

Algoritmi in podatkovne strukture 2

Algorithms and Data Structures 2

Borut Robič

Faculty of Computer and Information Science
University of Ljubljana



Predavanja

- ◆ Predavanja v slovenščini
- ◆ Vaje v slovenščini:
 - ◆ Začetek: v tednu 28.februar – 4.marec
 - ◆ doc. Uroš Čibej
 - ◆ As. Rok Gomišček
- ◆ Info:
 - ◆ e-Učilnica FRI

Literatura

- ◆ *Te prosojnice* (po spodnji literaturi)
- ◆ B. Robič *Algoritmi*, Založba FRI (izide letos)
- ◆ Dodatno (neobvezno):
 - ◆ Cormen et al.
 - ◆ Sedgewick et al.
 - ◆ Kleinberg & Tardos
 - ◆ Skiena
 - ◆ Sahni
 - ◆ ...

Vsebina ...

Uvod	(osnovni pojmi)
Urejanje	(navadno, spodnja meja čas.zaht., Heapsort)
Metode razvoja algoritmov	(kratki pregled)
Deli in vladaj	(Quicksort, Množenje števil, Matrično množenje, k -ti najmanjši el., DFT)
Dinamično programiranje	(Fibonaccijeva števila, Pretoki, Nahrbtnik, Najcenejše poti)
Požrešnost	(Zabojniki, Huffmanovo kodiranje, Razvrščanje opravil, Koši, Trgovski potnik)
Sestopanje	(Labirint)
Razveji in omeji	
Gola sila	
Naravna inteliganca	(Iskanje znakovnih podzaporedij)
Približnost	(Metični trgovski potnik)
Naključnost	(Razpoznavanje praštevil)

I.

Uvod



1. Asimptotična notacija

Definicija. Naj bo dana funkcija $g : \mathbb{N} \rightarrow \mathbb{N}$. Potem za $f : \mathbb{N} \rightarrow \mathbb{N}$ pišemo

- $f(n) = \mathcal{O}(g(n))$, če $\exists c > 0$, da je $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} \leq c$. f narašča **kvečjemu tako hitro** kot g .
- $f(n) = \Omega(g(n))$, če $\exists c > 0$, da je $c \leq \lim_{n \rightarrow \infty} \frac{f(n)}{g(n)}$. f narašča **vsaj tako hitro** kot g .
- $f(n) = \Theta(g(n))$, če $\exists c_1, c_2 > 0$, da je $c_1 \leq \lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} \leq c_2$. f narašča **podobno hitro** kot g .
- $f(n) = o(g(n))$, če je $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = 0$. f narašča **počasneje** od g .
- $f(n) = \omega(g(n))$, če je $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = +\infty$. f narašča **hitreje** od g .
- $f(n) \sim g(n)$, če je $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = 1$. f narašča **tako hitro** kot g .

Simbole \mathcal{O} , Ω , Θ , o , ω in \sim bi lahko definirali na ekvivalenten način brez limit:

- $f(n) = \mathcal{O}(g(n))$, če $\exists c, n_0 > 0 \forall n \geq n_0 : f(n) \leq cg(n)$.

Intuitivno: za neko konstanto $c > 0$ je od nekega zadosti velikega argumenta n_0 dalje vrednost $f(n)$ navzgor omejena z vrednostjo $cg(n)$.

- $f(n) = \Omega(g(n))$, če $\exists c, n_0 > 0 \forall n \geq n_0 : cg(n) \leq f(n)$.

Intuitivno: za neko konstanto $c > 0$ je od nekega zadosti velikega argumenta n_0 dalje vrednost $f(n)$ navzdol omejena z vrednostjo $cg(n)$.

- $f(n) = \Theta(g(n))$, če $\exists c_1, c_2, n_0 > 0 \forall n \geq n_0 : c_1g(n) \leq f(n) \leq c_2g(n)$.

Intuitivno: za neki konstanti $c_1, c_2 > 0$ je od nekega zadosti velikega argumenta n_0 dalje vrednost $f(n)$ omejena navzdol s $c_1g(n)$ in navzgor s $c_2g(n)$.

- $f(n) = o(g(n))$, če $\forall c > 0 \exists n_0 > 0 \forall n \geq n_0 : f(n) < cg(n)$.

Intuitivno: od nekega zadosti velikega argumenta n_0 dalje je vrednost $f(n)$ manjša od $g(n)$.

- $f(n) = \omega(g(n))$, če $\forall c > 0 \exists n_0 > 0 \forall n \geq n_0 : cg(n) < f(n)$.

Intuitivno: od nekega zadosti velikega argumenta n_0 dalje je vrednost $f(n)$ večja od $g(n)$.

S pomočjo zgornjih definicij dokažemo **osnovne lastnosti** asimptotične notacije:

1. $f(n) = \Theta(g(n)) \implies g(n) = \Theta(f(n)).$
2. $f(n) = \Theta(g(n)) \iff f(n) = \Omega(g(n)) \wedge f(n) = \mathcal{O}(g(n)).$
3. $c \neq 0$ konstanta $\implies \mathcal{O}(|c| \cdot g(n)) = \mathcal{O}(g(n)).$
4. $f(n) = \mathcal{O}(h(n)) \wedge g(n) = \mathcal{O}(k(n)) \implies f(n) + g(n) = \mathcal{O}(\max(h(n), k(n))).$
5. $f(n) = \mathcal{O}(h(n)) \wedge g(n) = \mathcal{O}(k(n)) \implies f(n) \cdot g(n) = \mathcal{O}(h(n) \cdot k(n)).$
6. $f(n) \cdot \mathcal{O}(g(n)) = \mathcal{O}(f(n) \cdot g(n)).$

Vaja. Poskusite dokazati. (Uporabite prejšnje definicije).

Te lastnosti uporabljam pri **razvoju izrazov** z asimptotično notacijo.
 Kako? Izraz, v katerem nastopa leva stran L kake od spodnjih vrstic, lahko preoblikujemo v izraz, kjer je L zamenjana z desno stranjo D :

$$\begin{array}{ccc}
 L & & D \\
 f(n) & \rightsquigarrow & \mathcal{O}(f(n)) \\
 c \cdot \mathcal{O}(f(n)) & \rightsquigarrow & \mathcal{O}(f(n)) \\
 \mathcal{O}(c \cdot f(n)) & \rightsquigarrow & \mathcal{O}(f(n)) \\
 f(n) - g(n) = \mathcal{O}(h(n)) & \rightsquigarrow & f(n) = g(n) + \mathcal{O}(h(n)) \\
 \mathcal{O}(f(n)) \cdot \mathcal{O}(g(n)) & \rightsquigarrow & \mathcal{O}(f(n) \cdot g(n)) \\
 \mathcal{O}(f(n)) + \mathcal{O}(g(n)) & \rightsquigarrow & \mathcal{O}(g(n)), \text{ če je } f(n) = \mathcal{O}(g(n)).
 \end{array}$$

Primer. Če bi za funkcijo $T(n)$ izračunali, da je $T(n) = 2 \log_3 n + 4n + 5n \log_6 n^7$, bi lahko z uporabo teh zamenjav ugotovili, da je $T(n) = \mathcal{O}(\log n) + \mathcal{O}(n) + \mathcal{O}(n \log n)$ in končno $T(n) = \mathcal{O}(n \log n)$. \square

Problemi – algoritmi - zahtevnost

Računski problem je vsak problem, katerega reševanje zahteva kakršno koli obliko računanja, ki jo zmore izvesti Turingov stroj ali kak njemu ekvivalentni model računanja (kot npr. RAM).

Računski problem Π definiramo tako, da zanj relevantne **formalne parametre** povežemo v neko smiselno vprašanje.

Npr.: $\Pi \equiv$ Ali graf $G(V, A)$ z dano množico vozlišč V in dano množico usmerjenih povezav $A \subseteq V \times V$ vsebuje Hamiltonov obhod? V tej definiciji so formalni parametri V , A in $G(V, A)$.

Primerek (ali **nalog**) π problema Π dobimo, ko v definiciji problema Π nadomestimo vse formalne parametre z **dejanskimi parametri**.

Npr.: $\pi \equiv$ Ali graf $G(V, E)$ z dano množico vozlišč $V = \{a, b, c, d\}$ in dano množico povezav $A = \{(a, b), (b, c), (c, d), (d, a)\}$ vsebuje Hamiltonov obhod?

Šele ko je dan (*konkreten*) primerek problema se lahko začne računanje rešitve. Primerki problema Π se razlikujejo v svojih dejanskih parametrih. Za nas bo pomembnejše, kako se razlikujejo v svoji velikosti.

Velikost primerka $\pi \in \Pi$ je dolžina besede $w(\pi)$, v kateri so kodirani dejanski parametri primerka π .

Predpostavka: kodirna abeceda ima vsaj dva znaka, kodiranje je brez redundanc.

Naša naloga: za dani računski problem Π sestaviti **algoritem** A , ki bo sposoben izračunati rešitev poljubnega primerka $\pi \in \Pi$.

Algoritem A bo imel **časovno zahtevnost** $T_A(n)$, če bo med reševanjem poljubnega primerka velikosti n opravil $T_A(n)$ korakov.

Funkcije $T_A(n)$ pogosto ne bomo znali/mogli/želeli natančno določiti, ker je spremenljiva (odvisna od arhitekture in tehnoloških lastnosti rač.). Zato bomo poskušali ugotoviti, kakšna je njena **hitrost naraščanja**.

Rekli bomo: Algoritem A ima časovno zahtevnost

- **kvečjemu** $\mathcal{O}(g(n))$, če bomo ugotovili: $T_A(n) = \mathcal{O}(g(n))$;
- **vsaj** $\Omega(h(n))$, če bomo ugotovili: $T_A(n) = \Omega(h(n))$; in
- $\Theta(k(n))$, če bomo ugotovili: $T_A(n) = \Omega(k(n))$ in $T_A(n) = \mathcal{O}(k(n))$.

Posebej bomo rekli, da je časovna zahtevnost algoritma A

- **konstantna**, če bo $T_A(n)$ konstantna funkcija: $T_A(n) = \mathcal{O}(1)$;
- **logaritmična**, če bo $T_A(n)$ logaritmična funkcija: $T_A(n) = \mathcal{O}(\log n)$;
- **polilogaritmična**, če bo $T_A(n)$ polilogaritmična funkcija: $T_A(n) = \text{poly}(\log n)$;
- **polinomska**, če bo $T_A(n)$ polinomska funkcija: $T_A(n) = \text{poly}(n)$;
- **kvazipolinomska**, če bo $T_A(n) = 2^{\text{poly}(\log n)}$;
- **subeksponentna**, če bo $T_A(n) = 2^{o(n)}$;
- **eksponentna**, če bo $T_A(n) = 2^{\text{poly}(n)}$.

Kvazipolinomski alg. je asimptotično počasnejši od vsakega polinomskega alg. (a hitrejši od vsakega exponentnega). Subeksponentni alg. je asimptotično hitrejši od vsakega eksponentnega alg. (a počasnejši od vsakega polinomskega).

Podobno bomo rekli, da ima algoritem A **prostorsko zahtevnost** $S_A(n)$, če bo med reševanjem poljubnega primerka velikosti n vsaj enkrat rabil $S_A(n)$ pomnilniških besed. Tudi pri $S_A(n)$ nas bo zanimalo asimptotično vedenje.

Funkcija $T_\Pi(n)$ bo **časovna zahtevnost problema** Π ,
če bo za Π obstajal algoritmom A s čas. zahtevnostjo $T_A(n) = T_\Pi(n)$.

Π bo imel asimptotično čas. zahtevnost reda $\mathcal{O}(g(n))$, $\Omega(h(n))$ ali $\Theta(k(n))$,
če bo za Π obstajal algoritmom A s časovno zahtevnostjo $T_A(n)$,
ki je reda $\mathcal{O}(g(n))$, $\Omega(h(n))$ ali $\Theta(k(n))$.

Podobno definiramo **prostorsko zahtevnost** $S_\Pi(n)$ problema.

II.

Urejanje



2. Navadno urejanje

Problem. Dana so števila a_1, a_2, \dots, a_n , $n \geq 1$.

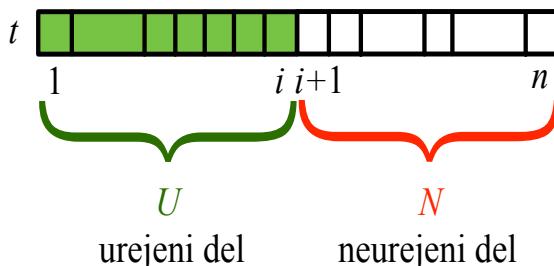
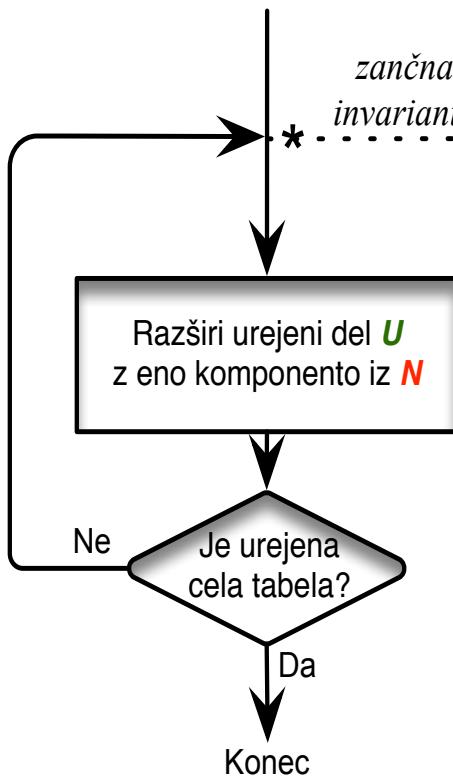
Poisci razporeditev $a_{i_1}, a_{i_2}, \dots, a_{i_n}$, da bo $a_{i_1} \leq a_{i_2} \leq \dots \leq a_{i_n}$.

Primer. Za $a_1 = 5, a_2 = 9, a_3 = 7, a_4 = 2, a_5 = 8, a_6 = 1$ je iskana razporeditev $a_6, a_4, a_1, a_3, a_5, a_2$.

Za ta problema poznamo nekaj enostavnih, **navadnih algoritmov**, za katere velja

- vsi korakoma širijo trenutno urejeni del U tabele t (od začetnega $U = t[1]$ do končnega $U = t[1..n]$);
- razlikujejo se v načinu, kako razširijo $U = t[1..i]$ z elementom iz neurejenega dela $N = t[i+1..n]$.
- če razširitev $U = t[1..i]$ zahteva $R(i)$ časa, je čas. zahtevnost urejanja cele tabele $T(n) = \sum_{i=1}^{n-1} R(i)$.

Zgradba navadnih algoritmov urejanja.



*Zančna invarianta je situacija, ki velja ob vsakem vstopu v točko * diagrama poteka.*

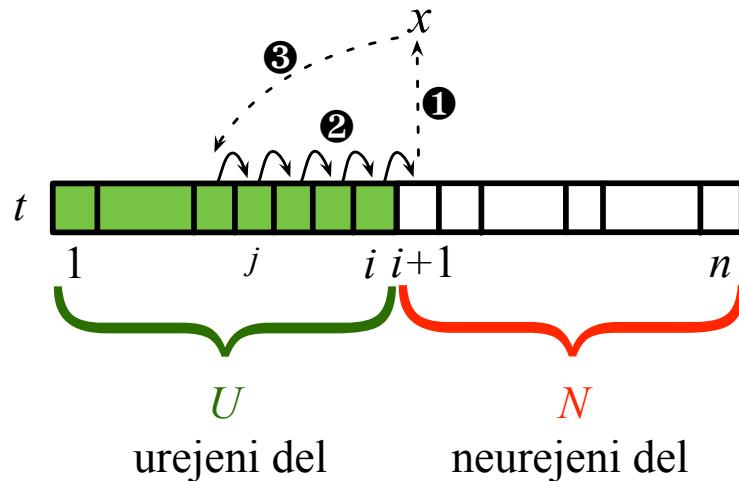
```

procedure NavadnoUrejanje(t);
begin
    for i:=1 to n-1 do
        Razsiri(t[1..i])
    endfor
end.

```

Navadni algoritmi na različne načine razširijo urejeni del $U = t[1 \dots i]$, tj. implementirajo proceduro **Razsiri**($t[1 \dots i]$):

Razširitev **z vstavljanjem**: prvi element v N , $t[i+1]$, se premesti (vrine) na ustrezeno mesto v $U = t[1..i]$.

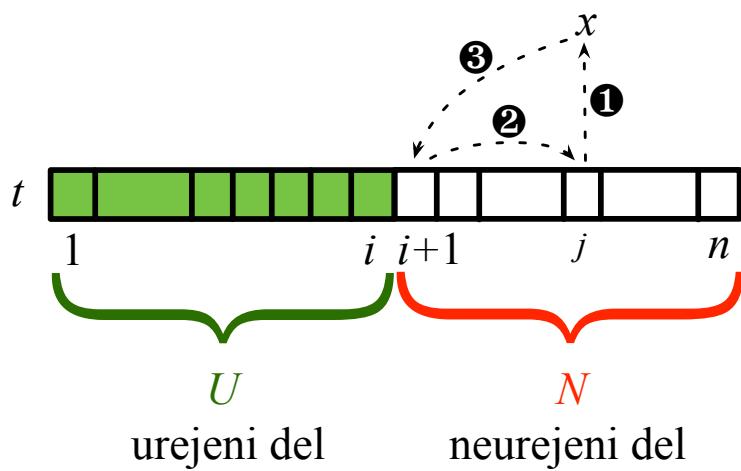


```

procedure Vstavi&Razsiri(t[1..i]);
begin
    x:=t[i+1]; j:=i;
    while j>=1 and x<t[j] do
        t[j+1]:=t[j]; j--;
    endwhile;
    t[j+1]:=x
end.
    
```

V najslabšem primeru je treba prestaviti i elementov v U ; zato je $R(i) = \mathcal{O}(i)$.

Razširitev z **izbiranjem**: prvi element v N , $t[i + 1]$, se zamenja z najmanjšim elementom v N .

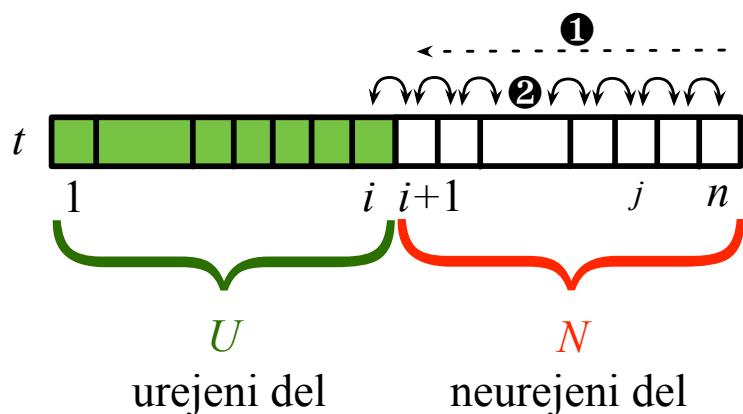


```

procedure Izberi&Razsiri(t[1..i]);
begin
    j := i+1;
    while j<=n do
        if t[j]< t[i+1] then
            x:=t[j]; t[j]:=t[i+1]; t[i+1]:=x
        endif;
        j++
    endwhile
end.
```

Da najdemo najmanjšega v N , je treba pregledati cel N ; zato je $R(i) = \Theta(n - i)$.

Razširitev **z menjavanjem**: med sprehodom po N od $t[n]$ do $t[i+1]$ se vsak $t[j]$ ($i+1 \leq j \leq n$) zamenja s $t[j-1]$, če je $t[j-1] > t[j]$.



```

procedure Menjaj&Razsiri(t[1..i]);
begin
    for j := n downto i+1 do
        if t[j-1] > t[j] then
            x := t[j-1]; t[j-1] := t[j]; t[j] := x
        endif
    endfor
end.
```

Med sprehodom po N se opravi $n-i$ primerjanj; zato je $R(i) = \Theta(n-i)$.

Ti trije načini širjenja $U[1..i]$ vodijo do treh navadnih algoritmov urejanja:

Navadno vstavljanje (InsertionSort):

```
procedure NavadnoVstavljanje(t);
begin
    for i:=1 to n-1 do
        Vstavi&Razsiri(t[1..i])
    endfor
end.
```

Časovna zahtevnost je $T(n) = \sum_{i=1}^{n-1} R(i) = \sum_{i=1}^{n-1} c_i i = c \sum_{i=1}^{n-1} i = c \frac{1}{2}(n-1)n = \mathcal{O}(n^2)$,
kjer so $c_i \in \mathbb{R}^+$ in $\min_{1 \leq i < n} c_i \leq c \leq \max_{1 \leq i < n} c_i$.

Navadno izbiranje (SelectionSort):

```
procedure NavadnoIzbiranje(t);
begin
    for i:=1 to n-1 do
        Izberi&Razsiri(t[1..i])
    endfor
end.
```

Časovna zahtevnost je $T(n) = \sum_{i=1}^{n-1} R(i) = \sum_{i=1}^{n-1} c_i(n-i) = c \sum_{i=1}^{n-1} (n-i) = \mathcal{O}(n^2)$, kjer so $c_i \in \mathbb{R}^+$ in $\min_{1 \leq i < n} c_i \leq c \leq \max_{1 \leq i < n} c_i$.

Navadno menjavanje (BubbleSort):

```
procedure NavadnoMenjavanje(t);
begin
    for i:=1 to n-1 do
        Menjaj&Razsiri(t[1..i])
    endfor
end.
```

Časovna zahtevnost je $T(n) = \sum_{i=1}^{n-1} R(i) = \sum_{i=1}^{n-1} c_i(n-i) = c \sum_{i=1}^{n-1} (n-i) = \mathcal{O}(n^2)$,
kjer so $c_i \in \mathbb{R}^+$ in $\min_{1 \leq i < n} c_i \leq c \leq \max_{1 \leq i < n} c_i$.

Opisani algoritmi za urejanje n števil imajo časovne zahtevnosti reda $\mathcal{O}(n^2)$.

Zgornja meja $\mathcal{O}(n^2)$ pa je za vse te algoritme *tesna*: vsak jo pri nekih vhodnih podatkih doseže. (Razmislite pri katerih.)

Torej vsak od navadnih algoritmov v *najslabšem primeru* zahteva $\Theta(n^2)$ časa.

Sklep. *Navadni algoritmi urejanja n števil imajo časovno zahtevnost $\Theta(n^2)$.*

3. Spodnja meja časovne zahtevnosti urejenja

Vprašanje: Ali se da števila a_1, a_2, \dots, a_n urediti hitreje?

Vsak algoritem urejanja mora vsako od n števil *vsaj prebrati*; za to rabi $\Theta(n)$ časa. Sledi, da ima vsak algoritem urejanja n števil časovno zahtevnost $\Omega(n)$.

Vprašanje: Ali je $\Omega(n)$ *tesna* spodnja meja za čas. zaht. urejanja n števil?
Vprašajmo drugače: Ali *obstaja* algoritem, ki uredi a_1, a_2, \dots, a_n v času $\Theta(n)$?

Odgovor na to bomo dobili v tem razdelku.

Kako do odgovora?

Pri navadnih algoritmih urejanja se poleg običajnih aritmetično-logičnih operacij pojavljata še dve operaciji: *primerjanje* števil a_i, a_j in *premestitev* števila a_i .

