnost algoritma za topološko ureditev grafa $G(V, A)$ je torej $O(|V|^2)$.

Primer. Za primer poglejmo, kako poteka topološko urejanje grafa s prejšnje slike.



Slika 9.3 Topološko urejanje grafa. Rdeča števila so nova imena vozlišč. Algoritem je v tretjem koraku med 1 in 2 izbral točko 2 in jo preimenoval v 3.

Primer. Poglejmo še, kako se konča poskus topološke ureditve cikličnega grafa.



Slika 9.4 Poskus, da bi topološko uredili *ciklični* graf G se konča, ko *neprazen* graf G' *nima* točke z vhodno stopnjo 0. To je znak, da je G cikličen graf.

Najcenejše poti iz izhodišča v acikličnem grafu

Vračamo se na problem najcenejših poti iz izhodišča grafa. Kadar je graf acikličen, mu lahko pridružimo ustrezen sistem Belmannovih enačb, ki je zelo prikladen za reševanje z metodo dinamičnega programiranja. To dosežemo tako, da dani graf topološko uredimo. Poglejmo podrobnosti.

Naj bo dan acikličen graf. Ko ga topološko uredimo, dobimo graf $G(V, A, c)$, ki ga odlikuje lastnost $(i, j) \in A \Rightarrow i < j$. Torej, če v $G(V, A, c)$ obstaja povezava iz i v j, potem je $i < j$. To pa vpliva tudi na Bellmanove enačbe na str. 43, v katerih pogoj $(k, i) \in A$ lahko nadomestimo s $k < i$, tako da se zdaj glasijo

$$\begin{aligned}
u_1 &= 0 \\
u_2 &= \min_{\substack{k \\ k<2}} \{u_k + c_{k,2}\} \\
u_3 &= \min_{\substack{k \\ k<3}} \{u_k + c_{k,3}\} \\
&\vdots \\
u_i &= \min_{\substack{k \\ k<i}} \{u_k + c_{k,i}\} \\
&\vdots \\
u_n &= \min_{\substack{k \\ k<n}} \{u_k + c_{k,n}\}
\end{aligned}$$

Računanje rešitve $u_1, u_2, \ldots, u_n$ takega sistema BE poteka po naraščajočem indeksu i: ko so izračunani $u_1, \ldots, u_{i-1}$, začnemo računati u_i. Torej

$$\begin{aligned}
u_1 &= 0 \\
u_2 &= u_1 + c_{1,2} \\
u_3 &= \min\{u_1 + c_{1,3},\ u_2 + c_{2,3}\} \\
u_4 &= \min\{u_1 + c_{1,4},\ u_2 + c_{2,4},\ u_3 + c_{3,4}\} \\
u_5 &= \min\{u_1 + c_{1,5},\ u_2 + c_{2,5},\ u_3 + c_{3,5},\ u_4 + c_{4,5}\} \\
&\vdots \\
u_i &= \min\{u_1 + c_{1,i},\ u_2 + c_{2,i},\ u_3 + c_{3,i},\ \ldots, u_{i-1} + c_{i-1,i}\} \\
&\vdots \\
u_n &= \min\{u_1 + c_{1,n},\ u_2 + c_{2,n},\ u_3 + c_{3,n},\ u_4 + c_{4,n},\ \ldots, u_{n-1} + c_{n-1,n}\}
\end{aligned}$$

Časovna zahtevnost. Izračun u_i zahteva $i - 1$ odštevanj in $i - 2$ primerjanji. Izračun rešitve sistema zato zahteva $\sum_{i=2}^{n}(i-1) = n(n-1)/2 = O(n^2)$ odštevanj in $\sum_{i=3}^{n}(i-2) = (n-1)(n-2)/2 = O(n^2)$ primerjanj.

Sklep.[2] *Časovna zahtevnost računanja najcenejših poti iz izhodišča v acikličnem grafu $G(V, A, c)$ je $O(|V|^2)$.*

[2]Sklep velja neodvisno od tega, ali je $G(V, A, c)$ že topološko urejen ali ne, saj tudi topološka ureditev zahteva $O(|V|^2)$ časa.

Najcenejše poti iz izhodišča v grafu s pozitivnimi cenami (Dijkstra)

Naj bo $G(V, A, c)$ usmerjen graf, v katerem so cene vseh povezav *pozitivne*; torej $(i, j) \in A \Rightarrow c_{i,j} > 0$. Pri tej družini grafov se pri računanju cen $u_1, u_2, \ldots, u_n$ ne bomo naslonili na sistem BE pač pa opisali algoritem, ki ga je zasnoval Dijkstra.[3] Dijkstra je razmišljal nekako takole:

1. V vsakem trenutku računanja cen $u_1, u_2, \ldots, u_n$ je vsaka cena u_i bodisi *dokončna* bodisi *začasna*. Začasne cene se bodo v nadaljevanju računanja še spreminjale, dokončne pa ne. Zato je v vsakem trenutku $V = D \cup Z$, kjer je D množica vozlišč z dokončnimi, Z pa množica vozlišč z začasnimi cenami. Na začetku računanja je že znana in dokončna cena $u_1 = 0$, cene $u_2, u_3, \ldots, u_n$ pa bo treba še izračunati, zato so – ne glede na njihovo inicializacijo – vse še začasne. Seveda jih je smiselno inicializirati na $u_i := c_{1,i}\,(= \infty,\ \text{če}\ (1, i) \notin A)$. Torej se algoritem začne takole:

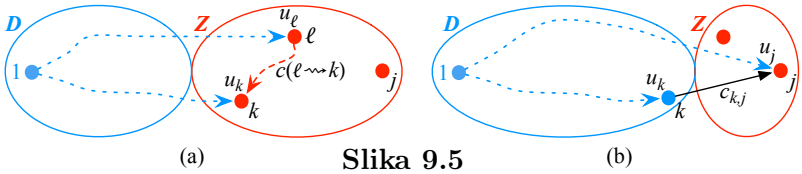
$$u_1 := 0; \quad \forall i > 1 : u_i := c_{1,i}; \quad D := \{1\}; \quad Z := \{2, 3, \ldots, n\};$$

2. V nadaljevanju želimo, da bi se množica D monotono večala, tako da bi vozlišča prestopala iz Z v D. Kako pa začasna cena nekega vozlišča postane dokončna? Dijkstra je opazil tole. Če je u_k *najmanjša začasna* cena, torej $u_k = \min_{j \in Z}\{u_j\}$, potem se u_k ne bo več zmanjšal. Zakaj? Denimo, da bi do vozlišča k vodila kaka cenejša pot, ki bi prečkala vsaj eno vozlišče $\ell \in Z$ (Slika 9.5a). Njena cena bi bila $u_\ell + c(\ell \rightsquigarrow k) < u_k$. Ker pa so cene vseh povezav pozitivne, bi morala biti cena $c(\ell \rightsquigarrow k) > 0$ in zato $u_\ell < u_k$. To pa bi bilo protislovno, sa je u_k po predpostavki najmanjša začasna cena. Ker je torej cena u_k dokončna, mora k prestopiti iz Z v D. Če zaradi prestopa postane $Z = \emptyset$, se algoritem konča, saj so vse cene $u_1, u_2, \ldots, u_n$ dokončne. Algoritem torej dopolnimo z ukazi

$$u_k := \min_{j \in Z}\{u_j\}; \quad D := D \cup \{k\}; \quad Z := Z - \{k\}; \quad \text{Če je } Z = \emptyset, \text{ končaj.}$$

3. Ko cena u_k postane dokončna, to lahko vpliva na začasne cene u_j vozlišč $j \in Z$. Zakaj? Začasne cene u_j, kjer $j \neq k$, so bile izračunane ne podlagi vozlišč v D, ko med njimi še ni bilo vozlišča k. Zdaj pa začasno najkrajša pot do $j \in Z$ lahko gre tudi čez k (Slika 9.5b). Zato algoritem še dopolnimo:

$$\forall j \in Z : u_j := \min\{u_j, u_k + c_{k,j}\}; \quad \text{Skoči na korak 2.}$$



Slika 9.5

[3] Edsger W. Dijkstra (izg. deikstra), 1930–2002, nizozemski računalnikar.

Po zgornjih korakih zapišimo Dijkstrov algoritem še v psevdokodi:

```
procedure Dijkstrov_Algoritem(G(V,A,c));
begin
  u_1 := 1;  for i := 2 to n do u_i := c_1,i;            // 1
  D := {1}; Z := {2,3,...,n};
  while NiPrazna(Z) then
    u_k := min_{j \in Z}{u_j};                           // 2
    D := D + {k}; Z := Z - {k};
    IF NiPrazna(Z) then
      forall j \in Z do u_j := min{u_j, u_k + c_k,j};    // 3
  endwhile
 end.
```

Časovna zahtevnost. Časovno zahtevnost algoritma narekuje zanka `while`. Njeno telo se izvede $n-1$-krat, saj v vsaki izvedbi Z zapusti eno vozlišče. Iskanje u_k zahteva največ $|Z|-1$ primerjanj, popravljanje preostalih začasnih cen pa $|Z|-1$ primerjanj in $|Z|-1$ seštevanj. Zahtevnost ostalih operacij v telesu zanke je $O(1)$. Izvedba telesa zanke torej zahteva $3(|Z|-1)+O(1)$ operacij. Ker je v i-ti izvedbi $|Z| = n-i$, zahteva i-ta izvedba telesa $3(n-i-1)+O(1)$ operacij. Torej je zahtevnost zanke $3\big((n-2)+(n-3)+(n-4)+\ldots 2+1\big)+(n-1)O(1)$, kar je reda $O(n^2)$.