V nadaljevanju pa se bomo osredotočili le na operacijo **primerjanja** in ocenili, kolikokrat jo je treba izvesti med urejanjem a_1, a_2, \dots, a_n .

Čemu ta omejitev?

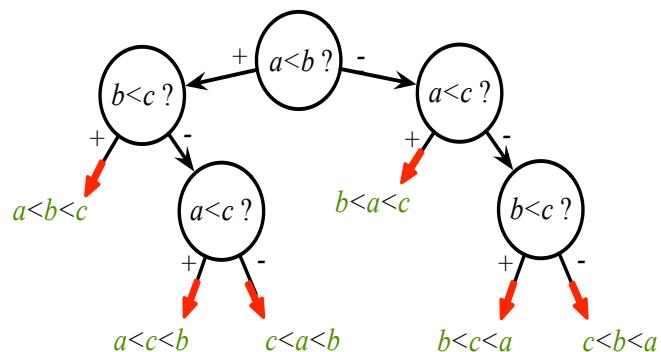
Naš namen je oceniti, *najmanj koliko primerjanj je potrebnih za ureditev števil a_1, a_2, \dots, a_n , ne glede na algoritmom urejanja.* Ker aritmetično-logičnih operacij in premestitev ne bomo šteli, bo ocena neka *spodnja meja za število vseh operacij*, ki jih mora izvesti *katerikoli znani ali še neznani algoritmom urejanja*.

Odločitvena drevesa in urejanje

Kako bi uredili a_1, a_2, \dots, a_n , če bi imeli na voljo le operacijo primerjanja a_i, a_j ?

Algoritem, ki bi smel uporabiti le to operacijo, bi moral ugotoviti, kako so ta števila razvrščena z relacijo \leq , ne bi pa jih premestil na njihova prava mesta.

Algoritem, ki to zmore, se ravna po **odločitvenem drevesu**, ki ga lahko priredimo številom a_1, a_2, \dots, a_n . Za motivacijo poglejmo primer, ko je $n = 3$.



Algoritem začne v korenju drevesa. Tam primerja a z b . Če je $a < b$, nadaljuje po povezavi $+$ („da“), sicer pa po povezavi $-$ („ne“).

V vsakem vozlišču, ki ga doseže, primerja dve števili in odvisno od rezultata nadaljuje po povezavi $+$ ali $-$.

To ponavlja, dokler ne vstopi v rdečo povezavo. Tam izve, da so a, b, c urejena tako, kot piše na koncu povezave.

Posplošitev

Zamisel posplošimo na pojubnih $n \geq 1$ števil.

Številom a_1, a_2, \dots, a_n priredimo dvojiško drevo T_n z naslednjimi lastnostmi:

- (i) vsako notranje vozlišče vsebuje neko primerjanje $a_i < a_j$ ($i \neq j$);
- (ii) v listih so vse permutacije števil a_1, a_2, \dots, a_n ;
- (iii) permutacija $a_{i_1}, a_{i_2}, \dots, a_{i_n}$ je v listu *če in samo če* $a_{i_1} \leq a_{i_2} \leq \dots \leq a_{i_n}$ izpolnjuje izide vseh primerjanj na veji do tega lista.

V splošnem lahko številom a_1, a_2, \dots, a_n priredimo več različnih dreves T_n . Oblika vsakega je odvisna od tega, kako so primerjanja pripojena notranjim vozliščem. Kljub razlikam v oblikah pa mora imeti vsako od teh dreves **$n!$ listov**.

Teorijo grafov pravi:

Če je višina $h(T)$ drevesa T definirana kot število notranjih vozlišč na najdaljši veji drevesa T , potem za vsako dvojiško drevo T z ℓ listi velja $h(T) \geq \lceil \log_2 \ell \rceil$.

V našem primeru je $T = T_n$ in $\ell = n!$, zato velja $h(T_n) \geq \lceil \log_2 n! \rceil$.

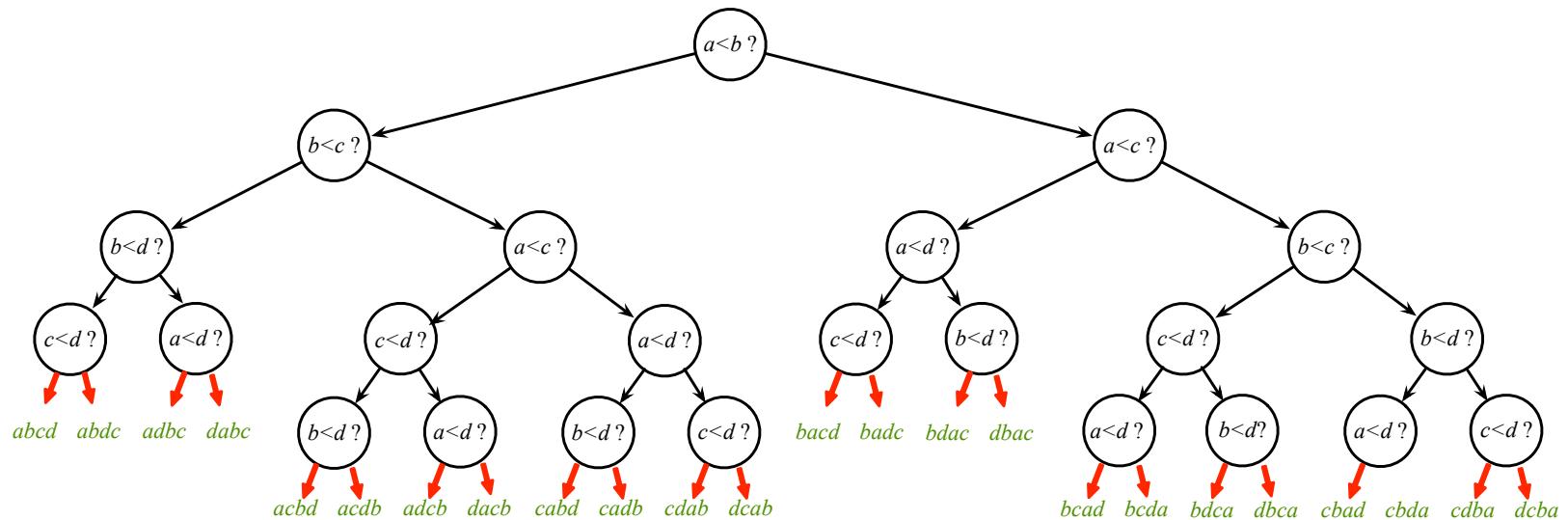
Ugotovili smo: **višina vsakega drevesa T_n je vsaj $\lceil \log_2 n! \rceil$.**

Naš algoritem bo pri nekih podatkih a_1, a_2, \dots, a_n izvedel (vzdolž neke veje) *vsaj $\lceil \log_2 n! \rceil$* primerjanj, da bo ugotovil, kako so podatki razvrščeni po velikosti. Poglejmo, koliko je $\lceil \log_2 n! \rceil$:

$$\begin{aligned} \lceil \log_2 n! \rceil &\geq \log_2 n! \approx // \text{Stirlingov obrazec: } n! \approx \sqrt{2\pi n} \left(\frac{n}{e}\right)^n // \approx \log_2 \left[\sqrt{2\pi n} \left(\frac{n}{e}\right)^n \right] = \\ &= \frac{1}{2} \log_2 2 + \frac{1}{2} \log_2 \pi + \frac{1}{2} \log_2 n + n \log_2 n - n \log_2 e = \\ &= n \log_2 n - c_1 n + c_2 \log_2 n + c_3 \geq c n \log n = \Omega(n \log n) // \text{kjer so } c_1, c_2, c_3, c \in \mathbb{R}^+. // \end{aligned}$$

Sklep. Vsak algoritem urejanja, ki uporablja operacijo primerjanja dveh števil, zahteva za ureditev vsaj enega zaporedja n števil **$\Omega(n \log n)$** operacij primerjanja.

Primer. Drevo T_4 .



4. Heapsort – Urejanje s kopico

Videli smo, da ni algoritma urejanja, ki bi uporabljal operacijo primerjanja in bil asimptotično hitrejši od $\Omega(n \log n)$.

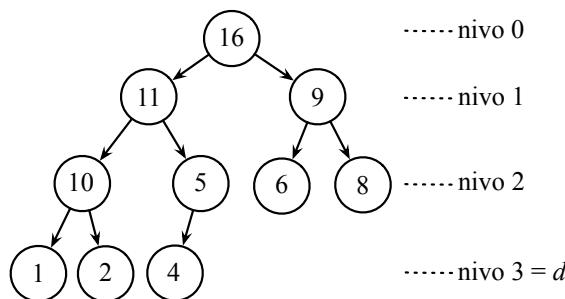
Zdaj se seveda vprašamo, ali morda obstaja algoritem, ki uporablja operacijo primerjanja in doseže najmanjšo možno časovno zahtevnost $\Theta(n \log n)$?

Odgovor je *da*; tak algoritem bomo opisali v tem razdelku. Uporabljal bo podatkovno strukturo, imenovano *kopica*.

Definicija. **Kopica** števil s_1, \dots, s_n je dvojiško drevo $T(V, E)$ z lastnostmi:

- $V = \{s_1, \dots, s_n\}$ je množica vozlišč;
- če ima vozlišče s_i sina s_j , je $s_i \geq s_j$ (oče je večji ali enak svojim sinom);
- za neki $d \in \mathbb{N}$ ima vsaka veja d ali $d - 1$ povezav; (drevo je karseda nizko);
- daljše veje so levo od krajsih (drevo je levo poravnano).

V splošnem lahko številom s_1, \dots, s_n priredimo več različnih kopic. Tu je primer.



Kopica desetih števil 4, 2, 1, 10, 5, 6, 16, 11, 9, 8 ($d = 3$; označeni so tudi **nivoji**, kjer so vozlišča).

Opazimo: (a) če je L - s_i - D kopica s korenom s_i in poddrvesoma L in D , sta tudi L in D kopici;
 (b) koren kopice je največje število v kopici;
 (c) vozlišča vzdolž veje so urejena po velikosti.

Koliko je d ? Kopica je *polno* drevo: vsak nivo ℓ , razen morda zadnji $\ell = d$, ima 2^ℓ vozlišč. Zato je d najmanjše naravno število, za katerega je $\sum_{\ell=0}^{d-1} 2^\ell < n \leq \sum_{\ell=0}^d 2^\ell$, tj. $2^d - 1 < n \leq 2^{d+1} - 1$ oz. $d - 1 < \log_2(n + 1) - 1 \leq d$. Sledi $d = \lceil \log_2(n + 1) - 1 \rceil$.

Dana števila a_1, a_2, \dots, a_n naj algoritem uredi v naraščajočem vrstnem redu, tj. razporedi v zaporedje $a_{i_1}, a_{i_2}, \dots, a_{i_n}$, da bo $a_{i_1} \leq a_{i_2} \leq \dots \leq a_{i_n}$.

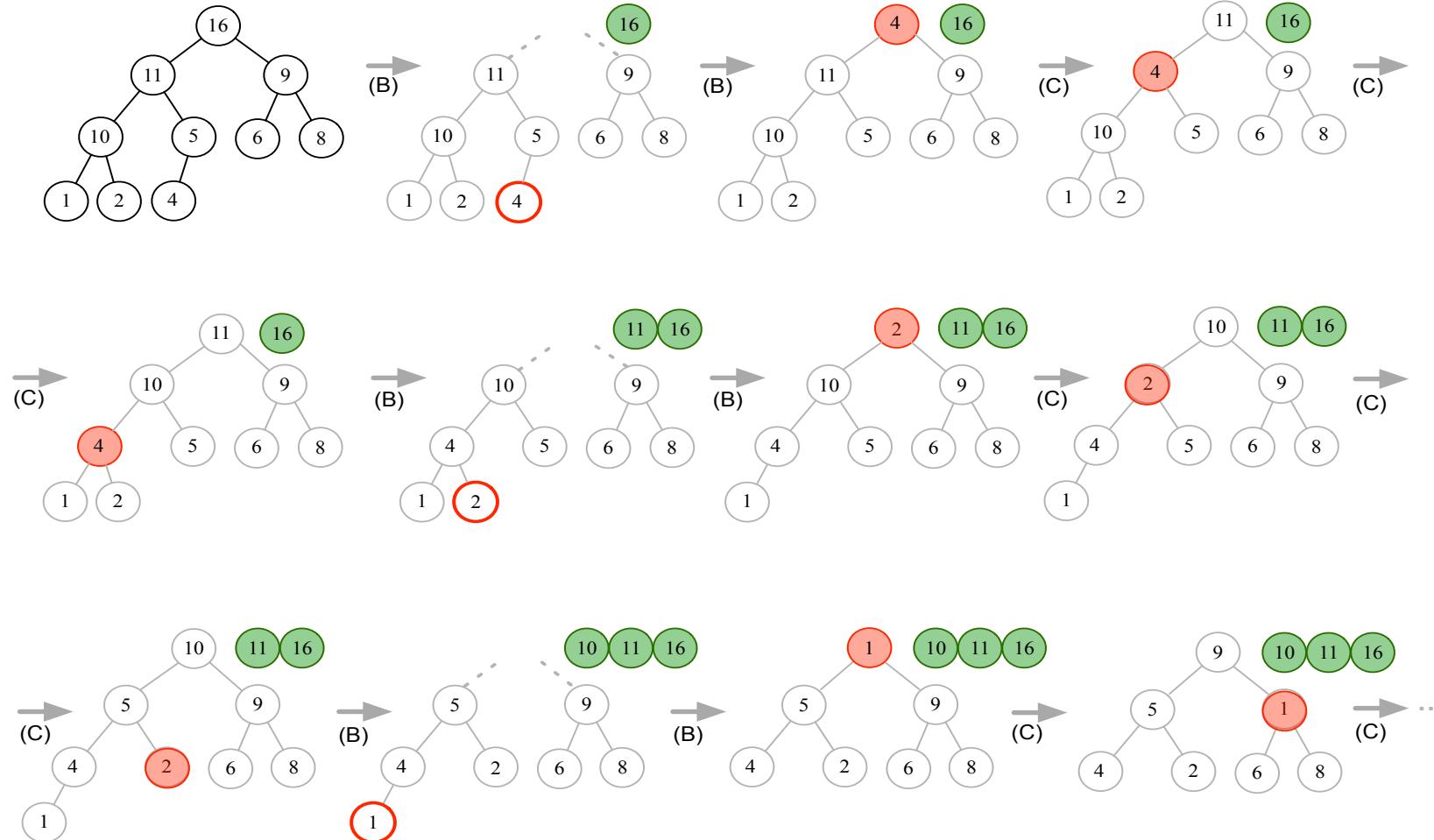
Kako bo naš algoritem uporabil kopico?

Zamisel:

- A. Iz števil a_1, a_2, \dots, a_n sestavi začetno kopico števil a_1, a_2, \dots, a_n .
- B. Izloči koren kopice in ga shrani, na njegovo mesto prestavi enega od listov (npr. skrajno desnega na najnižjem nivoju).
- C. Dobljeno drevo popravi v kopico. Kako? Prestavljeni list pogrezni vzdolž ustrezne veje na prvo mesto, kjer bo večji ali enak od tamkajšnjih sinov. Takrat bo drevo kopica, a z enim vozliščem manj.
- D. Ponovi koraka (B,C) nad to kopico.

Ko se ponavljata koraka (B,C), se nabirajo izločeni koreni, urejeni po velikosti, kopica pa kopni, dokler ni iz nje izločeno zadnje vozlišče. Takrat se algoritom ustavi.

Delovanje algoritma kaže naslednji primer.



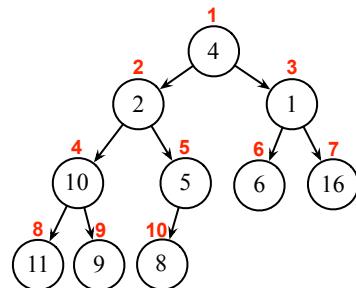
Zdaj pa to zamisel razvijmo v algoritmom. Poglejmo vsakega od korakov A,B,C,D.

A. Sestavljanje začetne kopice

Števila a_1, a_2, \dots, a_n naj bodo dana v tabeli t z n komponentami $t[i] = a_i$. Na tabelo t glejmo, kot da je zapis dvojiškega drevesa $T(t)$, definiranega takole:

- vozlišča drevesa $T(t)$ so števila $t[1], t[2], \dots, t[n]$;
- koren drevesa $T(t)$ je število $t[1]$;
- če je $t[i]$ oče v $T(t)$, sta $t[2i]$ oz. $t[2i+1]$ (če obstaja) njegov levi oz. desni sin.

Primer. Dana so števila 4, 2, 1, 10, 5, 6, 16, 11, 9, 8 v tabeli $t = \boxed{4 \ 2 \ 1 \ 10 \ 5 \ 6 \ 16 \ 11 \ 9 \ 8}$. Po zgornji definiciji tabela t opisuje tole drevo $T(t)$:



Komponente tabele t so črne, njihovi indeksi pa rdeči. Opazimo, da zaporedne komponente $t[1], t[2], \dots, t[10]$ zapolnijo nivoje drevesa $T(t)$ po vrsti od zgoraj navzdol, vsak nivo pa po vrsti od leve proti desni.

Drevo $T(t)$ v splošnem ni kopica (glej zgornji primer).

Preden se lahko začne (B), bo treba $T(t)$ preureediti v kopico. Kako?

Predpostavimo, da bi imeli na voljo proceduro `PopraviVKopico(t, i, n)`, ki bi *v dvojiškem drevesu $T(t)$ popravila drevo, katerega koren ima indeks i , v kopico – to bi znala storiti ob predpostavki, da sta obe poddrevesi tega korena že kopici.*

Potem bi lahko $T(t)$ preuredili v kopico tako, da bi po nivojih od spodaj navzgor popravili vsa drevesa s korenji na tekočem nivoju. (Ta vrstni red bi zagotovil, da bosta ob vsakem klicu `PopraviVKopico(t, i, n)` poddrevesi vozlišča i že kopici.)

Popravljanje bo teklo v želenem vrstnem redu, če bodo vozlišča obiskana po padajočih indeksih i (v zgornjem primeru **10, 9, 8, 7, 6, 5, 4, 3, 2, 1**). Listi drevesa $T(t)$ so že kopice, zato se popravljanje sme začeti z zadnjim očetom, tj. $i = \lfloor n/2 \rfloor$.

To izvede spodnji programski odsek, ki t preuredi tako, da $T(t)$ postane kopica.

```
for i:=(n div 2) downto 1 do
    PopraviVKopico(t, i, n)           | Moramo razviti!!
    endfor;
```

B. Izločanje korena in prestavljanje lista

Zdaj tabela t opisuje kopico $T(t)$.

Iz kopice $T(t)$ moramo izločiti koren in ga nadomestiti z zadnjim listom. To dosežemo tako, da zamenjamo prvo in n -to komponento tabele t in zabeležimo, da prvih $n-1$ komponent opisuje drevo, n -ta komponenta pa izločeni koren.

To se bo ponavljalo nad čedalje manjšimi kopicami. Zato bo t ostala sestavljena iz delov $t[1..r]$ in $t[r+1..n]$, kjer bo v prvem delu opis trenutnega drevesa, v drugem pa vsi dotedaj izločeni koreni. Prvi se bo manjšal, drugi večal, tj. r manjšal.

Spodnji programski odsek zamenja koren trenutne kopice z njenim zadnjim listom in vključi stari koren v drugi del tabele t tako, da zmanjša mejo r med njima.

```
x:=t[1]; t[1]:=t[r]; t[r]:=x;  
r--;
```

C. Popravljanje drevesa v kopico

Ko je koren izločen in nadomeščen z listom, je treba drevo, opisano v $t[1..r]$, popraviti v kopico. Kako?

V vsakem koraku popravljanja se prestavljeni list primerja s svojima sinovoma; če je kateri od njiju večji od lista, se list zamenja z večjim od sinov.

Tako se list pogrezne tja, kjer je večji ali enak od sinov. To opravi procedura **PopraviVKopico(t,i,r)**, ki predpostavlja, da sta poddrevesi vozlišča i že kopici.

```
procedure PopraviVKopico(t,r,i);      | V drev. t z r vozlisci popravi
begin                                     | drevo s korenom ind=i v kopico
    if i<=(r div 2) then                | Ce je koren oce,
        s:=2*i;                         | bo s indeks levega sina.
        if s+1<=r then                  | Ce obstaja se desni sin,
            if t[s]< t[s+1] then s++ endif | bo s indeks vecjega od sinov.
        endif;
        if t[i]< t[s] then              | Ce je oce manjsi od tega sina,
            x:=t[i]; t[i]:=t[s]; t[s]:=x; | ju zamenjaj.
            PopraviVKopico(t,r,s)       | Popravi v kopico poddrevo,
        endif                           | katerega koren ima indeks s.
    endif
end.
```

D. Algoritem Heapsort

Zdaj lahko zgornje odseke združimo v cel algoritem za urejanje s kopicami.

```
procedure Heapsort(t,n);
begin
    for i:=(n div 2) downto 1 do      |Sestavi zacetno kopico t[1..n]
        PopraviVKopico(t,n,i);
    endfor;
    r:=n;
    while r>1 do
        x:=t[1]; t[1]:=t[r]; t[r]:=x; |Zamenjaj v t[1..r] koren in list
        r--;                         |Izloci bivsi koren t[r]
        PopraviVKopico(t,r,1);        |Popravi drevo t[1..r] v kopico
    endwhile
end.
```

Časovna zahtevnost algoritma Heapsort

Časovna zahtevnost sestavljanja začetne kopice (korak A)?

Groba analiza koraka A: Če bi se `PopraviVKopico(t, n, i)` klicala pri vsakem $i = n, n-1, \dots, 1$ in če bi pri vsakem i izvedla $d = \lceil \log_2(n+1) - 1 \rceil$ pogrezanj, bi sestavljanje začetne kopice zahtevalo $nd = n\lceil \log_2(n+1) - 1 \rceil = \mathcal{O}(n \log n)$ pogrezanj (primerjanj in zamenjav). Sklep: A bi zahtevala $\mathcal{O}(n \log n)$ časa.

Natančna analiza koraka A: Ocena $\mathcal{O}(n \log n)$ ni napačna, je pa pregroba. S podrobnejšo analizo pokažemo (glej knjigo), da korak A zahteva $\Theta(n)$ časa!

Koraka B in C se izvedeta n -krat. Izvedba enega para B,C zahteva kvečjemu $1 + d$ operacij (primerjanj in zamenjav), izvedba n parov pa kvečjemu $n(d + 1)$, kar je zaradi $d \leq \log_2 n + 1$ reda $\mathcal{O}(n \log n)$. Natančna analiza da enak rezultat.

Sklep. Algoritem Heapsort ima časovno zahtevnost $\Theta(n \log n)$.

III.

Metode razvoja algoritmov



5. Kratek pregled metod

Želimo si splošnih pristopov k razvoju algoritmov, tj. *metod, ki nas bodo sistematično usmerjale pri naših poskusih, da bi razvili algoritmom za dani problem.* Obravnavali bomo naslednje metode:

- **Deli in vladaj.**
- **Dinamično programiranje.**
- **Požrešnost (ali Požrešna metoda.)**
- **Sestopanje.**
- **Gola sila.**
- **(Naravna inteligenco.)**

Običajno preizkusimo več metod. Različne metode vodijo do različnih algoritmov za dani problem. Šele analiza teh algoritmov in eksperimentalno testiranje na realnih primerkih problema razkrijeta, kateri algoritmom je najustreznejši.