Sklep. *Časovna zahtevnost računanja najcenejših poti iz izhodišča v grafu $G(V,A,c)$ s pozitivnimi cenami je $O(|V|^2)$.*

Najcenejše poti iz izhodišča v splošnem grafu (Bellman-Ford)

Spoznali smo, kako računamo najcenejše poti iz izhodišča v acikličnem grafu in v grafu s pozitivnimi cenami povezav. Kaj pa, če je graf cikličen in ima tudi povezave z negativnimi cenami? (Nima pa negativnih ciklov, kot smo razložili na str. 41.) V tem primeru topološka ureditev grafa in Dijkstrov algoritem nista več uporabna. Na srečo pa sta algoritem za te splošne grafe odkrila Bellman in Ford.[4]

Naj bo $G(V, A, c)$ utežen usmerjen graf, kjer je $V = \{1, 2, \ldots, n\}$ množica vozlišč, $A \subseteq V \times V$ množica usmerjenih povezav med vozlišči in $c : V \times V \to \mathbb{R} \cup \{\infty\}$ funkcija, ki vsakemu paru $(i, j) \in V \times V$ priredi njegovo ceno $c_{i,j}$. Zahtevamo tudi, da za vse $1 \leqslant i, j \leqslant n$ velja $c_{i,i} = 0$ in $(i, j) \notin A \Rightarrow c_{i,j} = \infty$.

Ključni premislek, ki sta ga naredila Bellman in Ford v razvoju svojega algoritma, je:

- Najcenejša pot iz 1 v i vsebuje kvečjemu $n - 1$ povezav (sicer bi se neko vozlišče ponovilo, dobljeni cikel bi bil pozitiven, pot čezenj pa ne najcenejša).
- Naj bo $u_i^{(p)} \equiv$ cena najcenejše poti iz 1 v i, ki vsebuje kvečjemu p povezav.
- Trivialna ekvivalenca: Za vsak X in p je $X \leqslant p \iff (X \leqslant p - 1) \vee (X = p)$.
- Ta ekvivalenca je podlaga za naslednjo Bellman-Fordovo ugotovitev:

  Najcenejša pot P iz 1 v i, ki vsebuje kvečjemu p povezav, vsebuje

  (a) *bodisi* kvečjemu $p - 1$ povezav;
  (b) *bodisi* natanko p povezav.
- Cena $u_i^{(p)}$ poti P je odvisna od alternative, ki velja za P; zato je

  (a) *bodisi* $u_i^{(p)} = u_i^{(p-1)}$;
  (b) *bodisi* $u_i^{(p)} = \min\limits_{\substack{k \\ (k,i)\in A}} \{u_k^{(p-1)} + c_{k,i}\}$.

  Ker je P splošen, ne moremo vedeti, katera alternativa velja, vemo pa, da mora biti njegova dejanska cena $u_i^{(p)}$ manjša izmed alternativnih cen; torej $u_i^{(p)} = \min\{u_i^{(p-1)}, \min\limits_{\substack{k \\ (k,i)\in A}} \{u_k^{(p-1)} + c_{k,i}\}\}$. To pa je *rekurzivna enačba* za $u_i^{(p)}$!
- *Začetne vrednosti* za rekruzivno enačbo so jasne: $u_1^{(1)} = 0$ in $u_i^{(1)} = c_{1,i}$ za $i > 1$.

S tem sta Bellman in Ford za splošni graf izpeljala naslednji sistem enačb BE:

$$u_i^{(p)} = \begin{cases} 0 & \text{če } i = 1, \text{ vsi } p; \\ c_{1,i} & \text{če } i > 1, p = 1; \\ \min\{u_i^{(p-1)}, \min\limits_{\substack{k \\ (k,i)\in A}} \{u_k^{(p-1)} + c_{k,i}\}\} & \text{če } i > 1, p > 1. \end{cases}$$

[4] Lester R. Ford Jr., 1927–2017, ameriški matematik.

Kako uporabimo ta sistem pri računanju cen u_i najcenejših poti iz izhodišča? Najprej vidimo, da je

$$u_i = u_i^{(n-1)}.$$

Torej bomo uporabili zgornji sistem in izračunali $u_i^{(n-1)}$ za vse $i = 1, 2, \ldots, n$. Kako? Recimo množici vseh cen $u_i^{(p)}$ pri izbranem p *p-generacija* cen. Iz sistema enačb vidimo, da za izračun p-generacije rabimo $(p-1)$-generacijo, saj je enačba za $u_i^{(p)}$ rekurzivna enačba *prve* stopnje. Zato bomo računali p-generacije kar lepo po vrsti, tj. po naraščajočem $p = 1, 2, \ldots, n-1$. Psevdokoda algoritma je torej

```
procedure Bellman_Fordov_Algoritem(G(V,A,c));
begin
  for p := 1 to n-1 do u_1^(p) := 0 endfor;
  for i := 2 to n do u_i^(1) := c_{1,i} endfor;
  for p := 2 to n-1 do
    for i := 2 to n do
      u_i^(p) := min{u_i^(p-1),min_{k,k->i \in A}{u_k^(p-1)+c_{k,i}}
    endfor
  endfor
end.
```

Časovna in prostorska zahtevnost. Časovno zahtevnost bo določala dvojna zanka `for p...for i`. Njeno telo (računanje $u_i^{(p)}$) se izvede $n(n-2)$-krat, izvedba telesa pa zahteva največ $n-1$ seštevanj in največ n primerjanj (operacij min). *Časovna* zahtevnost dvojne zanke in s tem Bellman-Fordovega algoritma je $O(n^3)$. Kaj pa prostorska zahtevnost algoritma? Prostor, ki je potreben za p-generacijo cen, obsega n pomnilniških besed. Pri izračunu p-generacije pa je potrebna še prejšnja generacija. Ostale spremenljivke zahtevajo $O(1)$ pomnilniških besed. Prostorska zahtevnost Bellman-Fordovega algoritma je zato $O(n)$.