IV.

Deli in vladaj



6. Quicksort – Hitro urejanje

Vprašanje.

Ali poleg algoritma Heapsort obstaja še kakšen algoritem, ki uporablja operacijo primerjanja števil in doseže najmanjšo možno časovno zahtevnost $\Theta(n \log n)$?

Odgovor je *da*; tak algoritem bomo opisali v tem razdelku.

Prva zamisel: Quicksort prvič

Števila a_1, a_2, \dots, a_n , ki jih želimo urediti v naraščajočem redu, so v tabeli $t[1..n]$. Zamislimo si naslednji algoritem:

- A. Izberi eno izmed komponent tabele $t[1..n]$, denimo $t[m]$. Njeno vsebino a_m označi s p , torej $p := t[m]$. (To bo t.i. *pivot* oz. *delilni element*.)

t	a_1	a_2					$\textcolor{red}{p}$								a_n
	1						m								n

- B. Prerazporedi vsebine komponent tabele t tako, da bodo t sestavljale tri tabele t_1, t_2, t_3 z lastnostmi, kot kaže spodnja slika:



- C. Uredi t_1 in t_3 , in to kar z algoritmom, ki ga razvijamo v teh točkah A,B,C. (Algoritem bo zato *rekurziven*.)

Quicksort. Zgornji algoritem imenujemo *Quicksort*. Izvajanje opisuje procedura Quicksort(t, r, s), ki tabelo $t[r..s]$ števil uredi v naraščajočem redu.

```
procedure Quicksort(t,r,s);  |Uredi tabelo t[r..s]
begin
    if s<=r then return endif; |t[r..s] je trivialna, zato koncaj
    p := Pivot(t,r,s);        |Izberi delilni element p
    j := Razdeli(t,r,s);     |Razdeli t[r..s] v t[r..j-1],t[j],t[j+1..s]
    Quicksort(t,r,j-1);      |Uredi tabelo t[r..j-1]
    Quicksort(t,j+1,s)       |Uredi tabelo t[j+1..s]
end.
```

Razlaga. Če ima t nič ali le eno komponento, je t urejena in Quicksort se konča. V nasprotnem Pivot(t, r, s) izbere p , procedura Razdeli(t, r, s) pa prerazporedi komponente tabele $t[r..s]$ ($=t$) v tabeli $t[r..j-1]$ ($=t_1$) in $t[j+1..s]$ ($=t_3$), med katerima je $t[j]=p$ ($=t_2$), in vrne j . Klica Quicksort($t, r, j-1$) in Quicksort($t, j+1, s$) uredita tabeli $t[r..j-1]$ ($=t_1$) in $t[j+1..s]$ ($=t_3$).

Raba. Števila a_1, a_2, \dots, a_n so v tabeli $t[1..n]$. Klic Quicksort($t, 1, n$) uredi t v naraščajočem vrstnem redu, in sicer *na mestu*, kjer je t .

Zdaj pa podrobneje opišimo še proceduri $\text{Pivot}(t, r, s)$ in $\text{Razdeli}(t, r, s)$.

A. Izbiranje delilnega elementa – $\text{Pivot}(t, r, s)$

Prvi način določanja pivota smo razkrili že v točki A: procedura $\text{Pivot}(t, r, s)$ izbere vsebino ene od komponent tabele $t[r..s]$, torej $p := t[m]$, kjer $r \leq m \leq s$. Kakšen naj bo indeks m ? Tu je več možnosti; na primer

- $m := r$;
- $m := s$;
- $m := \lfloor \frac{r+s}{2} \rfloor$;
- m := naključno izbran v $\{r, \dots, s\}$.

Pri drugem načinu $\text{Pivot}(t, r, s)$ izračuna p iz več komponent tabele $t[r..s]$; npr.

- $p := \frac{1}{2}(t[r] + t[s])$;
- $p := \frac{1}{3}(t[r] + t[\lfloor \frac{r+s}{2} \rfloor] + t[s])$;
- p := mediana $t[r..s]$;

Ali lahko mediano izračunamo hitro? Odgovorili bomo v enem od naslednjih poglavij. Pozor: pri drugem načinu se lahko zgodi, da je tabela t_2 prazna (zakaj?).

B. Razdelitev tabele s prerazporejanjem – $\text{Razdeli}(t, r, s)$

Radi bi, da bi prerazporejanje razmestilo vsebine komponent $t[r .. s]$ tako, da bo

1. za neki j ($r \leq j \leq s$) vsebina $t[j]$ dokončno na svojem mestu v tabeli $t[r .. s]$;
2. vsebina vsake od komponent $t[r], \dots, t[j-1]$ manjša ali enaka vsebini $t[j]$;
3. vsebina vsake od komponent $t[j+1], \dots, t[s]$ večja ali enaka vsebini $t[j]$.

Tako prerazporejanje izvede procedura $\text{Razdeli}(t, r, s)$ (na naslednji stani).

Ta prerazporedi vsebine komponent tabele $t[r .. s]$ ($=t$) v $t[r .. j-1]$ ($=t_1$) in $t[j+1 .. s]$ ($=t_3$), med katerima je $t[j]$ ($=t_2$), tako da so vse $t[r .. j-1] \leq t[j]$, ta pa \leq od vseh $t[j+1 .. s]$. Procedura vrne tudi indeks j .

$\text{Razdeli}(t, r, s)$ ima dva indeksa. Prvi gre od začetka t v desno, drugi od konca v levo. Če naletita na števili, ki sta na napačnih straneh pivota, se števili zamenjata, indeksa pa nadaljujeta do naslednje te situacije, dokler se ne srečata.

Pozor: Razdeli(t, r, s) predpostavlja, da je pivot že v prvi komponenti tabele $t[r..s]$. (Če Pivot(t, r, s) izbere kako drugo komponento $t[m]$ tabele $t[r..s]$, mora potem zamenjati še vsebine komponent $t[m]$ in $t[r]$, da bo pivot v $t[r]$.)

```
procedure Razdeli( $t, r, s$ ) return int; |Razdeli  $t[r..s]$  v  $t[r..j-1]$ ,  
begin                                         | $t[j], t[j+1..s]$  glede na  $p=t[r]$   
     $p:=t[r]; i:=r; j:=s+1;$   
    while true do  
        while  $t[++i] < p$  do if  $i=s$  break endif endwhile; |i od  $r+1$  v desno  
        while  $p < t[--j]$  do if  $j=r$  break endif endwhile; |j od  $s$  v levo  
        if  $j \leq i$  then break endif;                      |ce se i,j ne srecata,  
         $x:=t[i]; t[i]:=t[j]; t[j]:=x$                   |zamenjaj vsebine  $t[i], t[j]$   
    endwhile;  
     $x:=t[r]; t[r]:=t[j]; t[j]:=x;$                 |zamenjaj vsebine  $t[r], t[j]$   
    return j                                     |vrni indeks j  
end;
```

Opomba. Quicksort začetno nalogo „Uredi t “ problema „Uredi tabelo števil“ razdeli v dve manjši nalogi „Uredi t_1 “ in „Uredi t_3 “ istega problema. Ko manjši nalogi reši, postane (pri tem problemu) rešena tudi začetna naloga. Deljenje naloge na manjše naloge iste vrste in sestavljanje rešitev manjših nalog v rešitev dane naloge je bistvo metode **Deli in vladaj** (več o njej v naslednjem poglavju).

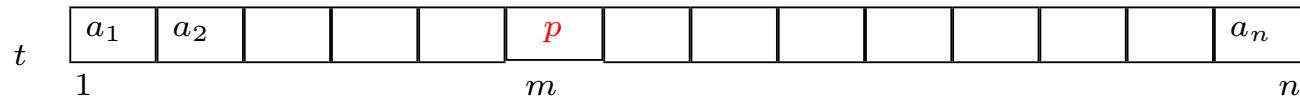
Druga zamisel: Quicksort drugič

Prejšnji algoritem *Quicksort* razdeli t v t_1, t_2, t_3 , kjer ima t_2 eno komponento p .

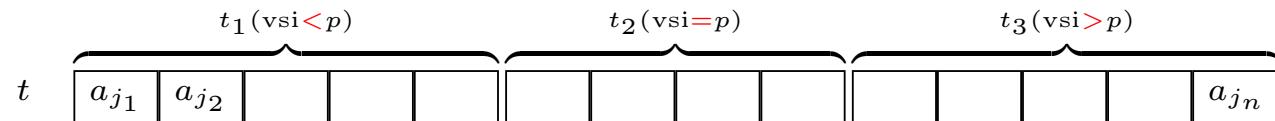
Če so a_1, a_2, \dots, a_n paroma različni ali je enakih malo, je to smiselno (ker v t_1, t_3 ostane nič ali malo števil p). Če pa se v a_1, a_2, \dots, a_n števila pogosto ponavljajo, je v t_2 bolje vključiti vse komponente z vsebinou p (da sta t_1, t_3 manjši).

To je podlaga za naslednjo različico algoritma *Quicksort*:

- Izberi v tabeli $t[1..n]$ pivot $p := t[m]$.



- Prerazporedi vsebine komponent tabele t tako, da bodo t sestavljevale tabele t_1, t_2, t_3 , kjer bodo v t_2 vsebine vseh komponent $= p$, v t_1 vsebine vseh komponent $< p$, v t_3 pa vsebine vseh komponent $> p$:



- Uredi vsako od t_1 in t_3 rekurzivno (pokliče samega sebe).

Quicksort. Procedura Quicksort(t, r, s) uredi $t[r..s]$ v naraščajočem redu.

```
procedure Quicksort(t,r,s);                                |Uredi tabelo t[r..s]
begin
    if s<=r then return endif;  |t[r..s] je trivialna, zato koncaj
    u:=r; i:=r+1; v:=s;
    p:=Pivot(t,r,s);          |Izberi delilni element p
    (u,v):=Razdeli(t,r,s);   |Razdeli t[r..s]:t[r..u-1],t[u..v],t[v+1..s]
    Quicksort(t,r,u-1);       |Uredi tabelo t[r..u-1]
    Quicksort(t,v+1,s)        |Uredi tabelo t[v+1..s]
end.
```

Razlaga. Če ima t nič ali eno komponento, je t že urejena in Quicksort se konča. Pivot(t, r, s) izbere p . Razdeli(t, r, s) prerazporedi $t[r..s]$ ($=t$) v tabeli $t[r..u-1]$ ($=t_1$) in $t[v+1..s]$ ($=t_3$), med katerima je tabela $t[u..v]$ ($=t_2$), in vrne meji u, v . Quicksort($t, r, u-1$) in Quicksort($t, v+1, s$) uredita tabeli $t[r..u-1]$ ($=t_1$) in $t[v+1..s]$ ($=t_3$).

Raba. Quicksort($t, 1, n$) uredi t (*na mestu*) v naraščajočem vrstnem redu.

A,B. Izbiranje delilnega elementa, razdelitev tabele s prerazporejanjem

Procedura $\text{Pivot}(t, r, s)$ je lahko takšna kot pri prvi različici.

Spremeni pa se $\text{Razdeli}(t, r, s)$, saj mora prerazporediti $t[r..s]$ v tri tabele (netrivialne) $t[r..u-1]$, $t[u..v]$, $t[v+1..s]$ in vrniti dva indeksa, u , v .

Razлага. Med sprehodom i od $r+1$ do v procedura $\text{Razdeli}(t, r, s)$ obnavlja u, v tako, da v vsakem trenutku velja naslednja *zančna invarianta*

$$(t[r]..t[u-1] < p) \wedge (t[u]..t[i-1] = p) \wedge (t[i]..t[v] \text{ neobiskani}) \wedge (t[v+1]..t[s] > p).$$

Ta nam pove, da bo $t[r..s]$ pri $i=v$ razdeljena v tri tabele $t[r..u-1]$, $t[u..v]$, $t[v+1..s]$, za katere bo veljalo $t[r..u-1] < p \wedge t[u..v] = p \wedge t[v+1..s] > p$.

Pozor: Razdeli(t, r, s) predpostavlja, da je *pivot* že v prvi komponenti tabele $t[r..s]$. (Pivot(t, r, s), ki bi izbral kako drugo komponento tabele $t[r..s]$, denimo $t[m]$, bi moral potem zamenjati še vsebine $t[m]$ in $t[r]$.)

```
procedure Razdeli( $t, r, s$ ) return (int,int);
begin
     $p := t[r]; u := r; v := s; i := r+1;$            |  $t[r]$  je pivot
    while  $i \leq v$  do
        if  $t[i] < p$ 
            then  $x := t[u]; t[u] := t[i]; t[i] := x; u++; i++$ 
        else if  $t[i] > p$ 
            then  $x := t[i]; t[i] := t[v]; t[v] := x; v--$ 
            else  $i++$ 
        endif
    endif
endwhile
end;
```

Analiza časovne zahtevnosti algoritma *Quicksort*

Kakšno je časovna zahtevnost $T(n)$ urejanja $t[1..n]$ z algoritmom *Quicksort*? Poglejmo prvo različico algoritma in ob korakih zapišimo njihove zahtevnosti:

```
procedure Quicksort(t,1,n);           | T(n)
begin
    if n<=1 then return endif;        | Theta(1)
    p := Pivot(t,1,n);                | Theta(1)
    j := Razdeli(t,1,n);              | Theta(n)
    Quicksort(t,1,j-1);              | T(j-1)
    Quicksort(t,j+1,n)               | T(n-j)
end.
```

$T(n)$ je vsota vseh prispevkov, $T(n) = \Theta(1) + \Theta(1) + \Theta(n) + T(j-1) + T(n-j)$. Če uporabimo osnovne lastnosti asimptotične notacije, dobimo

$$\textcolor{red}{T(n)} = T(j-1) + T(n-j) + \Theta(n). \quad (*)$$

Rešitev (*) je odvisna od j (kjer se znajde pivot, oz. od $|t_1| = j-1$ in $|t_3| = n-j$).

Glede na to definiramo **dve ekstremni izvedbi** algoritma Quicksort: pri prvi sta $|t_1|$ in $|t_3|$ karseda različna ($j = 1$), pri drugi pa karseda podobna ($j \approx \frac{n-1}{2}$).

Pri prvi ekstremni izvedbi enačba (*) preide v $T(n) = T(0) + T(n-1) + \Theta(n)$, ta pa zaradi $T(0) = \Theta(1)$ v

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

ki ima rešitev $T(n)$ z lastnostjo **$T(n) = \Theta(n^2)$** . (Analiza za *najslabši* primer)

Pri drugi ekstremni izvedbi (*) preide v $T(n) = T(\lceil \frac{n-1}{2} \rceil) + T(\lfloor \frac{n-1}{2} \rfloor) + \Theta(n)$, ta pa zaradi $T(\lceil \frac{n-1}{2} \rceil) + T(\lfloor \frac{n-1}{2} \rfloor) \leq 2T(\frac{n}{2})$ v

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

katere rešitev $T(n)$ ima lastnost **$T(n) = \Theta(n \log n)$** . (Analiza za *najboljši* primer)

Analiza za povprečni primer

V praksi je izvedba Quicksort redko ekstremna; največkrat je „nekje vmes“.

Analiza take izvedbe je bolj zapletena, ker mora upoštevati tudi *verjetnostni porazdelitvi* tabel t in pivotov, ki jih vrača *Pivot*.

Ti porazdelitvi običajno nista znani, zato bomo za prvi približek resničnosti vzeli, da so vse tabele t enako verjetne, prav tako tudi rezultati procedure *Pivot*.

Analizo bomo poenostavili (a vseeno dobili enak rezultat) še s tole predpostavko:

Predpostavka 1. Elementi tabele $t[1..n]$ naj bodo paroma različni.

Naš cilj je izračunati $\bar{T}(n)$, pričakovano časovno zahtevnost algoritma Quicksort.

Naj bo $D(k)$ dogodek, da je $p = \text{Pivot}(t, r, s)$ po velikosti k -ti najmanjsi element v $t[r..s]$. Naj bo $P(k)$ verjetnost dogodka $D(k)$.

Razdeli(t, r, s) prerazporedi $t[r..s]$ v $t[r..j-1], t[j]$ in $t[j+1..s]$. Torej je $t[j] = p$. Če je p po velikosti k -ti najmanjsi element v $t[r..s]$, je v prvi tabeli $k-1$ elementov, v drugi en element, v tretji pa $s-r+1-k$ elementov.

Pri začetni tabeli t je $r = 1, s = n$, zato je po prerazporejanju v prvi tabeli $k-1$ elementov, v drugi eden, v tretji pa $n-k$ elementov.

Zdaj imamo vse, kar rabimo, da lahko izrazimo pričakovani čas urejanja $t[1..n]$:

$$\bar{T}(n) = \sum_{k=1}^n P(k) [\bar{T}(k-1) + \bar{T}(n-k) + \Theta(n)]. \quad (**)$$

Tu je $\Theta(n)$ časovna zahtevnost procedur *Pivot* in *Razdeli*.

Enačbo $(**)$ želimo razviti, zato moramo poznati verjetnosti $P(k)$. Za začetek lahko smiselno predpostavimo, da so vse enake:

Predpostavka 2. Velja $P(k) = \frac{1}{n}$ za vsak $k = 1, \dots, n$.

Zdaj se enačba $(**)$ da preoblikovati.

Trditev. Iz $(**)$ sledi enačba

$$\bar{T}(n) = cn + \frac{2}{n} \sum_{k=0}^{n-1} \bar{T}(k). \quad (***)$$

Tudi $(***)$ je rekurzivna. Iz nje ni razvidno asimptotično vedenje funkcije $\bar{T}(n)$. Lahko pa strogo dokažemo tole:

Trditev. Za rešitev enačbe $(***)$ velja $\bar{T}(n) \leq 2(b+c)n \ln n$, kjer $b, c \in \mathbb{R}$.

Sklep. Algoritem Quicksort($t, 1, n$) ima pričakovano časovno zahtevnost $\Theta(n \log n)$.

7. Metoda *deli in vladaj*

Pri razvoju algoritmov včasih lahko uporabimo metodo, imenovano *Deli in vladaj*. Njeno bistvo je:

*Nalogo N problema P razdeli na manjše podnaloge,
te reši
in iz njihovih rešitev sestavi rešitev naloge N .*

Podnaloge so lahko primerki drugih računskih problemov. Posebno zanimivo pa je, ko so vse podnaloge primerki problema P . Ali nam bo uspelo najti take podnaloge, je odvisno od samega problema P .

Nevede smo metodo že srečali pri algoritmu *Quicksort*. Tam smo nalogo $N = \text{„uredi tabelo } t\text{”}$ razdelili s proceduro *Razdeli* na dve podnalogi $N_1 = \text{„uredi tabelo } t_1\text{”}$ in $N_2 = \text{„uredi tabelo } t_3\text{”}$. Podnalogi smo rešili kar s klicem samega algoritma *Quicksort*. Po izteku rekurzivnih klicev smo iz urejenih t_1 in t_3 ter trivialne t_2 brez dodatnega dela sestavili urejeno tabelo $t = t_1 t_2 t_3$.

Splošno.

Dana sta problem P in njegov primerek, tj. naloga $N \in P$, velikosti n . Denimo, da smo N razdelili na $p \geq 2$ podnalog N_1, \dots, N_p velikosti n_1, \dots, n_p , ki so vse primerki problema P . Denimo, da je P tak, da je treba rešiti a podnalog N_{i_1}, \dots, N_{i_a} , da lahko iz njihovih rešitev sestavimo rešitev naloge N . Ker so vse podnaloge primerki problema P , lahko za njihovo reševanje uporabimo kar A . Ogrodje algoritma A je torej tako:

```
procedure A(N);           |Algoritem A, razvit po metodi Deli in vladaj
begin
    if n<=1 then        |Ce N trivialna naloga, vrni trivialno resitev
        Vrni_resitev(N)
    else
        Razdeli(N);      |razdeli N na podnaloge N_1, ..., N_p
        A(N_i_1);         |resi N_i_1
        ...
        A(N_i_a);         |resi N_i_a
        Sestavi(N)        |sestavi resitve a podnalog v resitev naloge N
    endif
end.
```

Časovna zahtevnost algoritma A

Naj bodo $T(n)$, $R(n)$, $S(n)$ čas. zaht. procedur $A(N)$, $Razdeli(N)$ in $Sestavi(N)$. Iz algoritma A sledi, da je

$$T(n) = T(n_{i_1}) + \dots + T(n_{i_a}) + R(n) + S(n).$$

Če velikosti n_i niso znane, te enačbe ne moremo razviti v koristnejšo obliko. Če pa nam uspe razdeliti N na (približno) enako velike N_i , potem za neki $c \geq 2$ velja $n_i = \frac{n}{c}$ (ali pa $n_i \approx \frac{n}{c}$). Enačbo zdaj lahko preoblikujemo v

$$T(n) = aT\left(\frac{n}{c}\right) + R(n) + S(n).$$

Pri $n = 1$ je N trivialna in rešljiva v nekem konstantnem času $b > 0$.

Skratka: če nam uspe razdeliti N na c -krat manjše podnaloge, od katerih jih moramo a rešiti, potem se enačba za časovno zahtevnost $T(n)$ algoritma A glasi:

$$T(n) = \begin{cases} b & \text{če } n \leq 1; \\ aT\left(\frac{n}{c}\right) + R(n) + S(n) & \text{če } n > 1, \end{cases} \quad (*)$$

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

Kako izračunati $T(n)$ iz rekurzivne enačbe $(*)$?

Pišimo

$$f(n) = R(n) + S(n).$$

$f(n)$ je skupen čas, potreben za razdelitev N na p podnalog in sestavljanje rešitev a podnalog v rešitev naloge N . Zanimali nas bodo problemi, kjer $f(n)$ narašča kvečjemu polinomsko hitro.

V nadaljevanju izračunamo $T(n)$ za splošno $f(n)$, potem pa za polinomsko $f(n)$.

$T(n)$ za splošno funkcijo $f(n)$

Iz enačbe (*) lahko izpeljemo naslednji izraz za $T(n)$:

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

$T(n)$ za polinomsko funkcijo $f(n) = bn^d, d \geq 0$

Če ta $f(n)$ vstavimo v (**) dobimo

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

V oglatih oklepajih je geometrijska vrsta. Njena vsota je odvisna od velikosti $\frac{c^d}{a}$ v primerjavi z 1. Zato analiziramo vse tri možnosti.

Ugotovimo tole:

- če je $\frac{c^d}{a} > 1$, potem je $T(n) = \Theta(n^d)$
- če je $\frac{c^d}{a} = 1$, potem je $T(n) = \Theta(n^d \log n)$
- če je $\frac{c^d}{a} < 1$, potem je $T(n) = \Theta(n^{\log_c a})$

in strnemo v naslednji izrek.

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

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

kjer so $a \geq 1$, $b > 0$, $c \geq 2$ in $d \geq 0$, 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 1. Preizkusimo izrek na algoritmu *Quicksort*. Če pivot razdeli tabelo na prvo in tretjo podtabelo (srednja je trivialna, zato $p = 2$), ki sta vedno enako veliki, je *Quicksort* oblike A za $c \approx 2$. Ostali parametri so: $a = 2$, ker je treba urediti dve podtabeli; $b > 0$ konstanta brez vpliva; in $d = 1$, ker ima razdelitev tabele na tri podtabele 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 $T(n) = \Theta(n^d \log n) = \Theta(n \log n)$. To se sklada z analizo algoritma *Quicksort* za najboljši primer.