Sklep. *Časovna zahtevnost računanja najcenejših poti iz izhodišča v splošnem grafu $G(V, A, c)$ je $O(|V|^3)$, prostorska zahtevnost pa $O(|V|)$.*

Opomba. Spoznali smo algoritme za računanje cen najcenejših poti iz izbranega izhodišča najprej v acikličnih grafih, nato v grafih s pozitivnimi cenami povezav, in končno v splošnih grafih. Ves čas pa sta veljali predpostavki, da so ti grafi *povezani* (sestavljeni iz enega samega dela) in da *nimajo negativnih ciklov*. Preden bi v praksi uporabili kakega od opisanih algoritmov na danem grafu $G(V, A, c)$, bi morali preveriti, ali graf izpolnjuje oba pogoja. Zato bi zdaj lahko nadaljevali z opisom algoritmov, s katerimi preverjamo povezanost $G(V, A, c)$. Nato bi opisali, kako ugotovimo, ali $G(V, A, c)$ vsebuje negativen cikel. (Sicer že znamo s topološkim urejanjem ugotoviti, ali je $G(V, A, c)$ cikličen, a to ne bi zadoščalo: zdaj bi morali *poiskati* (*generirati*) vse cikle v $G(V, A, c)$ in za vsakega preveriti njegovo ceno.) Čeprav so ti algoritmi zanimivi, bomo raje nadaljevali po naši glavni poti.

10 Najcenejše poti med vsemi pari

Problem, ki nas zanima zdaj, je razširitev problema iz prejšnjega poglavja. Namesto iskanja cen najcenejših poti iz izbranega izhodišča, zdaj želimo za *vsako vozlišče* (= *izhodišče*) izračunati cene najcenejših poti iz njega v ostala vozlišča. Povedano drugače, za vsak par vozlišč i in j želimo izračunati ceno najcenejše poti iz i v j.

Definicija problema

Dan je utežen usmerjen graf $G(V, A, c)$, kjer je $V = \{1, 2, \ldots, n\}$ množica vozlišč, $A \subseteq V \times V$ množica usmerjenih povezav in $c : V \times V \to \mathbb{R} \cup \{\infty\}$ funkcija, ki vsakemu paru $(i, j) \in V \times V$ priredi njegovo *ceno* $c_{i,j}$. Za vsak $i \in V$ je $c_{i,i} = 0$, in če $(i, j) \notin A$, je $c_{i,j} = \infty$. Cena u_{i_z,i_k} usmerjene poti $i_z = i_1 \to i_2 \to \ldots \to i_\ell = i_k$ iz vozlišča i_z v vozlišče i_k je vsota cen na njenih povezavah,

$$u_{i_z,i_k} = \sum_{j=1}^{j=\ell-1} c_{i_j,i_{j+1}}.$$

Cikel v grafu $G(V, A, c)$ je *negativen*, če je njegova cena negativno število. Spet predpostavljamo, da $G(V, A, c)$ nima negativnih ciklov (razlog je isti kot na str. 41).

Problem: Za vsak par vozlišč $i, j \in V$ poišči ceno $u_{i,j}$ najcenejše poti iz i v j.

Motivacija

Algoritem, ki reši ta problem, se nam sam ponuja. Če vsakega od algoritmov $\mathbf{A}(G)$ za iskanje cen najcenejših poti iz izhodišča 1 grafa G prilagodimo tako, da postane izhodišče vhodni parameter algoritma $\mathbf{A}(G$,izhodisce), je rešitev na dlani: algoritem A(G,izhodisce) zaženemo n-krat, vsakokrat z drugim izhodiščem:

```
procedure Najcenejse_med_vsemi_pari(G);
begin
  for izhodisce := 1 to n do A(G,izhodisce) endfor
end.
```

Časovna zahtevnost algoritma je $O(|V|^3)$, če je $\mathbf{A}$ algoritem za aciklične G; $O(|V|^3)$, če je $\mathbf{A}$ Dijkstrov algoritem; in $O(|V|^4)$, če je $\mathbf{A}$ Bellman-Fordov algoritem.

Vseeno pa se nam zdi časovna zahtevnost $O(|V|^4)$ pri splošnih grafih velika. Bi se dalo problem rešiti hitreje? Odgovor je *da*. Prva izboljšava, t.i. *posplošeni* Bellman-Fordov algoritem, izkorišča dolčeno podobnost z matričnim množenjem in zmanjša časovno zahtevnost na $O(|V|^3 \log |V|)$. Logaritem izvira iz asociativnosti matričnega produkta, kar omogoči, da potenco M^n neke matrike M izračunamo z $\lceil \log n \rceil$ matričnimi množenji. Podrobneje pa te izboljšave ne bomo opisali, ker sta druga dva raziskovalca, Floyd[1] in Warshall,[2] odkrila boljši algoritem.