Primer 2. Preizkusimo izrek na algoritmu *BinSearch* za *dvojiško iskanje* elementa e v urejeni tabeli. Algoritem poznamo: če je srednji element s tabele enak iskanemu e , je iskanje uspešno, sicer se iskanje nadaljuje v podtabeli levo oz. desno od s (če $e < s$ oz. $s < e$). *BinSearch* je torej oblike A . Očitno je $p = 2$ (vedno sta dve podtabeli); $c \approx 2$ (podtabeli sta enako veliki); $a = 1$ (iskanje se nadaljuje v eni podtabeli); $d = 0$ (delitev 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.

8. Množenje števil

Naj bosta dani n -mestni števili $a = a_{n-1} \dots a_1 a_0$ in $b = b_{n-1} \dots b_1 b_0$, zapisani v mestnem številskem sistemu z bazo B . Potem je njun produkt m -mestno število $ab = c = c_{m-1} \dots c_1 c_0$, kjer je $m \geq n$.

Problem. *Koliko množenj je potrebnih za izračun produkta dveh n -mestnih števil?*

Poglejmo, koliko množenj zahteva algoritmom, ki smo se ga naučili v osnovni šoli.

$$\begin{array}{ccccccccc}
 a_{n-1} & a_{n-2} & \dots & a_2 & a_1 & a_0 & \times & b_{n-1} & b_{n-2} & \dots & b_2 & b_1 & b_0 \\
 \hline
 u_n & u_{n-1} & u_{n-2} & \dots & u_2 & u_1 & u_0 & & & & & & \\
 & v_n & v_{n-1} & \dots & v_3 & v_2 & v_1 & v_0 & & & & & \\
 & w_n & \dots & w_4 & w_3 & w_2 & w_1 & w_0 & & & & & \\
 & & & & & & & \ddots & & & & & \\
 & & & & & & & & z_n & z_{n-1} & \dots & z_3 & z_2 & z_1 & z_0 \\
 \hline
 c_{m-1} & c_{m-2} & c_{m-3} & \dots & \dots & \dots & \dots & c_n & c_{n-1} & \dots & c_3 & c_2 & c_1 & c_0
 \end{array}$$

Računanje vsake vrstice zahteva n množenj $a_i \cdot b_j$ (in kvečjemu n seštevanj); zato računanje vseh n vrstic zahteva n^2 množenj $a_i \cdot b_j$ (in največ n^2 seštevanj). Končno seštevanje vrstic zahteva kvečjemu $n(n - 1) + n = n^2$ seštevanj.

Sklep. Šolski algoritem zmnoži dve n -mestni števili v času reda $\Theta(n^2)$.

Ali se da zmnožiti dve n -mestni števili hitreje kot v času $\Theta(n^2)$? Se ju da zmnožiti v času $\Theta(n^\alpha)$ za neki $\alpha < 2$, ali pa morda celo v času $\Theta(n \log n)$?

Hitreje od $\Theta(n)$ se ne da, saj je treba izračunati $\Theta(n)$ števil c_0, c_1, \dots, c_{m-1} .

Pokazali bomo, da se ju da zmnožiti v času $\Theta(n^{\log_2 3}) = \Theta(n^{1.584})$.

Karatsubov algoritem

Za izračun produkta ab je ruski matematik Anatolij Katsuba razvil hitrejši algoritem, in sicer z metodo *Deli in vladaj*: dano nalogo „Izračunaj $c=ab$ “ razdeli v tri manjše naloge „Izračunaj $z_1=x_1y_1$ “, „Izračunaj $z_2=x_2y_2$ “ in „Izračunaj $z_3=x_3y_3$ “, jih reši in iz njihovih rešitev z_1, z_2, z_3 sestavi rešitev c dane naloge.

Kako je Karatsuba iz a, b določil števila x_i, y_i in iz delnih rešitev z_1, z_2, z_3 sestavil c ? Števili a in b zapišimo kot

$$a = a_L B^m + a_D$$

$$b = b_L B^m + b_D$$

kjer je

$$\begin{array}{c} \overbrace{a_{n-1} \dots a_m}^{a_L} \overbrace{a_{m-1} \dots a_0}^{a_D} \\[10pt] \overbrace{b_{n-1} \dots b_m}^{b_L} \overbrace{b_{m-1} \dots b_0}^{b_D} \end{array}$$

in $a_D, b_D < B^m$. Potem je iskani produkt c enak

$$ab = (a_L B^m + a_D)(b_L B^m + b_D) = \underbrace{a_L b_L}_{z_2} B^{2m} + \underbrace{(a_L b_D + a_D b_L)}_{z_1} B^m + \underbrace{a_D b_D}_{z_0}.$$

Karatsuba je opazil, da se da z_0, z_1, z_2 izračunati s samo *tremi* množenji takole:

- najprej izračunamo z_0 in z_2 (po njuni definiciji: $z_0 = a_D b_D$ in $z_2 = a_L b_L$),
- potem pa z_1 takole: $z_1 = (a_L + a_D)(b_L + b_D) - z_2 - z_0$.

Tako izračunani z_1 je enak $z_1 = a_L b_D + a_D b_L$ iz definicije.

Ta izboljšava zmanjša število podproblemov, ki jih je treba rešiti, z $a = 4$ na $a = 3$. Druga dva parametra algoritma pri tem ostaneta nespremenjena.

Ker je še vedno $\frac{c^d}{a} = \frac{2^1}{3} < 1$, bo zahtevnost Karatsubovega algoritma

$$T(n) = \Theta(n^{\log_c a}) = \Theta(n^{\log_2 3}) = \Theta(n^{1.58}).$$

To pa je hitreje od šolskega algoritma!

```

procedure Karatsuba(a,b) return integer; |Vrne produkt a*b
begin
    if a<B or b<B then retun a*b           |Ce je a ali b enomestno...
    else                                |
        n := min(size(a),size(b));          |...sicer ju razdeli:
        m := n div 2;                      |krajse med a,b je n-mestno
        (aL,aD) := split(a,m);            |razdeli a,b v aL,aD,bL,bD
        (bL,bD) := split(b,m);
        z0 := Karatsuba(aD,bD);          |izracunaj z0,z1,z2
        z2 := Karatsuba(aL,bL);
        z1 := Karatsuba(aL+aD,bL+bD)-z2-z0;
        return shift_left(z2,2*m)+shift_left(z1,m)+z0      |sestavi a*b
    endif
end.

```

Razlaga. Operacija `size(x)` določi, koliko mesten je argument `x`. Najbolje je, če sta oba a in b isti potenci števila 2, kar lahko dosežemo, če pri krajšem v njegovo dolžino štejemo ustrezno število vodilnih ničel. Števila a_L, a_D, b_L, b_D dobimo z operacijo `(xL,xD) := split(x,m)`, ki razdeli x v xL in m -mestni xD . Pri sestavljanju rezultata iz z_0, z_1, z_2 uporabimo operacijo `shift_left(x,y)`, ki pomakne x za y mest v levo. Če nam te operacije niso na voljo, si pomagamo z množenjem oz. deljenjem z B^m .

9. Množenje matrik

Če sta $A = (a_{ij})$ in $B = (b_{ij})$ matriki reda $n \times n$, je njun produkt $C = AB$ matrika reda $n \times n$ s komponentami $c_{ij} = \sum_{k=1}^n a_{ik}b_{kj}$. Običajni algoritem za izračun matrike C 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]
            endfor
        endfor
    endfor
end.
```

Izračun C torej zahteva n^3 skalarnih množenj in $n^2(n - 1)$ skalarnih seštevanj. Časovna zahtevnost algoritma je zato $\Theta(n^3)$. Ker so aditivne operacije $(+, -)$ hitrejše od multiplikativnih (\times, \div) , jih lahko zanemarimo, ne da bi to bistveno vplivalo na asimptotično časovno zahtevnost algoritma.

Problem. Ali se da matriki reda $n \times n$ zmnožiti v času, manjšem od $\Theta(n^3)$?

Ta čas je gotovo vsaj $\Omega(n^2)$, saj je treba izračunati n^2 komponent matrike C .

Poskus z metodo Deli in vladaj

Poskusimo razviti z metodo Deli in vladaj boljši algoritem za množenje matrik.
Predpostavka: n je potenca števila 2 (da bo lažje 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 lahko izrazimo matrični produkt AB z matričnimi produkti 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

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

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

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

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

Nalogo AB smo razdelili na 8 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 $8 \cdot T\left(\frac{n}{2}\right)$, ki je potreben za izračun 8 matričnih produktov $A_{ij}B_{kl}$, in časa $4\frac{n}{2} \cdot \frac{n}{2}$, ki je potreben za izračun 4 matričnih vsot (vsaka zahteva $\frac{n}{2} \cdot \frac{n}{2}$ skalarnih vsot). Torej je $T(n) = 8 \cdot T\left(\frac{n}{2}\right) + 4 \cdot \frac{n}{2} \cdot \frac{n}{2}$ oz.

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

Ali je časovna zahtevnost $T(n)$ tega algoritma manjša od $\Theta(n^3)$? Glavni izrek metode Deli in vladaj nam za funkcijo $T(n)$ pove naslednje: ker so v zgornji enačbi parametri $p = 8, a = 8, b = 1, c = 2, d = 2$, 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 čas. zahtevnosti algoritma *MatProdukt*.

Da bi dosegli *asimptotično* manjši $T(n)$, bi morali najti taka števila a, c, d , ki bi izpolnila zahtevo

$$\frac{c^d}{a} < 1 \quad \wedge \quad \log_c a < 3.$$

Taka števila je lahko najti: npr., če obdržimo $c = 2, d = 2$, zahtevo izpolnijo $a = 5, 6, 7$.

Izziv pa je sestaviti *algoritem* vrste Deli in vladaj, ki bo imel take parametre. Npr., če izberemo parametre $a = 7, c = 2, d = 2$, to pomeni, da bi moral tak algoritom izračunati C (štiri podmatrike C_{ij}) s pomočjo le 7 produktov $A_{ij}B_{kl}$.

Je to sploh možno?

Strassenovo matrično množenje

Volker Strassen je leta 1969 odkril, da je to možno! Bistvo njegovega algoritma je v drugačnem računanju podmatrik C_{ij} . Tule je njegovo odkritje.

Če definiramo naslednjih osem matričnih produktov P_{ij} ,

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

potem lahko štiri podmatrike C_{ij} izračunamo takole:

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

Ta definicija je zvita: izrazi za C_{ij} so *linearni*, produkta P_{11} in P_{21} pa *enaka!* Ker bo treba izračunati le enega od njiju, bo za izračun vseh štirih C_{ij} potrebnih le *sedem* matričnih množenj.

S predpostavko, da je n potenca števila 2, Strassenov algoritem zapišemo takole:

```
procedure Strassen(A,B,n) return C;
begin
    if n=1 then C := AB      |Ce sta A in B trivialni matriki (skalarja)
    else
        (A_11,A_12,A_21,A_22) := Razdeli(A,n);          |Razdeli A v bloke
        (B_11,B_12,B_21,B_22) := Razdeli(B,n);          |Razdeli B v bloke
        P_11 := Strassen(A_11+A_22, B_11+B_22, n/2);
        P_12 := Strassen(A_12-A_22, B_21+B_22, n/2);
        P_13 := Strassen(A_11+A_12, B_22, n/2);
        P_14 := Strassen(A_22, B_11-B_21, n/2);
        P_22 := Strassen(A_21-A_11, B_12+B_11, n/2);
        P_23 := Strassen(A_22+A_21, B_11, n/2);
        P_24 := Strassen(A_11, B_22-B_12, n/2);
        C_11 := P_11+P_12-P_13-P14;                      |Sestavi bloke Cij ...
        C_12 := P_13-P_24;
        C_21 := P_23-P_14;
        C_22 := P_21+P_22-P_23-P24;
        C := Sestavi(C_11,C_12,C_21,C_22, n/2)         |... in iz njih C
    endif
end.
```

Kakšna je časovna zahtevnost 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 $T(n) = 7 \cdot T\left(\frac{n}{2}\right) + 18 \cdot \frac{n}{2} \cdot \frac{n}{2}$, oz.

$$T(n) = 7 T\left(\frac{n}{2}\right) + 4.5 n^2.$$

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 po Glavnem izreku metode Deli in vladaj

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

Sklep. Strassenovo množenje matrik reda $n \times n$ ima časovno zahtevnost $\Theta(n^{2.80735})$.

10. k -ti najmanjši element

k -ti najmanjši element tabele t je element, od katerega je manjših ali enakih natanko $k - 1$ elementov tabele t . To je element, ki bi bil na k -tem mestu v urejeni t . Npr., v $(7,2,2,5,6,1)$ je 4-ti najmanjši element 5, ker je na 4. mestu v $(1,2,2,5,6,7)$.

Problem: V neurejeni tabeli t z n števili poišči k -to najmanjše število ($1 \leq k \leq n$).

Poишčimo kako spodnjo in zgornjo mejo za časovno zahtevnost $T(n)$ tega problema. Meji nam bosta pomagali, da ne bomo iskali algoritma, ki ga sploh ni ali pa algoritma, ki bi bil nepotrebno počasen. Najbolje bi bilo, če bi našli *tesni* meji za $T(n)$, tj. *največjo spodnjo* in *najmanjšo zgornjo* mejo za $T(n)$. Žal je iskanje tesnih mej pogosto zelo zahtevno.

Na rešitev našega problema lahko vpliva vsak element tabele. Zato mora vsak algoritmom za ta problem *vsaj prebrati vseh n elementov*. Zato je zahtevnost vsakega algoritma vsaj linearja, tj. $T(n) = \Omega(n)$. Torej ne bomo iskali algoritmov, ki bi imeli časovno zahtevnost asimptotično manjšo od $\Theta(n)$.

Po drugi strani pa si takoj lahko zamislimo *očiten algoritem*, ki reši naš problem. Ta algoritem (1) uredi tabelo t (v času $\Theta(n \log n)$, npr. z algoritmom *Heapsort*) in (2) vrne element na njenem k -tem mestu. Torej je smiselno iskati algoritem, ki bo imel časovno zahtevnost $T(n)$ asimptotično manjšo od $\Theta(n \log n)$.

Zamisel.

Metoda Deli in vladaj nas napelje, da si zamislimo naslednji algoritem:

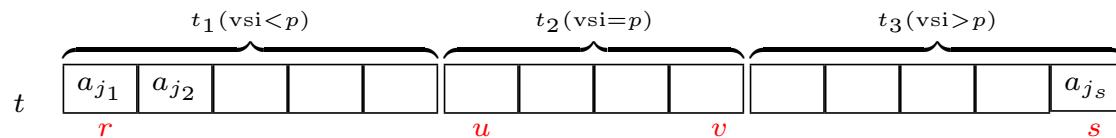
- Izberimo element p (*pivot*) in prerazporedimo elemente tabele t v tabele t_1, t_2, t_3 , da bodo v t_1 vsi elementi $< p$, v t_2 vsi $= p$, v t_3 pa vsi $> p$.
- Iz velikosti $|t_1|$ in $|t_2|$ lahko ugotovimo, v kateri od t_1, t_2, t_3 je k -ti najmanjši element tabele t . Namreč: če je $k \leq |t_1|$, je v t_1 ; če je $|t_1| < k \leq |t_1| + |t_2|$, je v t_2 ; in če je $k > |t_1| + |t_2|$, je v t_3 . Iskanje bomo zato nadaljevali le v tisti tabeli.
- Če je v t_1 , z rekurzivnim klicem tega algoritma poiščemo k -ti najmanjši element v t_1 ; če je v t_2 , je iskani element p ; in če je v t_3 , z rekurzivnim klicem tega algoritma poiščemo $k - |t_1| - |t_2|$ -ti najmanjši element v t_3 .

Torej se reševanje trenutne naloge nadomesti z reševanjem ene od treh podnalog.

Zamisel algoritma je zapisana spodaj. Dodali smo ji del, ki reši trivialno veliko nalogo (nalogo z manj kot n_{\min} elementi za neki n_{\min} , ki ga bomo morali še določili). Procedura *Uredi* je lahko kako od navadnih urejanj, proceduri *Pivot* in *Razdeli* pa smo opisali pri algoritmu *Quicksort*.

```
procedure Isci(k,t) return int;    |k-ti najmanjsi element v tabeli t
begin
    if |t| < n_min then Uredi(t); return(t[k])
    else
        p := Pivot(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(p)
        else Isci(k-|t_1|-|t_2|,t_3)
        endif
    endif
end.
```

Oznake t, t_1, t_2, t_3 zamenjajmo z oznakami $t[r..s], t[r..u-1], t[u..v], t[v+1..s]$, da postanejo dosegljive meje r, s, u, v teh tabel. Glede na to proceduram prilagodimo njihove argumente, procedura *Razdeli* pa naj vrne še meji u, v tabele t_2 .



```

procedure Isci(k,t,r,s) return int;           | k-ti najmanjsi v t[r..s]
begin
    if s-r+1 < n_min then Uredi(t,r,s); return(t[r+k-1])
    else
        p := Pivot(t,r,s);
        (u,v) := Razdeli(t,r,s);
        if k <= u-r then Isci(k,t,r,u-1)
        else if k <= v-r+1 then return(p)
        else Isci(k-v+r-1,t,v+1,s)
        endif
    endif
end.

```

Raba. Števila so v $t[1..n]$. $\text{Iscli}(k, t, 1, n)$ vrne k -ti najmanjši element v t .

Vprašanja: *Koliko je n_{\min} ? Kako izbrati p ? Kakšna je časovna zahtevnost algoritma?*

Enostavni algoritem

Poiskemo odgovore po najlažji poti: naj bo $n_{\min} = 1$ in p naključno izbran element tabele t . Kakšno časovno zahtevnost $T(n)$ ima tedaj ta algoritem?

Analizirajmo algoritem za *najslabši primer*. V pesimističnem scenariju bo p vedno največji element v t , ki bo poleg tega en sam. Zato bo v t_1 $n-1$ elementov, v t_2 en sam element p , t_3 pa bo prazna. Iskanje se bo nadaljevalo v t_1 , kjer se bo po pesimističnem scenariju zgodilo podobno. Torej bo v najslabšem primeru $T(n) = T(n-1) + \Theta(n)$. To rekurzivno enačbo smo rešili pri analizi algoritma *Quicksort* za najslabši primer, zato vemo, da je $T(n) = \Theta(n^2)$. To pa nam ni všeč, saj že *očitni algoritem* vedno najde iskani element v času $\Theta(n \log n)$.

Blum-Floyd-Pratt-Rivest-Tarjanov algoritem *IsciBFPR*T

Enostavni algoritem ima v najslabšem primeru kvadratno časovno zahtevnost. Morda se nam zdi, da je vzrok za to v naključnem izbiranju pivota p . Toda: tudi če bi p izbirali s kakim že omenjenim pravilom ($p=\text{prvi}$; $p=\text{zadnji}$; $p=\text{srednji}$ v t) bi imel algoritem v najslabšem primeru kvadratno časovno zahtevnost.

Morda pa bi bilo treba p izbirati bolj premišljeno? Morda tako, da nobena od tabel t_1 in t_3 ne bi bila „pretirano“ večja od druge, saj bi bilo potem manj pomembno, v kateri od njiju se bo nadaljevalo iskanje. (Tabela t_2 nas ne skrbi, ker „iskanje“ v njej vrne p v času $\Theta(1)$.) Vprašanje je torej:

Kako zagotoviti, da nobena od dolžin $|t_1|$ in $|t_3|$ ne bo „pretirano“ večja od druge?

(Kaj pomeni „pretirano“ nam zdaj še ni povsem jasno.)

Odgovor so našli M. Blum, R. Floyd, V. Pratt, R. Rivest in R. Tarjan.
Pivot p moramo izračunati z novo proceduro *PivotBFPR*T takole:

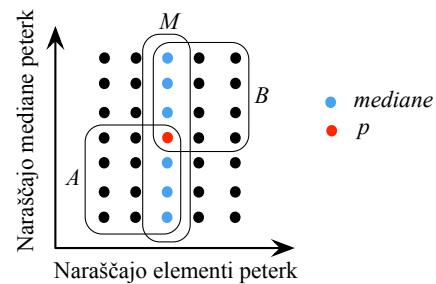
1. Tabelo t razdelimo v peterke (tabele po pet zaporednih elementov).
2. V vsaki peterki poiščemo njeno mediano (tretji najmanjši element).
3. Naj bo M tabela vseh dobljenih median.
4. Potem naj bo p mediana elementov tabele M .

Ta p uporabimo v proceduri *Razdeli*, ki razdeli t v tabele t_1, t_2, t_3 .

Prednost novega pivota p pa nam razkrije naslednja trditev.

Trditev. Po razdelitvi tabele t s tako izračunanim p velja $|t_1| \leq \frac{3}{4}|t|$ in $|t_3| \leq \frac{3}{4}|t|$.

Dokaz. Uredimo M , vsaki mediani pritaknimo ostale štiri elemente iz njene peterke, da postane urejena, in peterke organizirajmo kot kaže slika. V A so vsi elementi tabele t , ki so $\leq p$. (Podobno so v B vsi, ki so $\geq p$; teh je $|B| = 3\lfloor \frac{|M|}{2} \rfloor = 3\lfloor \frac{1}{2} \cdot \frac{|t|}{5} \rfloor = 3\lfloor \frac{|t|}{10} \rfloor$.) Ker so v t_1 vsi $< p$, je $t_1 \subseteq A$. Sledi $|t_1| \leq |A| \leq |t| - |B| = |t| - 3\lfloor \frac{|t|}{10} \rfloor \leq \lceil \frac{7}{10}|t| \rceil \leq \frac{3}{4}|t|$. (Podobno za $|t_3| \leq \frac{3}{4}|t|$.) \square



Če je p izračunan z novim postopkom, po razdelitvi t v t_1, t_2, t_3 nobena od t_1 in t_3 ni „pretirano“ večja od druge, kjer „pretirano“ pomeni „več kot trikrat“.

(Zakaj? Ker je $|t_1| + |t_3| = |t| - |t_2|$, ima v skrajnem primeru ena od njiju $\frac{3}{4}|t|$ elementov, druga pa $|t| - |t_2| - \frac{3}{4}|t| = \frac{1}{4}|t| - |t_2|$, kar je vsaj $\frac{1}{4}|t| - 1$. Razmerje $\frac{|t_1|}{|t_3|}$ je zato kvečjemu $\frac{\frac{3}{4}|t|}{\frac{1}{4}|t|-1} < 3$.)

Kako *PivotBFPR*T izračuna p ? Mediano peterke lahko najde v času $O(1)$, npr. z odločitvenim drevesom petih elementov. Ker je vseh peterk $\lceil \frac{|t|}{5} \rceil$, priprava M zahteva $\Theta(1) \cdot \lceil \frac{|t|}{5} \rceil = \Theta(|t|)$ časa. Kako pa določi mediano $\lceil \frac{|t|}{5} \rceil$ elementov tabele M ? Mediana je element, od katerega je manjših ali enakih polovica elementov tabele. Mediana p je zato $\lceil \frac{|M|}{2} \rceil$ -ti najmanjši element tabele M , kjer je $|M| = \lceil \frac{|t|}{5} \rceil$. *PivotBFPR*T mora torej poklicati proceduro *IsciBFPR* $(|M|/2, M)$, da ji ta izračuna mediano tabele M . Tu je rekurzija posredna: *IsciBFPR*T pokliče *PivotBFPR*T, ki pokliče *IsciBFPR*T itd. Algoritem *IsciBFPR*T za iskanje k -tega najmanjšega elementa se zdaj glasi

```

procedure IsciBFPR(k,t,r,s) return int;    |k-ti najmanjsi v t[r..s]
begin
  if s-r+1 < n_min then Uredi(t,r,s); return(t[r+k-1])
  else
    p := PivotBFPR(t,r,s);
    (u,v) := Razdeli(t,r,s);
    if k <= u-r then IsciBFPR(k,t,r,u-1)
    else if k <= v-r+1 then return(p)
    else IsciBFPR(k-v+r-1,t,v+1,s)
    endif
  endif
end.
```

Časovna zahtevnost algoritma *IsciBFPRT*

Analizirajmo časovno zahtevnost $T(n)$ algoritma *IsciBFPRT* za najslabši primer. Zakaj za najslabši primer? Zato, ker nas zanima, ali raba pivota, ki ga izračuna *PivotBFPRT*, izboljša najslabšo časovno zahtevnost $\Theta(n^2)$ enostavnega algoritma. Zapišimo algoritem *IsciBFPRT* pri parametrih $r = 1$ in $s = n$:

```
procedure IsciBFPRT(k,t,1,n) return int;      |k-ti najmanjsi v [1..n]
begin
    if n < n_min then Uredi(t,1,n); return(t[k])
    else
        p := PivotBFPRT(t,r,s);
        (u,v) := Razdeli(t,r,s);
        if k <= u-r then IsciBFPRT(k,t,r,u-1)
        else if k <= v-r+1 then return(p)
        else IsciBFPRT(k-v+r-1,t,v+1,s)
        endif
    endif
end.
```

Naj bo $T(n)$ najslabša časovna zahtevnost zgornjega algoritma. Poskušajmo izraziti $T(n)$ v vsakem od možnih primerov, tj. ko je $n \geq n_{\min}$ in ko je $n < n_{\min}$:

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

$$\begin{aligned}
 T(n) = \Theta(n) &\quad \text{—časovna zahtevnost računanja } M \\
 + T\left(\frac{n}{5}\right) &\quad \text{—najslabša časovna zahtevnost računanja } p \\
 + \Theta(n) &\quad \text{—časovna zahtevnost razporejanja } t \text{ v } t_1, t_2, t_3 \\
 + T(\max\{|t_1|, |t_3|\}) &\quad \text{—časovna zahtevnost iskanja po večji od } t_1, t_3
 \end{aligned}$$

Zaradi prejšnje trditve sledi $T(n) = T\left(\frac{n}{5}\right) + cn + T\left(\frac{3}{4}n\right)$, kjer je $c > 0$.

- $n < n_{\min}$. Vzemimo za n_{\min} tisto velikost, do katere je *ureditev* tabele t (velikosti $n < n_{\min}$) hitrejša od cn . Za take velikosti n je $T(n) \leq cn$. Število n_{\min} najdemo eksperimentalno; literatura navaja $n_{\min} = 50$.

Možna primera vodita do rekurzivne enačbe za najslabšo časovno zahtevnost:

$$T(n) \leq \begin{cases} cn & \text{če } n < n_{\min}; \\ T\left(\frac{n}{5}\right) + cn + T\left(\frac{3}{4}n\right) & \text{če } n \geq n_{\min}. \end{cases}$$

Zdaj moramo iz te enačbe nekako izluščiti le še to, kako hitro narašča $T(n)$.

Tu nas prijetno preseneti naslednja ugotovitev:

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

Ideja dokaza. Najprej moramo dokazati, da trditev velja za vse $n \leq n_{\min} = 50$. Nato za osnovo indukcije vzamemo trditev pri $n = n_{\min} = 50$. Veljavnost indukcijskega koraka z n na $n + 1$ preverimo pri $n \geq n_{\min} = 50$. (Podrobnosti tukaj opustim in jih prepustim za vajo.) \square

Sklep. V neurejeni tabeli t lahko k -ti najmanjši element najdemo v času $\Theta(n)$.

Opomba. Na začetku smo videli, da iskanje k -tega najmanjšega elementa v neurejeni tabeli velikosti n zahteva vsaj $\Omega(n)$ časa. Pravkar pa smo dokazali, da to iskanje zahteva največ $\mathcal{O}(n)$ časa. Ker sta spodnja in zgornja meja časovne zahtevnosti tega problema enaki, in ima izboljšani algoritmom časovno zahtevnost $\Theta(n)$, je *asimptotično optimalen*.

11. Diskretna Fourierova transformacija

Diskretna Fourierova transformacija ima pomembno vlogo v praksi. Algoritem za njen izračun bomo razvili z metodo Deli in vladaj. Pred tem pa bralca motivirajmo.

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

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

Tedaj je

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

Prodot $r(x)$ 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 izračunamo 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 $k = 0, \dots, i$. Zato za c_i , kjer je $i = 0, \dots, n + m$, velja

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

Izračun c_i zahteva $i + 1$ množenj. Izračun vseh $n + m + 1$ koeficientov c_i zato zahteva $\sum_{i=0}^{n+m} (i + 1) = \sum_{i=0}^{n+m} i + \sum_{i=0}^{n+m} 1 = \frac{(n+m)(n+m+1)}{2} + n + m + 1 = \Theta((n + m)^2)$ množenj.

Sklep. Izračun produkta $r(x) = p(x)q(x)$ zahteva kvečjemu $\mathcal{O}((n+m)^2)$ množenj.

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

Zamisel

- $p(x) = \sum_{i=0}^n a_i x^i$ lahko predstavimo v obliki (a_0, \dots, a_n) . Temu rečemo *koeficientna predstavitev* (k.p.) polinoma $p(x)$. Da je $p(x)$ predstavljen v tej obliki, zapišemo s $p(x) \stackrel{\text{k.p.}}{=} (a_0, \dots, a_n)$. Podobno je $q(x) \stackrel{\text{k.p.}}{=} (b_0, \dots, b_m)$.

Polinome pa lahko natančno opišemo tudi z njihovimi vrednostmi v izbranih številih. Npr., $p(x)$ lahko podamo v obliki (p_0, \dots, p_n) , kjer so $p_i = p(t_i)$ njegove vrednosti v $n + 1$ paroma različnih številih t_0, \dots, t_n . Temu rečemo *vrednostna predstavitev* (v.p.) polinoma $p(x)$. Da je $p(x)$ predstavljen v tej obliki, zapišemo s $p(x) \stackrel{\text{v.p.}}{=} (p_0, \dots, p_n)$. Podobno je tudi $q(x) \stackrel{\text{v.p.}}{=} (q_0, \dots, q_m)$, kjer so $q_i = q(t'_i)$ in t'_0, \dots, t'_m paroma različna števila.

Primer. Koeficientna predstavitev polinoma $p(x) = 3x^4 + 5x^2 - 2x + 1$ je $p(x) \stackrel{\text{k.p.}}{=} (1, -2, 5, 0, 3)$, vrednostna v točkah $(-2, -1, 0, 1, 2)$ pa $p(x) \stackrel{\text{v.p.}}{=} (p(-2), p(-1), p(0), p(1), p(2)) = (73, 11, 1, 7, 65)$.

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

$$p(x) \stackrel{\text{v.p.}}{=} (p_0, \dots, p_n) \quad \text{in} \quad q(x) \stackrel{\text{v.p.}}{=} (q_0, \dots, q_m),$$

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

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

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

$$r(x) \stackrel{\text{v.p.}}{=} (r_0, \dots, r_s),$$

kjer je $r_i = r(t_i) = p(t_i)q(t_i) = p_i q_i$ za $i = 0, \dots, s$.

Koliko je s , stopnja polinoma $r(x)$? Stopnji polinomov $p(x)$ in $q(x)$ sta n in m (saj imata njuni v.p. $n+1$ in $m+1$ elementov), zato je $r(x)$ stopnje

$$s = n + m.$$

Torej moramo izračunati $n + m + 1$ vrednosti r_0, \dots, r_{n+m} .

Ker je $r_i = p_i q_i$, bomo množili *istoležne* vrednosti iz (p_0, \dots, p_n) in (q_0, \dots, q_m) .

Opazimo težavo: Za izračun $n+m+1$ vrednosti r_i bi morali v.p. polinomov $p(x)$ in $q(x)$ vsebovati vsaka po $n+m+1$ vrednosti (ne pa le $n+1$ in $m+1$). Zato vsako od teh v.p. *dopolnimo* z vrednostmi polinoma v dodatnih točkah.

Dobimo $p(x) \stackrel{\text{v.p.}}{=} (p_0, \dots, p_{n+m})$ in $q(x) \stackrel{\text{v.p.}}{=} (q_0, \dots, q_{n+m})$.

Zdaj lahko izračunamo $r(x) \stackrel{\text{v.p.}}{=} (r_0, \dots, r_{n+m})$ z $n+m+1$ množenji $r_i := p_i q_i$.

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

Problem

Zmanjšanje števila množenj s $\Theta((n+m)^2)$ na $\Theta(n+m)$ smo dosegli z uporabo drugačne predstavitev polinomov $p(x)$, $q(x)$ in $r(x)$. Kaj pa če $p(x)$ in $q(x)$ ne bi bila dana v vrednostni temveč koeficientni predstavitevi? Potem bi morali njuni koeficientni predstavitvi najprej *pretvoriti* v vrednostni, kot kaže spodnja slika:

$$\begin{array}{ccc} p(x) \stackrel{\text{k.p.}}{=} (a_0, \dots, a_n) & \xrightarrow{\quad} & p(x) \stackrel{\text{v.p.}}{=} (p_0, \dots, p_{n+m}) \\ q(x) \stackrel{\text{k.p.}}{=} (b_0, \dots, b_m) & \xrightarrow{\quad} & q(x) \stackrel{\text{v.p.}}{=} (q_0, \dots, q_{n+m}) \\ & \vdots & \dots \\ & \downarrow & \downarrow \\ r(x) \stackrel{\text{v.p.}}{=} (r_0, \dots, r_{n+m}) & & \end{array}$$

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

Ocenimo, koliko množenj bi morala zahtevati pretvorba k.p. *enega* polinoma, da bi se pretvarjanje obeh izplačalo. Kadar imata polinoma isto stopnjo $n = m$, zahteva običajni postopek $\Theta((n + n)^2) = \Theta(n^2)$ množenj. Če bi pretvorba predstavitve *enega* polinoma zahtevala $\Theta(n^2)$ množenj, bi naša zamisel zahtevala $\Theta(n^2) + \Theta(n^2) + \Theta(2n)$ množenj, kar je še vedno reda $\Theta(n^2)$. Zato potrebujemo postopek, ki bo k.p. polinoma stopnje n pretvoril v v.p. z *asimptotično manj kot* $\Theta(n^2)$ množenji. Torej moramo rešiti naslednji problem:

Problem. *Kako pretvoriti k.p. polinoma stopnje n v v.p. z manj kot $\Theta(n^2)$ množenji?*

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

Primer. Naj bo $p(x) = 3x^4 + 5x^2 - 2x + 1$. Po Hornerju izračunamo vrednost $p(t)$ pri $t = 2$ takole: $p(t) = (((3t + 0)t + 5)t - 2)t + 1 = (((3 \cdot 2 + 0) \cdot 2 + 5) \cdot 2 - 2) \cdot 2 + 1 = 65$. \square

Vrednost polinoma $p(x) = a_n x^n + a_{n-1} x^{n-1} + a_{n-2} x^{n-2} + \dots + a_1 x + a_0$ pri $x = t$ izračunamo takole: $p(t) = (\dots ((a_n \cdot t + a_{n-1}) \cdot t + a_{n-2}) \cdot t + \dots a_1) \cdot t + a_0$. Izračun $p(t)$ zahteva n množenj. Ker pa je za pretvorbo k.p. $p(x)$ v v.p. treba izračunati $n + 1$ vrednosti $p(t_0), p(t_1), \dots, p(t_n)$, bi pretvorba zahtevala $n(n + 1) = \Theta(n^2)$ množenj. To pa pomeni, da *uporaba Hornerjevega postopka ne reši našega problema*.

Najti moramo algoritmom za konstrukcijo vrednostne predstavitev polinoma, ki bo asimptotično hitrejši od tega s Hornerjevim postopkom.

Ali tak algoritmom obstaja? Da. Zdaj v zgodbo vstopi *Diskretna Fourierova transformacija* (DFT), ki skupaj z metodo *Deli in vladaj* omogoči razvoj takega algoritma. Algoritmom se imenuje *Hitra Fourierova transformacija* (FFT) in ima časovno zahtevnost $\Theta(n \log n)$.

Diskretna Fourierova transformacija (DFT)

Najprej opišimo DFT v splošnem. Pri DFT igra pomembno vlogo matematična struktura imenovana vektorski prostor.

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

Linearna transformacija vektorskega prostora V vase je preslikava $T : V \rightarrow V$ z lastnostjo $T(\alpha \cdot \mathbf{u} + \beta \cdot \mathbf{v}) = \alpha \cdot T(\mathbf{u}) + \beta \cdot T(\mathbf{v})$, za poljubna vektorja \mathbf{u}, \mathbf{v} in poljubna skalarja α, β .

Linearno transformacijo vektorskega prostora končne dimenzijske d predstavimo z matriko reda $d \times d$. Kadar obstaja njena inverzna matrika, obstaja tudi inverzna transformacija.

Linearnih transformacij vektorskega prostora dimenzijske d je veliko. Med njimi nas bo zanimala t.i. *Diskretna Fourierova Transformacija* (DFT). Ključno vlogo pri DFT bo igrал *primitivni koren enote*. To je skalar ω , ki ga definirata dve lastnosti:

1. $\omega^d = 1$ — ω je *koren enote*
2. $\omega^k \neq 1$ za $k = 1, \dots, d - 1$. — ω je *primitivni koren enote*

V obsegu \mathbb{C} je $\omega = e^{\iota \frac{2\pi}{d}}$, kjer je ι *imaginarna enota*.

(Imaginarno enoto bomo označili z ι , da ne bomo zamenjevali z eksponentom i .)

Definicija. Diskretna Fourierova transformacija reda d je linearna transformacija prostora V dimenzije d , ki jo predstavlja matrika F reda $d \times d$ s komponentami

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

Inverzno DFT predstavlja matrika F^{-1} s komponentami

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

(Vaja: preverite, da je $FF^{-1} = F^{-1}F = I$.)

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 $0 \leq i, j \leq d-1$. Dobimo $\omega^{0*0} = 1$; $\omega^{0*1} = 1$; $\omega^{1*0} = 1$; $\omega^{1*1} = -1$. Torej je $F = \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}$. Izračunajmo še F^{-1} . Njene komponente so $F_{ij}^{-1} = \frac{1}{d} \omega^{-i*j}$. Ker so $\omega^{-0*0} = 1$; $\omega^{-0*1} = 1$; $\omega^{-1*0} = 1$; $\omega^{-1*1} = -1$, je $F^{-1} = \frac{1}{2} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}$.

Uporaba. Vrnimo se k pretvorbi k.p. polinoma stopnje n v v.p. v $n+1$ točkah. Spet je $p(x) = \sum_{i=0}^n a_i x^i$, njegova k.p. pa (a_0, \dots, a_n) . Ker je urejena $n+1$ -terka števil, jo lahko smatramo za vektor z $n+1$ komponentami, $\mathbf{a} = (a_0, \dots, a_n)$. Naj bo V vektorski prostor nad \mathbb{C} , ki vsebuje vse vektorje z $n+1$ komponentami. Torej je $\mathbf{a} \in V$. Izračunajmo, kam ga preslika DFT reda $d = \dim V = n+1$.

$$F\mathbf{a} = \begin{pmatrix} \omega^{0*0} & \dots & \omega^{0*j} & \dots & \omega^{0*n} \\ \vdots & & \vdots & & \vdots \\ \omega^{i*0} & \dots & \omega^{i*j} & \dots & \omega^{i*n} \\ \vdots & & \vdots & & \vdots \\ \omega^{n*0} & \dots & \omega^{n*j} & \dots & \omega^{n*n} \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ \vdots \\ a_j \\ \vdots \\ a_n \end{pmatrix} = \begin{pmatrix} \sum_{j=0}^n a_j \omega^{0*j} \\ \vdots \\ \sum_{j=0}^n a_j \omega^{i*j} \\ \vdots \\ \sum_{j=0}^n a_j \omega^{n*j} \end{pmatrix} = \begin{pmatrix} \sum_{j=0}^n a_j (\omega^0)^j \\ \vdots \\ \sum_{j=0}^n a_j (\omega^i)^j \\ \vdots \\ \sum_{j=0}^n a_j (\omega^n)^j \end{pmatrix} = \begin{pmatrix} p(\omega^0) \\ \vdots \\ p(\omega^i) \\ \vdots \\ p(\omega^n) \end{pmatrix}$$

DFT reda $n+1$ preslika k.p. polinoma $p(x)$ stopnje n v v.p. na točkah $\omega^0, \dots, \omega^n$. Podobno DFT reda $m+1$ preslika k.p. $q(x)$ stopnje m v v.p. na točkah $\omega^0, \dots, \omega^m$.

Sklep. DFT reda $\blacksquare + 1$ pretvori k.p. polinoma stopnje \blacksquare v v.p. na točkah $\omega^0, \dots, \omega^{\blacksquare}$.

Zdaj lahko dopolnimo sliko

$$\begin{array}{ccc}
 p(x) \stackrel{\text{k.p.}}{=} (a_0, \dots, a_n) & \xrightarrow{\quad \quad \quad} & p(x) \stackrel{\text{v.p.}}{=} (p_0, \dots, p_{n+m}) \\
 q(x) \stackrel{\text{k.p.}}{=} (b_0, \dots, b_m) & \xrightarrow{\quad \quad \quad} & q(x) \stackrel{\text{v.p.}}{=} (q_0, \dots, q_{n+m}) \\
 & & \vdots \quad \dots \quad \vdots \\
 & & r(x) \stackrel{\text{v.p.}}{=} (r_0, \dots, r_{n+m})
 \end{array}$$

v sliko

$$\begin{array}{ccc}
 p(x) \stackrel{\text{k.p.}}{=} (a_0, \dots, a_n, \overbrace{0, \dots, 0}^m) & \xrightarrow{\quad \quad \quad \text{DFT reda } n+m+1 \quad \quad \quad} & p(x) \stackrel{\text{v.p.}}{=} (p_0, \dots, p_{n+m}) \\
 q(x) \stackrel{\text{k.p.}}{=} (b_0, \dots, b_m, \overbrace{0, \dots, 0}^n) & \xrightarrow{\quad \quad \quad \text{DFT reda } n+m+1 \quad \quad \quad} & q(x) \stackrel{\text{v.p.}}{=} (q_0, \dots, q_{n+m}) \\
 & & \vdots \quad \dots \quad \vdots \\
 & & r(x) \stackrel{\text{v.p.}}{=} (r_0, \dots, r_{n+m})
 \end{array}$$

kjer k.p. polinoma $p(x)$ dopolnimo z m ničlami, k.p. polinoma $q(x)$ pa z n ničlami, da sta oba navidez stopnje $n+m$. Potem DFT reda $n+m+1$ vsakemu priredi njegovo v.p. na točkah $\omega^0, \dots, \omega^{n+m}$.

Primer. Naj bo $p(x) = 2x^3 + 2x^2 + x + 1$. Njegova k.p. je $\mathbf{a} = (1, 1, 2, 2)$. Kakšna je v.p. \mathbf{a}' 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

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

K.p. $\mathbf{a} = (1, 1, 2, 2)$ se preslika v v.p. polinoma $p(x)$ v točkah $(t_0, t_1, t_2, t_3) = (\omega^0, \omega^1, \omega^2, \omega^3) = (1, \iota, -1, -\iota)$ tako:

$$DFT(\mathbf{a}, 4) = F\mathbf{a} = \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & \iota & -1 & -\iota \\ 1 & -1 & 1 & -1 \\ 1 & -\iota & -1 & \iota \end{pmatrix} \begin{pmatrix} 1 \\ 1 \\ 2 \\ 2 \end{pmatrix} = \begin{pmatrix} 6 \\ -1 - \iota \\ 0 \\ -1 + \iota \end{pmatrix} = \begin{pmatrix} p(1) \\ p(\iota) \\ p(-1) \\ p(-\iota) \end{pmatrix} = \mathbf{a}'.$$

Preizkusimo še inverzno DFT, ali preslika pravkar izračunano v.p. \mathbf{a}' nazaj v k.p. \mathbf{a} . Po definiciji je

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

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

$$F^{-1}\mathbf{a}' = \frac{1}{4} \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & -\iota & -1 & \iota \\ 1 & -1 & 1 & -1 \\ 1 & \iota & -1 & -\iota \end{pmatrix} \begin{pmatrix} 6 \\ -1 - \iota \\ 0 \\ -1 + \iota \end{pmatrix} = \begin{pmatrix} 1 \\ 1 \\ 2 \\ 2 \end{pmatrix} = \mathbf{a}.$$

□

Primer. Naj bo $p(x) = 3x^4 + 5x^2 - 2x + 1$. V kaj se z DFT preslika njegova k.p. $\mathbf{a} = (1, -2, 5, 0, 3)$? Stopnja polinoma je $n = 4$, zato bo DFT reda $d = n + 1 = 5$. Primitivni koren enote je $\omega = e^{i \frac{2\pi}{d}} = e^{i \frac{2\pi}{5}}$. Fourierova matrika je

$$F = \begin{pmatrix} \omega^{0*0} & \omega^{0*1} & \omega^{0*2} & \omega^{0*3} & \omega^{0*4} \\ \omega^{1*0} & \omega^{1*1} & \omega^{1*2} & \omega^{1*3} & \omega^{1*4} \\ \omega^{2*0} & \omega^{2*1} & \omega^{2*2} & \omega^{2*3} & \omega^{2*4} \\ \omega^{3*0} & \omega^{3*1} & \omega^{3*2} & \omega^{3*3} & \omega^{3*4} \\ \omega^{4*0} & \omega^{4*1} & \omega^{4*2} & \omega^{4*3} & \omega^{4*4} \end{pmatrix} = \begin{pmatrix} 1 & 1 & 1 & 1 & 1 \\ 1 & \omega^1 & \omega^2 & \omega^3 & \omega^4 \\ 1 & \omega^2 & \omega^4 & \omega^6 & \omega^8 \\ 1 & \omega^3 & \omega^6 & \omega^9 & \omega^{12} \\ 1 & \omega^4 & \omega^8 & \omega^{12} & \omega^{16} \end{pmatrix} = \begin{pmatrix} 1 & 1 & 1 & 1 & 1 \\ 1 & \omega^1 & \omega^2 & \omega^3 & \omega^4 \\ 1 & \omega^2 & \omega^4 & \omega^1 & \omega^3 \\ 1 & \omega^3 & \omega^1 & \omega^4 & \omega^2 \\ 1 & \omega^4 & \omega^3 & \omega^2 & \omega^1 \end{pmatrix}.$$

Koeficientna predstavitev $\mathbf{a} = (1, -2, 5, 0, 3)$ se preslika v vrednostno predstavitev \mathbf{a}' polinoma v točkah $(t_0, t_1, t_2, t_3, t_4) = (\omega^0, \omega^1, \omega^2, \omega^3, \omega^4) = ((e^{i \frac{2\pi}{5}})^0, (e^{i \frac{2\pi}{5}})^1, (e^{i \frac{2\pi}{5}})^2, (e^{i \frac{2\pi}{5}})^3, (e^{i \frac{2\pi}{5}})^4) = (1, e^{i \frac{2\pi}{5}}, e^{i \frac{4\pi}{5}}, e^{i \frac{6\pi}{5}}, e^{i \frac{8\pi}{5}})$
 $= (1, e^{i \frac{2\pi}{5}}, e^{i \frac{4\pi}{5}}, e^{-i \frac{\pi}{5}}, e^{-i \frac{3\pi}{5}})$ takole:

$$F\mathbf{a} = \begin{pmatrix} 1 & 1 & 1 & 1 & 1 \\ 1 & \omega^1 & \omega^2 & \omega^3 & \omega^4 \\ 1 & \omega^2 & \omega^4 & \omega^1 & \omega^3 \\ 1 & \omega^3 & \omega^1 & \omega^4 & \omega^2 \\ 1 & \omega^4 & \omega^3 & \omega^2 & \omega^1 \end{pmatrix} \begin{pmatrix} 1 \\ -2 \\ 5 \\ 0 \\ 3 \end{pmatrix} = \begin{pmatrix} 7 \\ 1 - 2\omega + 5\omega^2 + 3\omega^4 \\ 1 - 2\omega^2 + 5\omega^4 + 3\omega^3 \\ 1 - 2\omega^3 + 5\omega + 3\omega^2 \\ 1 - 2\omega^4 + 5\omega^3 + 3\omega \end{pmatrix} = \begin{pmatrix} p(\omega^0) \\ p(\omega^1) \\ p(\omega^2) \\ p(\omega^3) \\ p(\omega^4) \end{pmatrix} = \mathbf{a}'.$$

□

I. Razdelitev problema na dva podproblema

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

1. V $p(x)$ so koeficienti s sodimi in lihimi indeksi. Prvi so $a_0, a_2, a_4, \dots, a_{n-1}$, drugi pa $a_1, a_3, a_5, \dots, a_n$. Vsakih je r . S prvimi definirajmo polinom $p_S(x)$, z drugimi pa polinom $p_L(x)$ takole:

$$p_S(x) \stackrel{\text{def}}{=} a_0 + a_2x + a_4x^2 + \cdots + a_{n-1}x^{r-1} \quad \text{oz.} \quad \mathbf{a}_S \stackrel{\text{def}}{=} (a_0, a_2, a_4, \dots, a_{n-1})$$

$$p_L(x) \stackrel{\text{def}}{=} a_1 + a_3x + a_5x^2 + \cdots + a_nx^{r-1} \quad \text{oz.} \quad \mathbf{a}_L \stackrel{\text{def}}{=} (a_1, a_3, a_5, \dots, a_n)$$

Koeficientni predstavitevi novih polinomov smo označili z \mathbf{a}_S in \mathbf{a}_L .

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

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

3. Izraz nam pove, da lahko vrednosti $p(x)$ v d točkah $x = \omega^0, \omega^1, \dots, \omega^n$ izračunamo v dveh korakih:

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

4. Hoteli smo razdeliti problem $DFT(\mathbf{a}, d)$ v dva podproblema *iste vrste* kot $DFT(\mathbf{a}, d)$. Toda računanji vrednosti $p_S(x^2)$ in $p_L(x^2)$ v koraku 3a nista videti problema iste vrste kot $DFT(\mathbf{a}, d)$. Zakaj? Medtem ko $DFT(\mathbf{a}, d)$ sprašuje po vrednostih polinoma pri d argumentih x , podproblem sprašujeta po vrednostih polinomov pri *kvadratih* teh argumentov; kaže tudi, da podproblem ne zahtevata računanja na *manj* argumentih kot zahteva $DFT(\mathbf{a}, d)$. *Ali torej nismo razdelili $DFT(\mathbf{a}, d)$ v dva manjša istorodna podproblema?* Smo. Natančnejsi razmislek nam bo odkril, kakšen je pravi pogled na podproblema.

a) Za poljuben naravni k velja $(\omega^{k+r})^2 = \omega^{2k}$.

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. \square

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

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

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

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

c) Če je ω d -ti primitivni koren enote, je ω^2 r -ti primitivni koren enote.

Dokaz. ω^2 je r -ti koren: $\omega^2 = (e^{i\frac{2\pi}{d}})^2 = e^{2i\frac{2\pi}{d}} = e^{i\frac{2\pi}{d/2}} = e^{i\frac{2\pi}{r}}$. ω^2 je primitiven: Za $k = 1, \dots, r-1$ je $(\omega^2)^k = (e^{i\frac{2\pi}{r}})^k = e^{i\frac{2\pi k}{r}} \neq 1$, za $k=r$ pa $(\omega^2)^r = (e^{i\frac{2\pi}{r}})^r = 1$. \square

d) Pišimo $\psi = \omega^2$. Zaradi 4b in 4c lahko korak 3a izrazimo takole:

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

Če predpostavimo, da je tudi $r = \frac{d}{2}$ sodo število, postane jasno, 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 pol manj argumentov. Zato oba poimenujemo z istim imenom kot osnovni problem, a drugimi parametri: $DFT(\mathbf{a}_S, r)$ in $DFT(\mathbf{a}_L, r)$.

Sklep: Če je $r = \frac{d}{2}$ sod, korak 3a sestavlja problema $DFT(\mathbf{a}_S, r)$ in $DFT(\mathbf{a}_L, r)$.

(Na začetku smo predpostavili, da je d sod, zdaj pa smo predpostavili, da je sod tudi $\frac{d}{2}$. Da bo tako vse do trivialnih podproblemov, bomo morali predpostaviti, da je d potenca števila 2.)

Zdaj smo uspeli razdeliti osnovni problem $DFT(\mathbf{a}, d)$ v dva istorodna podproblema $DFT(\mathbf{a}_S, r)$ in $DFT(\mathbf{a}_L, r)$. S tem je opravljen prvi del razvoja algoritma FFT(\mathbf{a}, d) z metodo *Deli in vladaj*. Kot že vemo, bo učinkovitost algoritma odvisna tudi od učinkovitosti, s katero bo sestavil rešitvi podproblemov $DFT(\mathbf{a}_S, r)$ in $DFT(\mathbf{a}_L, r)$ v rešitev problema $DFT(\mathbf{a}, d)$. Ta del algoritma 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

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

V naslednjih dveh korakih jih sestavimo takole:

5. Računanje vrednosti v *prvi polovici* poteka po enačbi iz koraka 2:

$$p(\omega^k) = p_S(\omega^{2k}) + \omega^k p_L(\omega^{2k}) = \underbrace{p_S(\psi^k)}_A + \omega^k \underbrace{p_L(\psi^k)}_B.$$

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

6. Računanje vrednosti v *drugi polovici* nas presenetí, ker zahteva le odštevanja:

$$\begin{aligned} p(\omega^{r+k}) &= p_S(\omega^{2(r+k)}) + \omega^{r+k} p_L(\omega^{2(r+k)}) = / \text{zaradi 4a} \\ &= p_S(\omega^{2k}) + \omega^{r+k} p_L(\omega^{2k}) = / \text{zaradi 4c} \\ &= p_S(\psi^k) + \omega^{r+k} p_L(\psi^k) = / \text{ker } \omega^{r+k} = -\omega^k \text{ (Dokaz:vaja.)} \\ &= p_S(\psi^k) - \underbrace{\omega^k p_L(\psi^k)}_C. \end{aligned}$$

Ko se bo računala vrednost $p(\omega^{r+k})$, bo C že izračunan, saj se je izračunal v koraku 5. 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 skupno zahteva $r = \frac{d}{2}$ množenj in $2r = d$ aditivnih operacij.

Algoritem FFT v psevdokodi

Zdaj lahko zapišemo algoritem FFT. Predpostavljamo, da je d potenca števila 2.
(To lahko vedno dosežemo, če vektor podaljšamo z nekaj ničelnimi komponentami.)