Floyd-Warshallov Algoritem

Zamisel, ki sta jo imela Floyd in Warshall v razvoju svojega algoritma, je:

- Naj bo $u_{i,j}^{(m)} \equiv$ cena najcenejše poti iz i v j, na kateri imajo vsa vmesna vozlišča oznake kvečjemu m. (Pri $m = 0$ vmesnih vozlišč ni.)
- Velja $u_{i,j} = u_{i,j}^{(n)}$.
- Najcenejša pot P iz i v j, na kateri imajo vsa vmesna vozlišča oznake $\leqslant m$,
  (a) *bodisi* ne gre čez vozlišče m
  (b) *bodisi* gre čez vozlišče m (torej je $\mathrm{P} = i \to \cdots \to m \to \cdots \to j$).
- Cena $u_{i,j}^{(m)}$ poti P je odvisna od alternative, ki velja za P; zato je
  (a) *bodisi* $u_{i,j}^{(m)} = u_{i,j}^{(m-1)}$
  (b) *bodisi* $u_{i,j}^{(m)} = u_{i,m}^{m-1} + u_{m,j}^{(m-1)}$ (zaradi načela optimalnosti).

  Pot P je splošna, zato ne vemo, katera alternativa velja. Zagotovo pa vemo, da mora biti njena dejanska cena $u_{i,j}^{(m)}$ manjša izmed alternativnih cen; torej $u_{i,j}^{(m)} = \min\{u_{i,j}^{(m-1)}, u_{i,m}^{(m-1)} + u_{m,j}^{(m-1)}\}$. To pa je *rekurzivna enačba* za $u_{i,j}^{(m)}$!
- *Začetne vrednosti* za to enačbo so $u_{i,j}^{(0)} = c_{i,j}$.

S tem sta Floyd in Warshall za splošni graf izpeljala naslednji sistem enačb:

$$u_{i,j}^{(m)} = \begin{cases} c_{i,j} & \text{če } m = 0; \\ \min\{u_{i,j}^{(m-1)}, u_{i,m}^{(m-1)} + u_{m,j}^{(m-1)}\} & \text{če } 1 \leqslant m \leqslant n. \end{cases}$$

Vseh enačb je $n^3 + n^2$ (n^3 za $m, i, j = 1, \ldots, n$ ter n^2 za $m = 0$ in $i, j = 1, 2, \ldots, n$).

[1] Robert Floyd, 1936–2001, ameriški računalnikar.
[2] Stephen Warshall, 1935–2005, ameriški računalnikar.

Ker so naš cilj vrednosti $u_{i,j}$ in je $u_{i,j} = u^n_{i,j}$, bomo izračunali iz zgornjega sistema vrednosti $u^n_{i,j}$. Kako? Spet vidimo iz rekurzivne enačbe za $u^m_{i,j}$, da m-generacijo cen izračunamo iz $(m-1)$-generacije. Torej bomo računali m-generacije po naraščajočih vrednostih $m = 1, 2, \ldots, n$. Psevdokoda algoritma je zato

```
procedure Floyd_Warshallov_Algoritem(G(V,A,c));
begin
  for i := 1 to n do
    for j := 1 to n do
      u_{i,j}^(0) := c_{i,j}
    endfor
  endfor;
  for m := 1 to n do
    for i := 1 to n do
      for j := 1 to n do
        u_{i,j}^(m) := min{u_{i,j}^(m-1),u_{i,m}^(m-1)+u_{m,j}^(m-1)}
      endfor
    endfor
  endfor
end.
```

Časovna zahtevnost. Časovno zahtevnost diktira trojna zanka. Telo zanke (računanje $u^{(m)}_{i,j}$) se izvede n^3-krat, izvedba telesa pa zahteva eno seštevanje in eno primerjanje. *Časovna* zahtevnost Floyd-Warshallovega algoritma je zato $O(n^3)$. To je bistveno izboljšanje naivnega algoritma, ki n-krat zažene Bellman-Fordov algoritem.

Prostorska zahtevnost. Vsaka generacija vsebuje n^2 števil (za vse pare i, j), zato zahteva n^2 pomnilniških besed. Ker vsako generacijo izračunamo s pomočjo prejšnje generacije, potrebujemo za obe generaciji $2n^2 = O(n^2)$ pomnilniški besed. Ostalih spremenljivk v algoritmu je konstantno mnogo, $O(1)$. Torej je prostorska zahtevnost Floyd-Warshallovega algoritma reda $O(n^2)$.

Sklep. *Časovna zahtevnost računanja najcenejših poti med vsemi pari vozlišč v splošnem grafu $G(V, A, c)$ je $O(|V|^3)$, prostorska zahtevnost pa $O(|V|^2)$.*

Mala izboljšava: računanje *na mestu*

Zanimivo je, da lahko Floyd-Warshallov algoritem implementiramo tako, da mu zadošča prostor le za *eno* generacijo, torej za vsa števila $u^{(m)}_{i,j}$, $1 \leqslant i, j \leqslant n$. Implementacija je taka, da na prostoru $(m-1)$-generacije računa m-generacijo tako, da z nobenim novim številom ne povozi kakega starega števila, ki ga bo še potreboval. Pravimo tudi, da algoritem izračuna rezultat *na mestu* (lat. *in situ*). Poglejmo, kako to dosežemo.