```
procedure FFT(a,d) return array;    | Izracunaj p = DFT(a,d)
begin
    if d=1 then return a_0
    else
        a_S := (a_0,a_2,...,a_{n-1});    | Razdeli a v a_S, a_L (1,2,3,4)
        a_L := (a_1,a_3,...,a_n);
        p_S := FFT(a_S,d/2);           | Izracunaj p_S = DFT(a_S,d/2)
        p_L := FFT(a_L,d/2);           | Izracunaj p_L = DFT(a_L,d/2)
        for k:=0 to n do             | Sestavi p iz p_S, p_L
            omega^k := exp{i*k*2*pi/d};
            p[k] := p_S[k mod d/2] + omega^k * p_L[k mod d/2]      | (5,6)
        endfor;
        return p
    endif
end.
```

Opombe. \mathbf{a}_S in \mathbf{a}_L sta tabeli $\mathbf{a}_S[0, 1, \dots, d/2-1] = \mathbf{a}_S = (a_0, a_2, \dots, a_{n-1})$ in $\mathbf{a}_L[0, 1, \dots, d/2-1] = \mathbf{a}_L = (a_1, a_3, \dots, a_n)$ k.p. polinomov p_S in p_L . Vrednostni predstavitevi polinomov p_S in p_L sta DFT (dimenzijski $\frac{d}{2} = \frac{n+1}{2}$) vektorjev \mathbf{a}_S in \mathbf{a}_L . Vrneta ju rekurzivna klica FFT v tabelah $\mathbf{p}_S = \mathbf{p}_S[0, 1, \dots, d/2-1]$ in $\mathbf{p}_L = \mathbf{p}_L[0, 1, \dots, d/2-1]$ s komponentami $\mathbf{p}_S[k] = p_S(\psi^k)$ in $\mathbf{p}_L[k] = p_L(\psi^k)$ za $k = 0, 1, \dots, \frac{d}{2} - 1$. Kot smo opisali v korakih 5 in 6, s temi komponentami izračunamo vrednosti $p(\omega^0), p(\omega^1), \dots, p(\omega^n)$, ki se shranijo v tabelo $\mathbf{p}[0, 1, \dots, n]$. Pri tem rabimo spremenljivko ω^k , ki ima vrednost $\omega^k = e^{i \frac{2\pi}{d} k}$.

Časovna zahtevnost algoritma FFT

Naj bo $T(d)$ časovna zahtevnost algoritma $\text{FFT}(\mathbf{a}, d)$. Potem je $T(\frac{d}{2})$ časovna zahtevnost vsakega od rekurzivnih klicev $\text{FFT}(\mathbf{a}_S, d/2)$ in $\text{FFT}(\mathbf{a}_L, d/2)$. Koraki 1, 3a, 4d ter 6, 7 povejo, da je skupna časovna zahtevnost procedur *Pripravi*, *Razdeli* in *Sestavi* enaka $\Theta(d)$. Zato je časovna zahtevnost algoritma $\text{FFT}(\mathbf{a}, d)$

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

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

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

Časovno zahtevnost algoritma $\text{FFT}(\mathbf{a}, d)$ smo izrazili z d , številom komponent v k.p. polinoma. Med d in stopnjo n polinoma je zveza $d = n + 1$, zato tudi za časovno zahtevnost FFT, izraženo z n , velja $T(n) = \Theta(n \log n)$.

Sklep. Algoritem FFT izračuna DFT reda d v času $\Theta(d \log d)$.

Uporaba

Razvili smo algoritem FFT, ki v času $\Theta(n \log n)$ pretvori k.p. polinoma stopnje n v njegovo v.p. na točkah $\omega^0, \dots, \omega^n$, 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 dá v.p. produkta $p(x)q(x)$.

Zdaj znamo izračunati *vrednostno predstavitev* produkta $p(x)q(x)$. Če pa potrebujemo *koeficientno predstavljen* $p(x)q(x)$, moramo uporabiti še *inverzno DFT*.

Inverzna diskretna Fourierova transformacija ...

Naj bo \mathbf{a} vektor dimenzijs d . Njegova DFT je vektor $\mathbf{a}' = F\mathbf{a}$, kjer je F matrika reda $d \times d$ s komponentami $F_{i,j} = \omega^{i*j}$ ($\omega = e^{\iota \frac{2\pi}{d}}$ pa d -ti primitivni koren enote).

Zdaj pa recimo, da bi bil znan vektor \mathbf{a}' . Kako bi izračunali originalni vektor \mathbf{a} ? Ker je $\mathbf{a}' = F\mathbf{a}$, je $F^{-1}\mathbf{a}' = F^{-1}F\mathbf{a} = I\mathbf{a} = \mathbf{a}$. Torej \mathbf{a} dobimo, če \mathbf{a}' transformiramo z F^{-1} , inverzno matriko matrike F . Pri definiciji DFT pa smo videli, da je F^{-1} matrika reda $d \times d$ s komponentami $F_{i,j}^{-1} = \frac{1}{d}\omega^{-i*j}$, torej

$$F^{-1} = \frac{1}{d} \begin{pmatrix} \omega^{-0*0} & \dots & \omega^{-0*j} & \dots & \omega^{-0*n} \\ \vdots & & \vdots & & \vdots \\ \omega^{-i*0} & \dots & \omega^{-i*j} & \dots & \omega^{-i*n} \\ \vdots & & \vdots & & \vdots \\ \omega^{-n*0} & \dots & \omega^{-n*j} & \dots & \omega^{-n*n} \end{pmatrix} = \frac{1}{d} G.$$

Sledi, da je $\mathbf{a} = F^{-1}\mathbf{a}' = \frac{1}{d}G\mathbf{a}'$. Torej \mathbf{a} dobimo tako, da \mathbf{a}' transformiramo z matriko G in dobljeni rezultat (vektor) pomnožimo z $\frac{1}{d}$.

Kakšna pa je transformacija, ki jo naredi množenje z matriko G ?

Matrika G nas močno spominja na matriko F , saj se od te razlikuje le po predznaku v eksponentih komponent. Ali je transformacija z G v kakšni zvezi s transformacijo z F ? Poglejmo. Če uvedemo $\tau := \omega^{-1}$, lahko G izrazimo takole:

$$G = (G_{i,j}) = (\omega^{-i*j}) = ((\omega^{-1})^{i*j}) = (\tau^{i*j}) = \begin{pmatrix} \tau^{0*0} & \dots & \tau^{0*j} & \dots & \tau^{0*n} \\ \vdots & & \vdots & & \vdots \\ \tau^{i*0} & \dots & \tau^{i*j} & \dots & \tau^{i*n} \\ \vdots & & \vdots & & \vdots \\ \tau^{n*0} & \dots & \tau^{n*j} & \dots & \tau^{n*n} \end{pmatrix}.$$

Zdaj vidimo: Če bi bilo število τ d-ti primitivni koren enote, bi bila tudi G Fourierova matrika, zato bi za izračun $G\mathbf{a}'$ lahko uporabili kar algoritom FFT!

Toda, ali τ je d -ti primitivni koren enote? Poglejmo.

Da bi bil τ d -ti primitivni koren enote, bi po definiciji moralo veljati dvoje:

$$1. \quad \tau^d = 1$$

— τ morabiti *koren* enote

$$2. \quad \tau^k \neq 1 \text{ za } k = 1, \dots, d-1.$$

— τ mora biti *primitivni* koren enote

Preverimo (upoštevajoč, da je $\tau = \omega^{-1}$ in ω d -ti primitivni koren enote):

$$1. \quad \tau^d = (\omega^{-1})^d = \omega^{-d} = (\omega^d)^{-1} = 1^{-1} = 1, \text{ ker } \omega^d = 1.$$

$$2. \quad \text{Za poljuben } k \in \{1, \dots, d-1\} \text{ je } \tau^k = (\omega^{-1})^k = (\omega^k)^{-1} \neq 1, \text{ ker } \omega^k \neq 1.$$

Sklep: $\tau = \omega^{-1}$ je d -ti primitivni koren enote, če je ω^{-1} d -ti primitivni koren enote.

Sledi, da je G Fourierova matrika.

Zato lahko $G\mathbf{a}'$ izračunamo z algoritmom FFT, v katerem ω nadomestimo z ω^{-1} (oz. v psevdokodi FFT nadomestimo vrstico $\text{omega}^k := \exp\{i*k*2*pi/d\}$ z vrstico $\text{omega}^k := \exp\{-i*k*2*pi/d\}$). Ker iščemo vektor $\mathbf{a} = \frac{1}{d}G\mathbf{a}'$, moramo rezultat takega FFT pomnožiti še s številom $\frac{1}{d}$.

Sklep. Inverzno DFT reda d danega vektorja izračuna prilagojeni FFT, ki ima d -ti primitivni koren enote ω zamenjan z ω^{-1} , dobljeni vektor pa pomnoži z $\frac{1}{d}$. Prilagojeni algoritem FFT izračuna inverzno DFT reda d v času $\Theta(d \log d)$.

... in njena uporaba pri množenju polinomov

Vrnimo se k sliki

$$\begin{array}{ccc}
 p(x) & \stackrel{\text{k.p.}}{=} & (a_0, \dots, a_n, \underset{m}{\overbrace{0, \dots, 0}}) \\
 q(x) & \stackrel{\text{k.p.}}{=} & (b_0, \dots, b_m, \underset{n}{\overbrace{0, \dots, 0}})
 \end{array}
 \xrightarrow[\text{DFT reda } n+m+1]{\quad} \begin{array}{c}
 p(x) \stackrel{\text{v.p.}}{=} (p_0, \dots, p_{n+m}) \\
 q(x) \stackrel{\text{v.p.}}{=} (q_0, \dots, q_{n+m}) \\
 \vdots \quad \dots \quad \vdots \\
 r(x) \stackrel{\text{v.p.}}{=} (r_0, \dots, r_{n+m})
 \end{array}$$

Istoležne komponente vrednostnih predstavitev polinomov $p(x)$ in $q(x)$ zmnožili in dobili v.p. polinoma $r(x) = p(x)q(x)$. Na spodnji sliki pa to v.p. pretvorimo z inverzno DFT v k.p. polinoma $r(x) = p(x)q(x)$, kar je bil naš cilj.

$$\begin{array}{ccc}
 p(x) & \stackrel{\text{k.p.}}{=} & (a_0, \dots, a_n, \underset{m}{\overbrace{0, \dots, 0}}) \\
 q(x) & \stackrel{\text{k.p.}}{=} & (b_0, \dots, b_m, \underset{n}{\overbrace{0, \dots, 0}})
 \end{array}
 \xrightarrow[\text{DFT reda } n+m+1]{\quad} \begin{array}{c}
 p(x) \stackrel{\text{v.p.}}{=} (p_0, \dots, p_{n+m}) \\
 q(x) \stackrel{\text{v.p.}}{=} (q_0, \dots, q_{n+m}) \\
 \vdots \quad \dots \quad \vdots \\
 r(x) \stackrel{\text{v.p.}}{=} (r_0, \dots, r_{n+m})
 \end{array}$$

$$r(x) \stackrel{\text{k.p.}}{=} (c_0, c_1, c_2, \dots, c_{n+m}) \xleftarrow[\text{DFT}^{-1} \text{ reda } n+m+1]{\quad} r(x) \stackrel{\text{v.p.}}{=} (r_0, \dots, r_{n+m})$$

Vsako DFT izračunamo s FFT v času $\Theta((n+m) \log(n+m))$. Množenje istoležnih komponent zahteva $\Theta(n+m)$ časa, inverzna DFT pa spet $\Theta((n+m) \log(n+m))$. Skupen čas je torej reda $\Theta((n+m) \log(n+m))$.

Sklep. Polinoma stopenj n in m lahko zmnožimo v času $\Theta((n+m) \log(n+m))$.

V.

Dinamično programiranje



12. Fibonaccijeva števila

Fibonaccijevo število F_n je definirano na rekurziven način takole:

$$F_n \stackrel{\text{def}}{=} \begin{cases} 0 & \text{če } n = 0; \\ 1 & \text{če } n = 1; \\ F_{n-1} + F_{n-2} & \text{če } n \geq 2. \end{cases}$$

Torej sta prvi dve števili 0 in 1, vsako drugo pa vsota dveh neposrednih predhodnikov.
Tule je prvih šestnajst Fibonaccijevih števil:

F_0	F_1	F_2	F_3	F_4	F_5	F_6	F_7	F_8	F_9	F_{10}	F_{11}	F_{12}	F_{13}	F_{14}	F_{15}
0	1	1	2	3	5	8	13	21	34	55	89	144	233	377	610

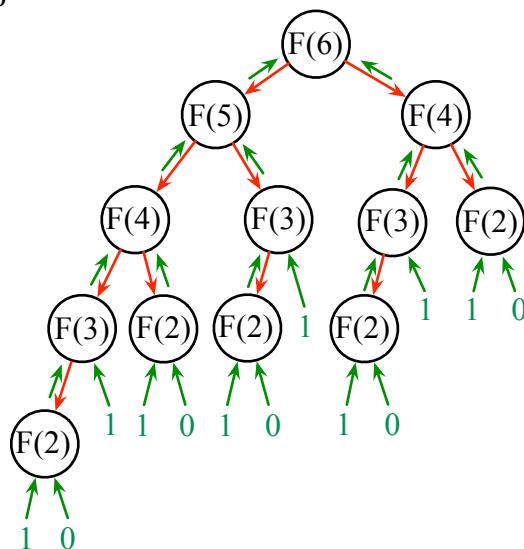
Problem: Za dani $n \in \mathbb{N}$ izračunaј F_n .

Rekurzivna definicija števila F_n nam takoj ponudi očiten algoritmom za izračun F_n , ki bo rekurzivno deloval po načelu Deli in vladaj:

```

procedure F(n) return int;
begin
    if n=0 then return 0 else
    if n=1 then return 1 else
        return(F(n-1) + F(n-2))
    endif endif
end.

```



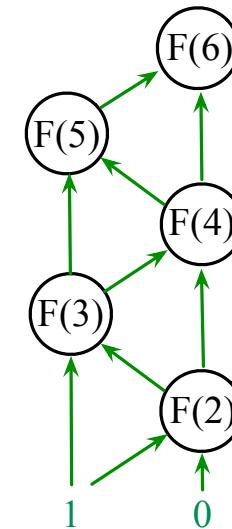
Izvajanje zgornjega algoritma pri $n = 6$ kaže drevo. Vozlišča so izvedbe procedur $F(i)$, rdeče povezave kljiči procedure F , zelene povezave pa posredovanja rezultatov kličočim proceduram F .

Izvedba pri $n=6$ razkrije hibo zgornjega algoritma: veliko izračunov se ponovi! Npr. $F(4)$ se izračuna 2-krat, $F(3)$ 3-krat, $F(2)$ pa 5-krat. Splošno: pri računanju F_n se $F(k)$, $2 \leq k \leq n$, sproži $n - k + 1$ -krat. Če je $T(k)$ časovna zahtevnost izvedbe $F(k)$, se zaradi vseh ponovitev porabi vsaj $\sum_{k=2}^n (n - k)T(k) = \Omega(n^2)$ časa (v primeru, ko bi bil $T(k) = \mathcal{O}(1)$).

Opomba. V zgornjem drevesu sta nalogi „Izračunaj F_5 ” (v levem poddrevesu) in „Izračunaj F_4 ” (v desnem poddrevesu) *ločeni*, a kljub temu zahtevata rešitev *istih* nalog (tj. „Izračunaj F_3 ” in „Izračunaj F_2 ”). To je vzrok za ponavljanje izračunov.

Algoritem tipa Deli in vladaj zaradi zgornje hibe ni učinkovit. Hočemo boljšega. Rekurzivna enačba za F_n nam odkrije še en očiten postopek: računanje $F(n)$ naj poteka „od spodaj navzgor“, od F_0 in F_1 čez vse vmesne F_i do rezultata F_n .

```
procedure F(n) return int;
begin
    b:=1; c:=0;
    if n=0 then return c else
    if n=1 then return b else
        for i:=2 to n do
            a:=b+c; c:=b; b:=a
        endfor;
    endif endif
    return(a)
end.
```



Izvajanje novega algoritma pri $n = 6$. Vsako vozlišče sešteje rezultata dveh predhodnikov.

Algoritem vedno sešteje zadnji Fibonaccijevi števili in zahteva $n - 1 = \Theta(n)$ korakov.

13. Metoda dinamičnega progamiranja

Pri razvoju algoritmov se rabi tudi metoda, imenovana *Dinamično programiranje*.

Bistvo metode je naslednje. *Nalogo problema P začnemo reševati tako, da najprej rešimo vse naloge tega problema, ki so trivialno velike (in zato trivialno rešljive), potem pa za poljubno netrivialno velikost n pokažemo, da se rešitev poljubne naloge te velikosti dá sestaviti iz rešitev nalog manjših velikosti (če so te naloge že rešene).*

Torej reševanje poteka „od spodaj navzgor”, od manjših nalog proti večjim.

Ta metoda je posebej primerna, če ima reševani problem naslednje lastnosti:

- **Enostavnost podnalog.** Nalogo problema lahko sorazmerno enostavno razdelimo v istorodne podnaloge, te pa enostavno opišemo z indeksi (npr. i, j).
- **Optimalnost rešitev.** Optimalno rešitev naloge lahko sorazmerno enostavno dobimo iz optimalnih rešitev njenih podnalog (npr. z min/max, \pm).

Pri tem se pogosto opremo še na *načelo optimalnosti*. Kaj pravi to načelo? Naj bo D_1, \dots, D_m zaporedje odločitev, ki jih algoritom sprejme, ko rešuje dano nalogu N . Pravimo, da je to zaporedje *optimalno*, če vodi do *optimalne rešitve* naloge N . Načelo optimalnosti pa pravi: *Vsako podzaporedje $D_i, D_{i+1}, \dots, D_{j-1}, D_j$ optimalnega zaporedja D_1, \dots, D_m je tudi optimalno.* To pomeni, da $D_i, D_{i+1}, \dots, D_{j-1}, D_j$ vodi od enega vmesnega rezultata do drugega optimalnega vmesnega rezultata.

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 najkrajša (optimalna) pot med svojima krajiščema: $A \rightsquigarrow C$ najkrajša iz A v C in $C \rightsquigarrow B$ najkrajša iz C v B . (Dokaz. "Če to ne bi bilo res, bi bila najkrajša neka druga etapa, denimo $A \rightarrow C$. Toda tedaj $A \rightsquigarrow B$ ne bi bila najkrajša pot iz A v B , čeprav smo predpostavili, da je!)

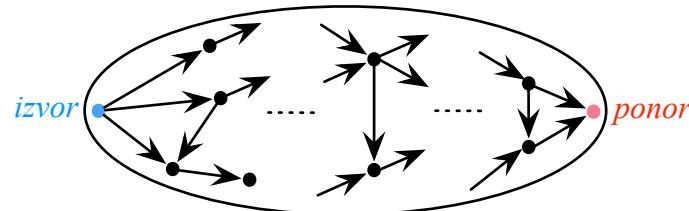
- **Skupne podnaloge.** Tudi če so podnaloge videti ločene, je včasih pri računanju njihovih opt. rešitev treba rešiti tudi kake skupne podnaloge.

Pri sestavljanju rešitve naloge velikosti n se včasih rabijo le rešitve nalog velikosti $n-1$ (tj. zadnja *generacija* rešitev), včasih pa tudi rešitve nalog še manjših velikosti, tja do neke najmanjše velikosti $n-k$. (Pri Fibonaccijevih številah je bil $k = 2$.) Očitno moramo rešitve nalog zadnjih k generacij začasno hraniti, dokler jih sestavljanje novih rešitev zagotovo ne bo več potrebovalo. To vpliva na porabo pomnilnika in prostorsko zahtevnost algoritma, ki je bil razvit po metodi dinamičnega programiranja.

14. Največji pretok

Motivacija

Dano je *omrežje*, ki ga sestavljajo vozlišča in povezave med njimi. V omrežju sta dve odlikovani vozlišči, *izvor* in *ponor*. Iz izvora priteka 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 povezava še zmore. Zato obstaja neki *največji pretok* dobrine od izvora do ponora takega omrežja. Problem, ki nas bo zanimal je, da za dano omrežje izračunamo največji pretok.



Definicija problema

Omrežje je označen usmerjen graf $G(V, A, c)$. $V = \{1, 2, \dots, n\}$ je množica vozlišč, $A \subseteq V \times V$ je množica usmerjenih povezav med vozlišči in $c : A \rightarrow \mathbb{R}_0^+$ funkcija, ki vsaki povezavi $(i, j) \in A$ privedi njeno kapaciteto $c_{i,j} \geq 0$. Vozlišče 1 je izvor, vozlišče n pa ponor omrežja. Iz izvora priteka dobrina s hitrostjo v [$\frac{\text{enota}}{\text{s}}$], 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) $0 \leq v_{i,j} \leq c_{i,j}$... pretok čez povezavo je med 0 in kapaciteto povezave;

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

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 algorithm

Najprej se lotimo reševanja problema s pojmom *prereza* grafa, ki ga je tudi sicer koristno poznati. Izkazalo se bo, da nas to vodi do algoritma z eksponentno časovno zahtevnostjo, kar je preveč.

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 ,

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

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

V splošnem ima G več $(1, n)$ -prerezov, zato bo tisti, ki ima najmanjšo kapaciteto, določal največi pretok v^* skozi G . Ugotovili smo, *kaj* je rešitev v^* :

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

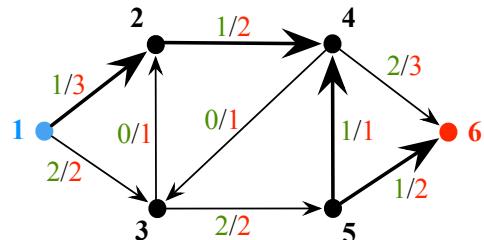
Poglejmo še, *kako* bi izračunali v^* iz te enačbe.

Izračunati moramo 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} . (Izpeljite.) Zato je časovna zahtevnost takega računanja v^* reda $\Theta(2^n)$, torej eksponentna!

Poiskati bo treba hitrejši algoritmom.

Ford-Fulkersonov algoritmem

Zamisel opišimo na primeru omrežja na spodnji sliki.



Temeljna pot iz izvora v ponor nima ciklov, smeri povezav na njej pa so zanemarjene. Odebeljene povezave sestavljajo temeljno pot. Vsaka povezava (i, j) ima oznako $v_{i,j}/c_{i,j}$.

Izberimo *temeljno pot* P , npr. $P \equiv 1 \xrightarrow{1/3} 2 \xrightarrow{1/2} 4 \xleftarrow{1/1} 5 \xrightarrow{1/2} 6$ in poskušajmo povečati pretok čez njo. Pri tem zanemarimo vsa vozlišča in povezave izven P .

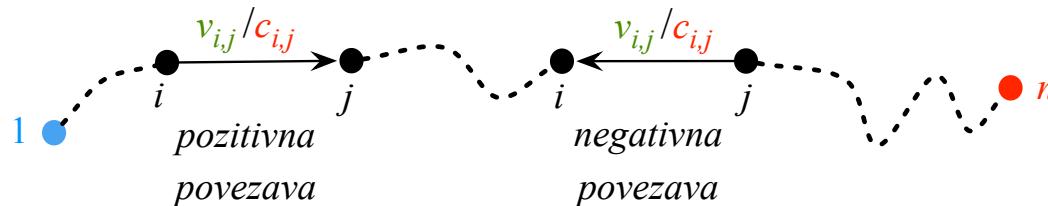
Povečajmo pretok $v_{1,2} = 1$ na 2. (To se da, ker je $c_{1,2} = 3$.) Ker zdaj v 2 doteka za 1 večji tok, bi moral iz 2 odtekati za 1 večji tok. Zato moramo povečati $v_{2,4} = 1$ na 2. (To se da, ker je $c_{2,4} = 2$.) Zdaj v 4 doteka za 1 večji tok, zato bi moral iz 4 proti 5 odtekati za 1 večji tok. Toda trenutni tok med 4 in 5 je usmerjen proti 4. Zato pretok $v_{5,4} = 1$ zmanjšamo za 1 na 0. (To smemo, saj je pretok lahko 0.) Ker zdaj iz 5 proti 4 teče za 1 manjši tok, moramo za 1 povečati tok iz 5 proti 6, tj. povečati $v_{5,6} = 1$ na 2. (To se da, ker je $c_{5,6} = 2$.) Ker je 6 ponor, nam je uspelo povečati pretok čez P za 1.

Z novimi pretoki v povezavah je $P \equiv 1 \xrightarrow{2/3} 2 \xrightarrow{2/2} 4 \xleftarrow{0/1} 5 \xrightarrow{2/2} 6$.

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, 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 odtekanje v vozlišče (a ne pod 0). Ker smo lahko vse spremembe uresničili, smo pretok čez P uspeli povečati.

Opišimo zamisel bolj natančno in splošno. Najprej nekaj definicij.

Predpostavimo, da je v omrežju $G(V, A, c)$ vsako vozlišče na neki temeljni poti (tako je v realnih 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 poti P kaže v smeri od izvora proti ponoru; sicer je (i, j) na poti P *negativna*.



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.

Množico vseh pozitivnih povezav na poti P označimo s P^+ , množico vseh negativnih povezav na poti 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.

Sledi, da pretoka čez zasičeno temeljno pot P ne moremo povečati (saj je pretok v vsaj eni pozitivni povezavi na P dosegel kapaciteto povezave, ali pa je v vsaj eni negativni povezavi na P padel na 0.) Povedano drugače: 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 $v_{i,j} > 0$.

Zamisel algoritma

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 zasivil vse in se ustavil. Zamisel pa takoj sproži tudi nekaj praktičnih vprašanj:

1. Ali je pretok čez $G(V, A, v)$ 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?

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

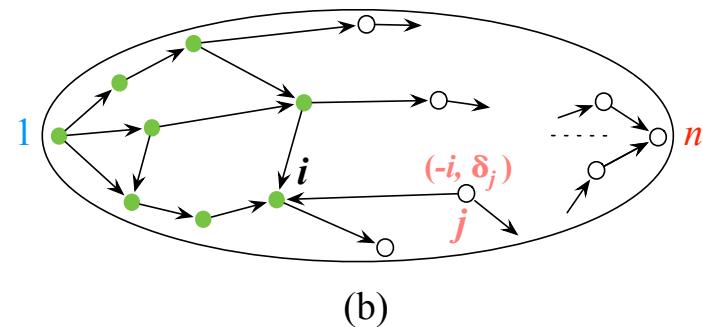
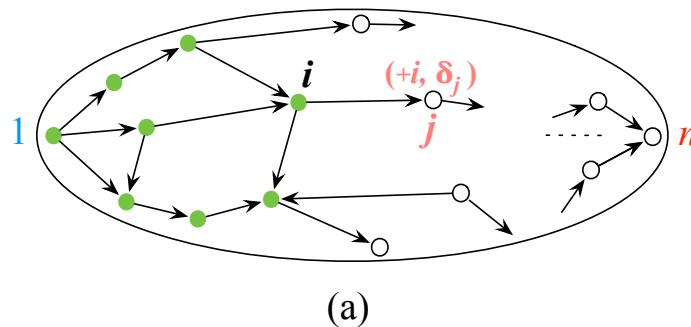
$$\min \left\{ \min_{(i,j) \in P^+} \{c_{i,j} - v_{i,j}\}, \min_{(i,j) \in P^-} \{v_{i,j}\} \right\}. \quad (*)$$

V nadaljevanju odgovorimo hkrati na vprašanji 2 in 3.

Zamislimo si metodo, s katero bomo poiskali in zasitili neko nezasičeno temeljno pot v $G(V, A, c)$. Iskanje naj se 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. Označevanje naj bo tako: *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:

- *Oznake vozlišč.* 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 $(-i, \delta_j)$.



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 oznak.* 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 $1 \rightarrow \dots \rightarrow i \rightarrow j$ povečati za največ δ_j . Zato to komponento definiramo kot

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

S tako definiranim δ_j lahko med prodiranjem skozi G sproti računamo vrednost izraza (*). (Vidimo, da mora izvor dobiti oznako (\uparrow, ∞) .)

- *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 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:
 - izberi nepregledano vozlišče i ;
 - označi kakega neoznačenega soseda j (kot je opisano zgoraj);
 - razglasji j za označenega in nepregledanega;
 - razglasji 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 dá zasiliti, 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 $P \equiv 1 \rightsquigarrow j, n$. (Če je $(j, n) \in A$, je $P \equiv 1 \rightsquigarrow j \rightarrow n$, če pa je $(n, j) \in A$, je $P \equiv 1 \rightsquigarrow j \leftarrow n$.) Tudi vozlišče j ima oznako, ki je bodisi $(+i, \delta_j)$ ali $(-i, \delta_j)$, kjer je $i \in V$. Zato je $P \equiv 1 \rightsquigarrow i, j, n$. Tako enega za drugim izsledimo še vsa ostala vozlišča na poti P in 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 , odkrito v prejšnji 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 vsaki izsleditvi prejšnjega vozlišča.)
- *Ponovitev.* Ko zasitimo rekonstruirano temeljno pot P , se pretok čez omrežje poveča za δ_n . Kljub temu lahko obstaja 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. ko označevanje vrne $\delta_n = 0$.

Časovna zahtevnost

Naj bo v^* največji pretok čez $G(V, A, c)$. Da Ford-Fulkersonov algoritmom izračuna v^* , mora zasiliti kvečjemu v^* temeljnih poti, saj vsako zasičenje temeljne poti poveča pretok čez $G(V, A, c)$ za vsaj 1. Vsaka temeljna pot ima kvečjemu $|A|$ povezav, zato označitev, izsleditev in zasičenje temeljne poti zahtevajo $\mathcal{O}(|A|)$ časa. Sledi, da je časovna zahtevnost Ford-Fulkersonovega algoritma reda

$$\mathcal{O}(v^*|A|).$$

Toda ni nam vč, 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 algoritmom ima časovno zahtevnost

$$\mathcal{O}(|V|\cdot|A|^2).$$

Izboljšani algoritmi

Problem največjega pretoka je pomemben, zato je bil in je še vedno predmet mnogih raziskav. Tule je seznam izboljšav Ford-Fulkersonovega algoritma in drugih algoritmov za ta problem, skupaj z njihovimi časovnimi zahtevnostmi v odvisnosti od $n = |V|$ in $m = |A|$ (posebej je $U \approx v^*$):

$\mathcal{O}(v^*m)$	1956	Ford, Fulkerson
$\mathcal{O}(m^2n)$	1969	Edmonds, Karp
$\mathcal{O}(mn^2)$	1970	Dinitz
$\mathcal{O}(n^3)$	1973	Karzanov
$\mathcal{O}(\sqrt{mn}^2)$	1976	Cherkassky
$\mathcal{O}(n^3)$	1978	Malhotra, Kumar, Maheshwari
$\mathcal{O}(m^{2/3}n^{5/3})$	1978	Galil
$\mathcal{O}(mn \log^2 n)$	1979	Galil, Naamad
$\mathcal{O}(mn \log n)$	1980	Sleator, Tarjan
$\mathcal{O}(mn \log(n^2/m))$	1985	Goldberg, Tarjan
$\mathcal{O}(mn \log_{\frac{m}{n \log n}} n)$	1994	King, Rao, Tarjan
$\mathcal{O}(m \min(n^{2/3}, m^{1/2}) \log \frac{n^2}{m} \log U)$	1998	Goldberg, Rao
$\mathcal{O}(mn)$	2013	Orlin

15. Nahrbtnik

Problem: Dana je množica $R = \{1, 2, \dots, n\}$ z elementi, ki predstavljajo neke *reči*, ter funkciji $v : R \rightarrow \mathbb{N}$ in $t : R \rightarrow \mathbb{N}$, ki vsaki reči i privedita *vrednost* $v(i)$ in *težo* $t(i)$. (Krajše: $v_i := v(i)$ in $t_i := t(i)$.) Dano je tudi število $a \in \mathbb{N}$, ki predstavlja *nosilnost nahrbtnika*. Naloga je naslednja:

Poišči podmnožico $P \subseteq R$, imenovano *plen*, za katero bo veljalo

$$\sum_{i \in P} t_i \leq a \quad (\textit{plen ni pretežek})$$

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 $v \in \{0, 1, \dots, V\}$. (Izbrani v je vrednost, ki je smiselna za pomnožice $P \subseteq R$. števila, ki so manjša od 0 ali večja od V ali necela, niso vrednosti kake $P \subseteq R$.)

Naj bo $i \in R$ poljubna reč. Definirajmo R_i kot množico *prvih i reči* množice R , tj. $R_i = \{1, \dots, i\}$. Množica R_i ima svoje podmnožice, vsaka od njih pa svojo vrednost in težo. Osredotočimo se na tiste podmnožice množice R_i , ki so težke kvečjemu a . Lahko se zgodi, da med njimi ni nobene, ki bi bila vredna v (ki je bil izbran zgoraj). Seveda pa se lahko tudi zgodi, da je med njimi *vsaj ena*, ki je vredna v – in tedaj je (vsaj) ena med njimi najlažja. To podmnožico označimo z $N_i(v)$. Strogo jo definiramo takole:

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

Kako naj uporabimo to nenavadno množico pri reševanju problema nahrbtnika?

Ideja je tale: po vrsti računaj množice $N_n(V)$, $N_n(V-1)$, $N_n(V-2)$, ... in končaj pri prvi, ki je definirana. Če je to množica $N_n(V-m)$, kjer je $0 \leq m \leq V$, velja $N_n(V)\uparrow$, $N_n(V-1)\uparrow$, $N_n(V-2)\uparrow$, ..., $N_n(V-m+1)\uparrow$ 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 je, kako izračunati množico $N_n(\cdot)$ pri danem argumentu \cdot .

Smiselno bi bilo, da poskušamo izraziti $N_i(\cdot)$ z eno ali več „manjšimi“ množicami enake vrste, tj. množicami $N_{i-1}(\cdot), N_{i-2}(\cdot), \dots, N_{i-k}(\cdot)$ za neki k . Ta zamisel računanja „od zgoraj navzdol“ bi nas vodila k rekurzivnemu algoritmu vrste *deli in vladaj* za izračun množic $N_n(\cdot)$.

Računanja množic $N_n(\cdot)$ pa bi se lahko lotili tudi od „spodaj navzgor“, tj. tako, da bi iz že izračunanih množic $N_{i-1}(\cdot), N_{i-2}(\cdot), \dots, N_{i-k}(\cdot)$ (za neki k) računali množice $N_i(\cdot)$. To bi bilo računanje množic $N_n(\cdot)$ po metodi *dinamičnega programiranja*. Prav to bomo uporabili v naslednjem razdelku (tam bo $k = 1$).

Pri obeh metodah moramo odkriti zvezo (*Kakšno?*) – če ta sploh obstaja – med rešitvijo naloge velikosti i in rešitvami nekaterih (*Katerih?*) nalog velikosti $i - 1, \dots, i - k$ za neki (*Kateri?*) k ($1 \leq k \leq i$).

Prav to je pogosto težko, ker zahteva kreativnost, inspiracijo, uvid in eksaktно dedukcijo, torej *naravno* inteligenco.

Algoritem z dinamičnim programiranjem

Naš cilj je izraziti množico $N_i(v)$ z eno ali več množicami $N_{i-1}(\cdot)$.

Če nam bo uspelo, bomo izrazili tudi težo $T_i(v)$ množice $N_i(v)$ s težami $T_{i-1}(\cdot)$.

Cilj bomo poskušali doseči na induktiven način: najprej bomo raziskali trivialne množice $N_1(\cdot)$ in njihove teže $T_1(\cdot)$, potem pa razmišljali, kakšna bi utegnila biti zveza med $N_i(v)$ in množicami $N_{i-1}(\cdot)$.

Če bomo kako zvezo našli, nas bo morda vodila še do zveze med $T_i(v)$ in $T_{i-1}(\cdot)$.

Zdaj pa začnimo:

$$i = 1$$

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:

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

Njihove teže so

$$\begin{array}{l} T_1(0) = 0; \\ T_1(v_1) = t_1; \\ \text{za } v \neq 0, v_1 \text{ je } T_1(v) \uparrow. \end{array}$$

$$i \geq 2$$

Zdaj je $R_i = \{1, \dots, i\}$. Ko bo $N_i(v)$ izračunana, bo bodisi vsebovala element i ali pa ga ne bo vsebovala. Poglejmo vsako možnost podrobno.

$$i \in N_i(v)$$

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

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

Teža te množice bo

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

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

$$i \notin N_i(v)$$

V tem primeru bo očitno

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

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

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

Po tem razmisleku se algoritom za reševanje problema NAHRBTNIK glasi:

```
procedure Nahrbtnik(R,t,v,a) return P;
begin
    Izracunaj V;
    for v := 0 to V do                      //trivialne naloge
        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                      //netrivialne naloge
        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) <= a
                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 + 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 $\mathcal{O}(1)$. Telo dvojne zanke zahteva $\mathcal{O}(1)$ operacij, izvede pa se $(n - 1)(V + 1)$ -krat. Zato je časovna zahtevnost celega algoritma $\mathcal{O}(nV)$.

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 funkcija *dolžine* vhodnih podatkov (tj. velikosti prostora zanje), ne pa njihove *velikosti* (magnitude). V izrazu $\mathcal{O}(nV)$ je V velikost vhodnih podatkov. Če V izrazimo z njegovo dolžino $d = \lceil \log V \rceil$, je časovna zahtevnost algoritma $\mathcal{O}(n2^d)$, torej *eksponentno* odvisna od dolžine vhodnih podatkov.

16. Najcenejše poti iz izhodišča

Definicija problema

Dan je utežen usmerjen graf $G(V, A, c)$, kjer je $V = \{1, 2, \dots, n\}$ množica vozlišč, $A \subseteq V \times V$ množica usmerjenih povezav med vozlišči in $c : V \times V \rightarrow \mathbb{R} \cup \{\infty\}$ 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 $c_{i,j} = \infty \iff (i, j) \notin A$. Usmerjena pot iz vozlišča $i_z \in V$ v $i_k \in V$ je zaporedje vozlišč i_1, i_2, \dots, i_ℓ , $\ell \geq 2$, kjer je $i_1 = i_z$ in $i_\ell = i_k$ ter $(i_j, i_{j+1}) \in A$ za $j = 1, \dots, \ell - 1$. Cena u_{i_z, i_k} te poti je vsota cen njenih povezav,

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

Cikel je usmerjena pot iz i_z v i_k , ki se konča v začetnem vozlišču, torej $i_z = i_k$. 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 njeni 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.