Predstavljajmo si, da so vsa števila $u_{i,j}^{(m-1)}$, $1 \leqslant i, j \leqslant n$, $(m-1)$-generacije v dvodimenzionalni matriki U, kjer so razporejena takole:

$$U = \begin{array}{c} \\ 1 \\ 2 \\ \\ m \\ \\ \\ \\ \\ \\ \\ \\ i \\ \\ n \end{array} \begin{array}{c} \begin{array}{ccccccccccc} 1 & 2 & & m & & & j & & & & n \end{array} \\ \left(\begin{array}{ccccccccccc} u_{1,1}^{(m-1)} & u_{1,2}^{(m-1)} & \cdots & & & & \vdots & & \cdots & & u_{1,n}^{(m-1)} \\ u_{2,1}^{(m-1)} & u_{2,2}^{(m-1)} & \cdots & & & & \vdots & & \cdots & & u_{2,n}^{(m-1)} \\ \vdots & \vdots & \ddots & & & & \vdots & & & & \vdots \\ & & & u_{m,m}^{(m-1)} & & & u_{m,j}^{(m-1)} & & & & \\ & & & \ddots & & & & & & & \\ & & & & \ddots & & \vdots & & & & \\ & & & & & \ddots & \vdots & & & & \\ & & & & & & \vdots & & & & \\ & & & & & & \ddots & & & & \\ & & & & & & \vdots & & & & \\ \cdots & \cdots & \cdots & u_{i,m}^{(m-1)} & \cdots & \cdots & u_{i,j}^{(m-1)} & \ddots & & & \\ \vdots & \vdots & & & & & & & \ddots & & \vdots \\ u_{n,1}^{(m-1)} & u_{n,2}^{(m-1)} & \cdots & & & & & & \cdots & & u_{n,n}^{(m-1)} \end{array}\right) \end{array}$$

Izračunajmo elemente m-generacije tako, kot pravi Floyd-Warshallov sistem enačb, vendar izračunano vrednost $u_{i,j}^{(m)}$ hrabro vpišimo nazaj na isto mesto v matriko U:

$$U_{i,j} = \min\{U_{i,j}, U_{i,m} + U_{m,j}\}.$$

Zanima nas, ali smo z vpisom izgubili staro vsebino $U_{i,j}$, ki jo bomo še potrebovali.

Poglejmo podrobneje, kaj se dogaja, ko računamo novo vrednost $U_{i,j}$. Izračun nove vsebine komponente $U_{i,j} = u_{i,j}^{(m)}$ uporabi tri stare vsebine komponent

$$\begin{aligned} U_{i,j} &= u_{i,j}^{(m-1)} \\ U_{i,m} &= u_{i,m}^{(m-1)} \\ U_{m,j} &= u_{m,j}^{(m-1)} \end{aligned}$$

spremeni pa le prvo od njih, $U_{i,j}$ (in še to le, če je $u_{i,m}^{(m-1)} + u_{m,j}^{(m-1)} < u_{i,j}^{(m-1)}$). Torej je novi $U_{i,j}$ odvisen samo od starega $U_{i,j}$ in od vsebin dveh komponent, ki sta v m-ti vrstici in m-tem stolpcu matrike U in ki po spremembi $U_{i,j}$ ostaneta nespremenjeni.

Zdaj pa pokažimo, da stare vsebine $U_{i,j}$ ne bomo potrebovali (zato jo res smemo zamenjati z novo). Naj bo $U_{k,\ell} \neq U_{i,j}$ poljubna druga komponenta izven m-te vrstice in m-tega stolpca. Računanje novega $U_{k,\ell}$ bo potrebovalo stari $U_{k,\ell}$ in

vsebini dveh komponent na m-ti vrstici in m-tem stolpcu. Ker $U_{i,j}$ ni na nobeni od njiju, računanje novega $U_{k,\ell}$ ne potrebuje starega $U_{i,j}$. Kaj pa, če je $U_{k,\ell} \neq U_{i,j}$ na m-ti vrstici ali m-tem stolpcu? Denimo, da je na m-ti vrstici. Tedaj je $k = m$, komponenta pa je $U_{m,\ell}$. Njena nova vrednost bo $U_{m,\ell} = \min\{U_{m,\ell}, U_{m,m} + U_{m,\ell}\}$. Toda $U_{m,m} = 0$, zato je $U_{m,\ell} = \min\{U_{m,\ell}, U_{m,\ell}\} = U_{m,\ell}$. Torej se $U_{m,\ell}$ ne spremeni, pri njenem izračunu pa starega $U_{i,j}$ očitno nismo rabili.

Sklep. *Računanje naslednje generacije je izvedljivo na prostoru prejšnje generacije.*

Zdaj lahko zapišemo Floyd-Warshallov algoritem še z zornega matrike U:

```
procedure Floyd_Warshallov_Algoritem(G(V,A,c));
begin
  for i := 1 to n do
    for j := 1 to n do
      U[i,j] := C[i,j]
    endfor
  endfor;
  for m := 1 to n do
    for i := 1 to n do
      for j := 1 to n do
        U[i,j] := min{U[i,j],U[i,m]+U[m,j]}
      endfor
    endfor
  endfor
end.
```

11 Iskanje znakovnih podnizov

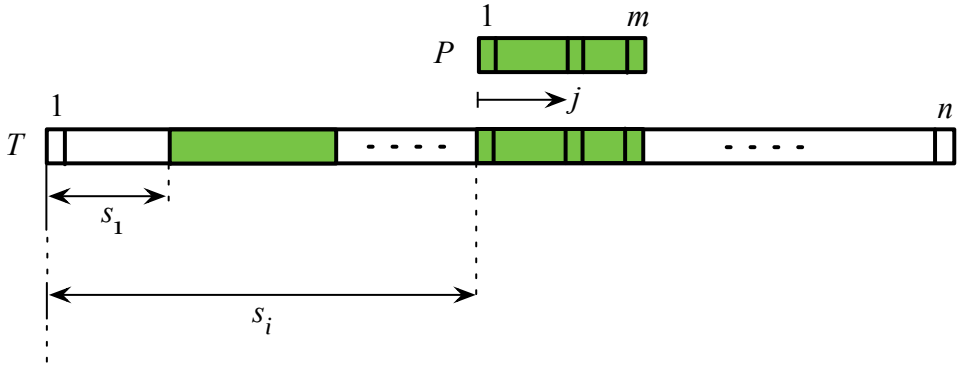
V tem poglavju se bomo ukvarjali s problemom, ki ga pogosto zastavimo urejevalniku besedil: v trenutnem besedilu poišči vse pojave danega znakovnega niza. Problem se lahko pojavi tudi v drugih okoljih, ki dopuščajo iskanje danega vzorca v neki linearno urejeni strukturi.

Definicija problema

Naj bosta T in P znakovna niza dolžin $|T| = n$ in $|P| = m$, kjer je $1 \leqslant m \leqslant n$. T imenujemo *besedilo*, P pa *vzorec*. Torej T in P nista prazna, P pa ni daljši od T. Znak na k-tem mestu v T označimo s $T[k]$, znak na k-tem mestu v P pa s $P[k]$. *Vprašanje:* Ali obstaja zaporedje naravnih števil $s_1, s_2, \ldots, s_z$ z lastnostima $z \geqslant 1$ in $0 \leqslant s_1 < s_1 < \ldots < s_z$, tako da za vsako število s_i velja

$$T[s_i + j] = P[j], \quad \text{za vse } j = 1, 2, \ldots, m?$$

Vprašanje postane bolj razumljivo s pomočjo slike 11.1:

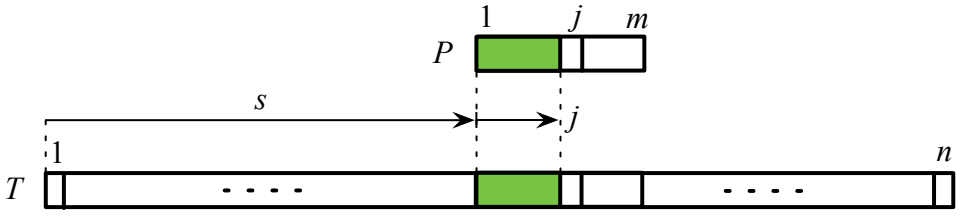


Slika 11.1 Pojavi vzorca P v besedilu T pri odmikih $s_1, s_2, \ldots, s_i$ od začetka T.

Števila $s_1, s_2, \ldots, s_z$ so torej odmiki od začetka T, kjer se pojavi P kot podniz v T. Problem torej sprašuje, pri katerih odmikih od začetka besedila T se pojavi vzorec P.

Naivni algoritem

Zamisel: za vsak $s = 0, 1, \ldots$ preveri, ali se v T pri odmiku s pojavi P (Slika 11.2).



Slika 11.2 Preverjanje, ali je P podniz v T pri odmiku s..

```
procedure Naivno_Iskanje(T,P);
begin
  for s := 0 to n-m do
     j:= 1;
     while j <= m and P[j] = T[s+j] do j := j+1 endwhile;
     if j > m then Izpisi(s) endif
  endfor
end.
```

Časovna zahtevnost v najslabšem primeru. V najslabšem primeru se pri vsakem $s = 0, 1, \ldots, n-m$ *cel* vzorec primerja z delom besedila; takrat se izvede m primerjanj pri vsakem s. Torej je skupno število primerjanj v najslabšem primeru $(n-m+1)m$. Ta izraz je funkcija dveh spremenljivk, m in n, kjer je n poljubno naravno število, m pa naravno število, omejeno z relacijo $1 \leqslant m \leqslant n$. Pri katerem m je vrednost izraza $(n-m+1)m$ največja, če se n ne spreminja? Pri katerih realnih x ima funkcija $f(x) = (n-x+1)x$ ekstreme? Pri tistih x, ki so rešitve enačbe $f'(x) = 0$, v našem primeru enačbe $-2x + n = 0$. Rešitev je ena, $x = n/2$, funkcija $f(x)$ pa ima v njej maksimum. Ker nas zanima, pri katerem *naravnem* x je vrednost izraza $(n-x+1)x$ največja, moramo izbrati $m = \lfloor n/2 \rfloor$ ali $m = \lceil n/2 \rceil$. Če izberemo prvega, ima naš izraz vrednost $(n - \lfloor n/2 \rfloor + 1)\lfloor n/2 \rfloor$, če izberemo drugega, pa $(n - \lceil n/2 \rceil + 1)\lceil n/2 \rceil$. V obeh primerih pa je ta vrednost reda $O(n^2)$.

Sklep: *V najslabšem primeru ima naivni algoritem časovno zahtevnost* $O(n^2)$.

Najslabši primer nastopi, ko je besedilo T sestavljeno iz n enakih znakov, denimo znakov a, vzorec P pa iz m znakov a. Taka besedila so možna, a redka. V praksi naletimo na besedila, ki črpajo znake iz velike abecede Σ, v kateri so črke, številke in posebni znaki. Táko besedilo je bolj naključno od niza a^n, prav tako tudi vzorec. Ker je verjetnost pojava najslabšega primera majhna, je rezultat $O(n^2)$ zgornje analize morda preveč pesimističen. Zato bomo v nadaljevanju analizirali časovno zahtevnost naivnega algoritma še za *povprečni primer*.

Časovna zahtevnost v povprečnem primeru. Naj bosta T in P naključna niza nad abecedo Σ, ki ima $\sigma \geqslant 2$ znakov. Poglejmo psevdokodo naivnega algoritma.

Kolikšno je *pričakovano* število primerjanj $P[j] = T[s+j]$ pri danem odmiku s? Označimo z $Z(j)$ verjetnost, da je j-to primerjanje zadnje. Tedaj je $Z(1)$ verjetnost, da je že prvo primerjanje zadnje. Ta verjetnost je enaka verjetnosti, da se že prva znaka $P[1]$ in $T[s+1]$ razlikujeta, ki pa je $\frac{\text{število različnih parov znakov iz } \Sigma}{\text{število vseh parov znakov iz } \Sigma} = \frac{\sigma^2-\sigma}{\sigma^2} = 1-\frac{1}{\sigma}$. Torej,

$$Z(1) = 1 - \frac{1}{\sigma}.$$

Kolikšna je verjetnost je $Z(k)$ pri $1 < k < m$? Verjetnost, da je k-to primerjanje zadnje je produkt verjetnosti $(\frac{1}{\sigma})^{k-1}$, da predhodna primerjanja *niso* bila zadnja in verjetnosti $1 - \frac{1}{\sigma}$, da je to primerjanje zadnje. Zato je pri $1 < k < m$

$$Z(k) = \left(\frac{1}{\sigma}\right)^{k-1}\left(1 - \frac{1}{\sigma}\right).$$

Verjetnost $Z(m)$ pa je produkt verjetnosti, da predhodna primerjanja *niso* bila zadnja in verjetnosti, da je to primerjanje zadnje – to pa je zagotovo že zaradi pogoja $j \leqslant m$ v zanki while. Torej,

$$Z(m) = \left(\frac{1}{\sigma}\right)^{m-1}.$$

Označimo z $\bar{j}$ *pričakovano število primerjanj* (pri danem odmiku s). Tedaj je

$$\bar{j} = \frac{1}{m}\sum_{k=1}^{m} kZ(k).$$

Zaradi enostavnosti vpeljimo $\delta := \frac{1}{\sigma}$. Tedaj:

$$\begin{aligned}
\bar{j} &= \frac{1}{m}\sum_{k=1}^{m} kZ(k) = \frac{1}{m}\Big[\sum_{k=1}^{m-1} k\delta^{k-1}(1-\delta) + m\delta^{m-1}\Big] < /\text{ker je } 1-\delta < 1/ \\
&< \frac{1}{m}\Big[\sum_{k=1}^{m-1} k\delta^{k-1} + m\delta^{m-1}\Big] = \frac{1}{m}\sum_{k=1}^{m} k\delta^{k-1} = \sum_{k=1}^{m}\frac{k}{m}\delta^{k-1} \leqslant /\text{ker je } \frac{k}{m} \leqslant 1/ \\
&\leqslant \sum_{k=1}^{m}\delta^{k-1} = \frac{1-\delta^m}{1-\delta} < /\text{ker je } \delta < 1/ < \frac{1}{1-\delta} < /\text{ker je } \delta = \frac{1}{\sigma} \leqslant \frac{1}{2}/ < 2.
\end{aligned}$$

Časovna zahtevnost naivnega algoritma v povprečnem primeru je potemtakem $(n-m+1)\bar{j} < 2(n-m+1) = O(n)$, ker je $1 \leqslant m \leqslant n$. To ni slabo, saj je linearna. Hkrati to potrjuje našo domnevo, da je $O(n^2)$ pesimističen rezultat, redek v praksi.

Zato se samo po sebi vsili naslednje vprašanje: Ali bi se dalo izboljšati tudi najslabšo časovno zahtevnost $O(n^2)$? Odgovor je *da*: izboljšan algoritem so odkrili Knuth, Morris in Pratt. Njihov algoritem, imenovan *Algoritem KMP*, ima linearno časovno zahtevnost tudi v najslabšem primeru. KONEC ZA 2019/2020.