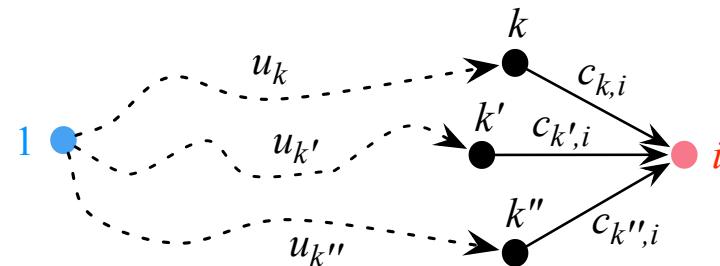
Zakaj? Naj bo $i_z \rightsquigarrow i_k$ ($i_k = i_z$) negativen cikel v G in $c_{i_z, i_k} \in \mathbb{R}^-$ njegova cena. Kakšna je najcenejša pot iz 1 v vozlišče i_z ? Denimo, da je to pot $1 = i_1, i_2, \dots, i_z$ s ceno u_{1, i_z} . Če bi to pot podaljšali z negativnim ciklom $i_z \rightsquigarrow i_k (= i_z)$, 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 $1 = i_1, i_2, \dots, i_z$ 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 predpostavljeni, da (1) v G obstaja usmerjena pot iz izhodišča 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 $1 \rightsquigarrow k \rightarrow i$, 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 $k \rightarrow i$, 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'' , ... 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

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



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

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

$$u_1 = 0$$

$$u_2 = \min_{\substack{k \\ (k,2) \in A}} \{u_k + c_{k,2}\}$$

$$u_3 = \min_{\substack{k \\ (k,3) \in A}} \{u_k + c_{k,3}\}$$

⋮

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

⋮

$$u_n = \min_{\substack{k \\ (k,n) \in A}} \{u_k + c_{k,n}\}$$

oz. krajše

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

BE

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

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

Ideja dokaza. Ugotoviti moramo, kaj v grafu $G(V, A, c)$ pomenijo u_1, u_2, \dots, u_n , ki so rešitev 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, \dots, u_n uspe (i) povezati vozlišča V grafa G v drevo T s korenom v 1 in povezavami $k \rightarrow i$, 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, \dots, u_n edina rešitev BE. \square

Posledica obeh ugotovitev je pomembna:

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

Problem *Najcenejših poti iz izhodišča* smo prevedli na problem *Reševanje sistema BE*. Praktično to pomeni, da lahko reševanje *Problema najcenejših poti iz izhodišča* nadomestimo z reševanjem sistema Bellmanovih enačb. Zato bomo našo pozornost preusmerili na vprašanje, kako rešiti sistem Bellmanovih enačb.

Reševanje sistema BE Bellmanovih enačb

Oglejmo si spet sistem Bellmanovih enačb

$$u_1 = 0$$

$$u_2 = \min_{\substack{k \\ (k,2) \in A}} \{u_k + c_{k,2}\}$$

$$u_3 = \min_{\substack{k \\ (k,3) \in A}} \{u_k + c_{k,3}\}$$

⋮

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

⋮

$$u_n = \min_{\substack{k \\ (k,n) \in A}} \{u_k + c_{k,n}\}$$

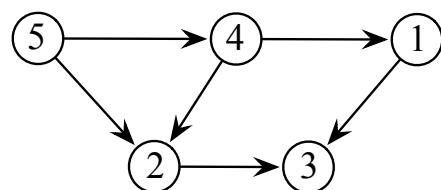
Kako naj ga rešimo? Enačba $u_i = \min_{k,(k,i) \in A} \{u_k + c_{k,i}\}$ razkriva, da za izračun u_i potrebujemo nekatere že izračunane u_k . Zato bi bila za reševanje sistema primerna metoda *dinamičnega programiranja*.

V nadaljevanju bomo obravnavali družine sistemov BE (oz. grafov $G(V, A, c)$), kjer je ta metoda posebno primerna.

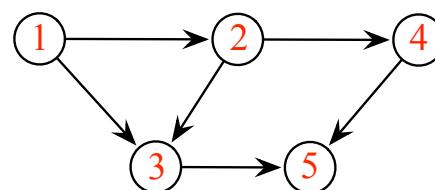
Prva taka družina sistemov BE pripada *acikličnim* grafom $G(V, A, c)$. Preden si jo bomo ogledali, moramo na kratko zaviti z glavne poti in spoznati, kaj je *topološko urejanje* grafov.

Topološko urejanje grafov

Naj bo $G(V, A)$ graf z množico vozlišč $V = \{1, 2, \dots, n\}$. Vozlišča želimo preimenovati tako, da bo po preimenovanju vsaka povezava tekla iz vozlišča z manjšim imenom v vozlišče z večjim imenom. Preimenovanje vozlišč naj opravi bijektivna funkcija $\tau : V \rightarrow V$, ki vsakemu $i \in V$ pripredi novo ime $\tau(i) \in V$, 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.



i	$\tau(i)$
1	4
2	3
3	5
4	2
5	1



Graf je topološko urejen s $\tau(i)$.

Nekateri grafi imajo več različnih topoloških ureditev. Npr., graf na prejšnji sliki topološko ureja tudi funkcija $\tau = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 3 & 4 & 5 & 2 & 1 \end{pmatrix}$. Vendar pa obstajajo grafi, ki jih ni mogoče topološko urediti. Tak je, na primer, usmerjeni cikel $3 \rightarrow 1 \rightarrow 2 \rightarrow 3$. *Kateri grafi se dajo topološko urediti in kateri ne?* Odgovor dá 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 $2, 3, \dots, n$). \square

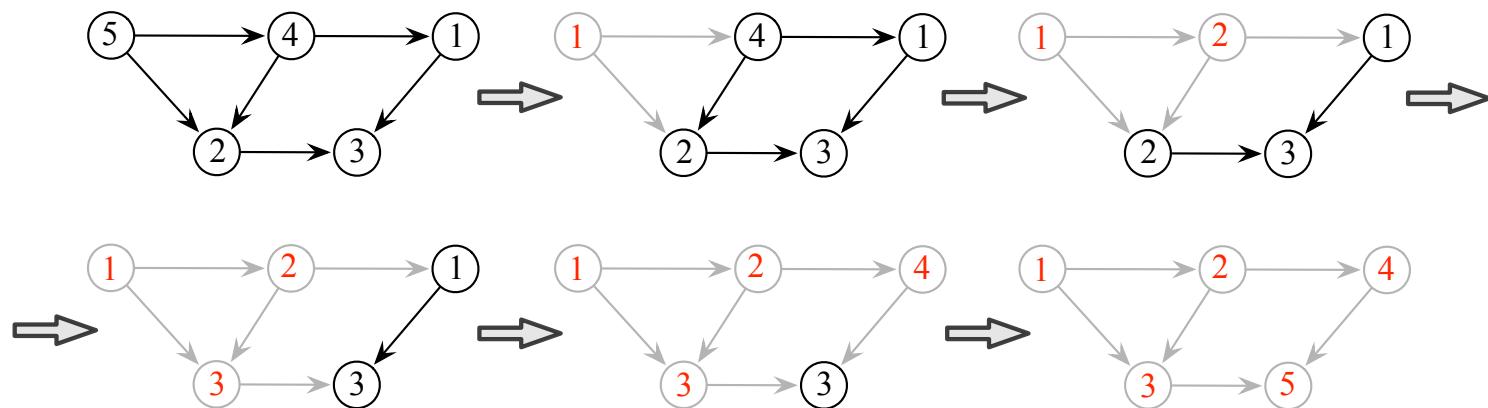
Drugi del dokaza je *konstruktiven*, kar pomeni, da opiše algoritom za topološko ureditev poljubnega acikličnega grafa. Zapišimo ta algoritom v psevdokodi:

```
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++;
        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.

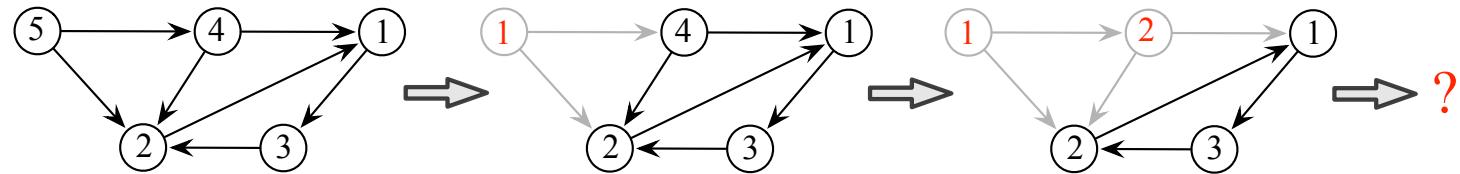
Časovno zahtevnost algoritma narekuje zanka `while`. Vsaka izvedba njenega telesa je hitrejša od prejšnje, ker se število vozlišč, ki jih je treba pregledati in med njimi eno izbrati, vsakokrat zmanjša za 1. Ker k -ta izvedba zahteva $\mathcal{O}(|V| - k)$ časa, vse zahtevajo $c(|V| + |V| - 1 + \dots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

Primer. Poglejmo, kako poteka topološko urejanje grafa s prejšnje slike.



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.



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

Če je graf *acikličen*, mu lahko pridružimo ustrezen sistem Belmannovih enačb, ki je primeren za reševanje z metodo dinamičnega programiranja. To dosežemo tako, da graf prej 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 je v $G(V, A, c)$ povezava $i \rightarrow j$, potem je $i < j$. To vpliva tudi na Bellmanove enačbe, kjer se pogoj $(k, i) \in A$ nadomesti s pogojem $k < i$, tako da se enačbe 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, \dots, u_n tega sistema BE teče po naraščajočem indeksu i : ko so izračunani u_1, \dots, u_{i-1} , lahko začnemo računati u_i :

$$u_1 = 0$$

$$u_2 = u_1 + c_{1,2}$$

$$u_3 = \min\{u_1 + c_{1,3}, u_2 + c_{2,3}\}$$

$$u_4 = \min\{u_1 + c_{1,4}, u_2 + c_{2,4}, u_3 + c_{3,4}\}$$

$$u_5 = \min\{u_1 + c_{1,5}, u_2 + c_{2,5}, u_3 + c_{3,5}, u_4 + c_{4,5}\}$$

⋮

$$u_i = \min\{u_1 + c_{1,i}, u_2 + c_{2,i}, u_3 + c_{3,i}, \dots, u_{i-1} + c_{i-1,i}\}$$

⋮

$$u_n = \min\{u_1 + c_{1,n}, u_2 + c_{2,n}, u_3 + c_{3,n}, u_4 + c_{4,n}, \dots, u_{n-1} + c_{n-1,n}\}$$

Algoritem za tak izračun u_1, u_2, \dots, u_n je zdaj enostaven (vaja!).

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

Sklep. Časovna zahtevnost računanja najcenejših poti iz izhodišča v acikličnem grafu $G(V, A, c)$ je $\mathcal{O}(|V|^2)$.

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, \dots, u_n ne bomo naslonili na sistem BE pač pa opisali algoritmom, ki ga je zasnoval Dijkstra. Dijkstra je razmišljal nekako takole:

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

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

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 k vodila kaka cenejša pot, ki bi šla čez vsaj eno vozlišče $\ell \in Z$ (Slika a). Njena cena bi bila $u_\ell + c(\ell \rightsquigarrow k) < u_k$. Ker so cene povezav pozitivne, bi bila $c(\ell \rightsquigarrow k) > 0$ in zato $u_\ell < u_k$. To bi bilo protislovno, saj je u_k po predpostavki najmanjša začasna cena.

Ker je cena u_k dokončna, mora k prestopiti iz Z v D . Če zaradi prestopa postane $Z = \emptyset$, se algoritmom konča, saj so vse cene u_1, u_2, \dots, u_n dokončne. Algoritom torej dopolnimo z ukazi

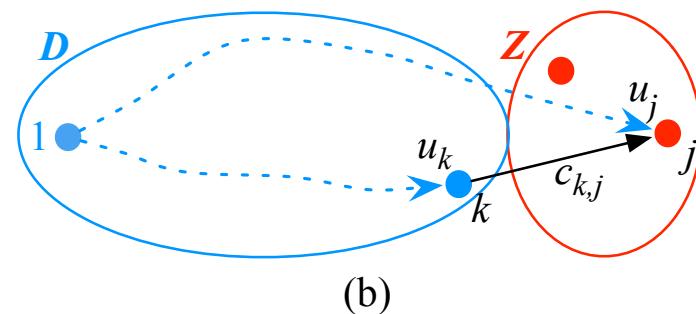
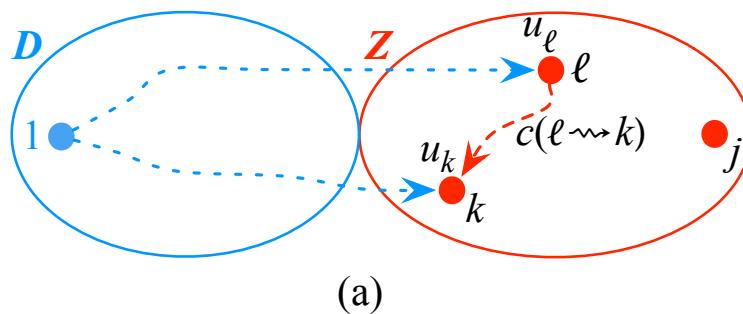
$$u_k := \min_{j \in Z} \{u_j\}; \quad D := D \cup \{k\}; \quad Z := Z - \{k\}; \quad \text{Če je } Z = \emptyset, \text{ končaj.}$$

3. Ko cena u_k postane dokončna, to lahko vpliva na ostale začasne cene $u_j, j \in Z$.

Zakaj? Začasne cene u_j , kjer $j \neq k$, so bile izračunane na 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 b).

Zato algoritem dopolnimo:

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



Po zgornjih korakih zapišimo Dijkstrov algoritom še v psevdokodi:

```
procedure Dijkstrov_Algoritom(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) do
        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. Ostale operacije v telesu zanke zahtevajo $\mathcal{O}(1)$ časa. Izvedba telesa zanke torej zahteva $3(|Z| - 1) + \mathcal{O}(1)$ operacij. Ker je v i -ti izvedbi $|Z| = n - i$, zahteva i -ta izvedba telesa $3(n - i - 1) + \mathcal{O}(1)$ operacij. Torej je zahtevnost zanke $3((n - 2) + (n - 3) + (n - 4) + \dots + 2 + 1) + (n - 1)\mathcal{O}(1)$, kar je reda $\mathcal{O}(n^2)$.

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

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.) V tem primeru topološka ureditev grafa in Dijkstrov algoritmom nista več uporabna. Na srečo pa sta algoritmom za te splošne grafe odkrila Bellman in Ford.

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

Ključni uvid Bellmana in Forda v razvoju algoritma, je:

- Najcenejša pot iz 1 v i ima 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 ima kvečjemu p povezav.
- Trivialna ekvivalenca: Za poljubna $x, p \in \mathbb{N}$ je $x \leq p \iff (x \leq p-1) \vee (x = p)$.
- Ta ekvivalenca je podlaga za naslednjo ugotovitev:
Najcenejša pot P iz 1 v i , ki ima kvečjemu p povezav, ima
 - bodisi* kvečjemu $p-1$ povezav;
 - bodisi* natanko p povezav.
- Cena $u_i^{(p)}$ poti P je odvisna od alternative (a) ali (b), ki velja za P ; zato je
 - bodisi* $u_i^{(p)} = u_i^{(p-1)}$;
 - bodisi* $u_i^{(p)} = \min_k_{(k,i) \in A} \{u_k^{(p-1)} + c_{k,i}\}$.
- Ker poti P še ne poznamo, ne vemo, katera alternativa velja, vemo pa, da mora biti cena $u_i^{(p)}$ poti P manjša izmed alternativnih cen (a') in (b'); torej $u_i^{(p)} = \min\{u_i^{(p-1)}, \min_k_{(k,i) \in A} \{u_k^{(p-1)} + c_{k,i}\}\}$. To je *rekurzivna enačba* za $u_i^{(p)}$!
- *Začetne vrednosti* za rekuzivno enačbo so jasne: $u_1^{(1)} = 0$ in $u_i^{(1)} = c_{1,i}$ za $i > 1$.

Tako sta Bellman in Ford za splošni graf izpeljala tale sistem BE:

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

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 $i = 1, 2, \dots, n$. Kako? Množici vseh cen $u_i^{(p)}$, $i = 1, 2, \dots, n$ pri izbranem p recimo 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 po vrsti, tj. po naraščajočem $p = 1, 2, \dots, n - 1$.

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 določa 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 $\mathcal{O}(n^3)$. Kaj pa prostorska zahtevnost? 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 $\mathcal{O}(1)$ pomnilniških besed. Prostorska zahtevnost Bellman-Fordovega algoritma je zato $\mathcal{O}(n)$.

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

17. Najcenejše poti med vsemi pari

Definicija problema

Dan je utežen usmerjen graf $G(V, A, c)$, kjer je $V = \{1, 2, \dots, n\}$ množica vozlišč, $A \subseteq V \times V$ množica usmerjenih povezav in $c : V \times V \rightarrow \mathbb{R} \cup \{\infty\}$ 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 $i_z = i_1 \rightarrow i_2 \rightarrow \dots \rightarrow i_\ell = i_k$ 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 predpostavljam, da $G(V, A, c)$ nima negativnih ciklov.

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

Motivacija

Algoritem B, ki reši ta problem, se sam ponuja. Če kakega od algoritmov A za iskanje cen najcenejših poti iz *izhodišča* = 1 (glej prejšnje poglavje) popravimo tako, da postane *izhodišče* njegov vhodni parameter, je B na dlani:

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

Če je A algoritem za aciklične G , je časovna zahtevnost algoritma B $\mathcal{O}(|V|^3)$; če je A Dijkstrov algoritem, je časovna zahtevnost B enaka $\mathcal{O}(|V|^3)$; in če je A Bellman-Fordov algoritem, ima B časovno zahtevnost $\mathcal{O}(|V|^4)$.

Vendar pa se nam zdi časovna zahtevnost $\mathcal{O}(|V|^4)$ algoritma B pri splošnih grafih velika. Bi lahko problem rešiti hitreje? Odgovor je *da*. Prva izboljšava, t.i. *posplošeni* Bellman-Fordov algoritem, izkorišča neko podobnost z matričnim množenjem in zmanjša časovno zahtevnost na $\mathcal{O}(|V|^3 \log |V|)$. Te izboljšave ne bomo opisali podrobneje, ker sta Floyd in Warshall, odkrila boljši algoritem.

Najcenejše poti med vsemi pari (Floyd-Warshall)

Zamisel, ki sta jo dobila Floyd in Warshall v razvoju 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 $m = 0$ je $u_{i,j}^{(0)} = c_{i,j}$.)
- Velja $u_{i,j} = u_{i,j}^{(n)}$.
- Najcenejša pot P iz i v j , na kateri imajo vsa vmesna vozlišča oznake $\leq m$,
 - (a) bodisi ne gre čez vozlišče m
 - (b) bodisi gre čez vozlišče m (torej je $P = i \rightsquigarrow m \rightsquigarrow j$).

V primeru (a) so na P le vozlišča $\leq m - 1$; v (b) gre P čez m natanko 1x.

- Cena $u_{i,j}^{(m)}$ poti P je odvisna od alternative (a) ali (b), 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).

Ker poti P ne poznamo, ne vemo, katera alternativa velja. Vemo pa, da mora biti cena $u_{i,j}^{(m)}$ poti P manjša izmed alternativnih cen (a') in (b'); torej $u_{i,j}^{(m)} = \min\{u_{i,j}^{(m-1)}, u_{i,m}^{(m-1)} + u_{m,j}^{(m-1)}\}$. To je *rekurzivna enačba* za $u_{i,j}^{(m)}$!

- Začetne vrednosti za to enačbo so $u_{i,j}^{(0)} = c_{i,j}$.

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

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

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

Ker iščemo vrednosti $u_{i,j}$ in je $u_{i,j} = u_{i,j}^{(n)}$, bomo iz zgornjega sistema izračunali vrednosti $u_{i,j}^{(n)}$. Kako? Iz rekurzivne enačbe za $u_{i,j}^{(m)}$ vidimo, da m -generacijo cen izračunamo iz $(m - 1)$ -generacije. Torej bomo računali m -generacije po naraščajočih vrednostih $m = 1, 2, \dots, n$.

Psevdokoda algoritma je zato

```
procedure Floyd_Warshallov_Algoritem(G(V,A,c));
begin
    for i := 1 to n do          //m = 0
        for j := 1 to n do
            u_{i,j}^(0) := c_{i,j}
        endfor
    endfor;
    for m := 1 to n do          //m > 0
        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 narekuje trojna zanka. Telo zanke (računanje $u_{i,j}^{(m)}$) se izvede n^3 -krat, izvedba telesa pa zahteva eno seštevanje in eno primerjanje. Časovna zahtevnost Floyd-Warshallovega algoritma je zato $\mathcal{O}(n^3)$. To je izboljšanje naivnega algoritma B, ki bi n -krat zagnal 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 iz prejšnje generacije, potrebujemo za obe generaciji $2n^2 = \mathcal{O}(n^2)$ pomnilniški besed. Ostale spremenljivke v algoritmu zahtevajo $\mathcal{O}(1)$ prostora. Zato je prostorska zahtevnost Floyd-Warshallovega algoritma reda $\mathcal{O}(n^2)$.

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

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 $u_{i,j}^{(m)}$, $1 \leq i, j \leq n$, čeprav njihov izračun zahteva tudi prejšnjo generacijo $u_{i,j}^{(m-1)}$, $1 \leq i, j \leq n$.

Implementacija računa m -generacijo na prostoru $(m - 1)$ -generacije tako, da z nobenim novim številom $u_{i,j}^{(m)}$ ne spremeni („povozi“) kakega števila $u_{i,j}^{(m-1)}$, ki se bo še potrebovalo. Pravimo, da algoritem izračuna rezultat *na mestu* (lat. *in situ*). Kako to dosežemo?

Mislimo si, da so vsa števila $(m - 1)$ -generacije razporejena v matriki U takole:

$$U = \begin{pmatrix} 1 & 2 & m & j & n \\ u_{1,1}^{(m-1)} & u_{1,2}^{(m-1)} & \dots & \vdots & \dots & u_{1,n}^{(m-1)} \\ u_{2,1}^{(m-1)} & u_{2,2}^{(m-1)} & \dots & \vdots & \dots & u_{2,n}^{(m-1)} \\ \vdots & \vdots & \ddots & \vdots & & \vdots \\ m & & u_{m,m}^{(m-1)} & u_{m,j}^{(m-1)} & & \\ & & \ddots & & \ddots & \\ & & & \ddots & \ddots & \\ & & & & \ddots & \\ i & \dots & \dots & \dots & u_{i,m}^{(m-1)} & \dots & u_{i,j}^{(m-1)} & \ddots \\ & \vdots & \vdots & & & & & \vdots \\ n & u_{n,1}^{(m-1)} & u_{n,2}^{(m-1)} & \dots & & \dots & u_{n,n}^{(m-1)} & \end{pmatrix}$$

Izračunajmo m -generacijo kot narekuje Floyd-Warshallov sistem enačb, vendar novi $u_{i,j}^{(m)}$ hrabro vpišimo nazaj na isto mesto: $U_{i,j} = \min\{U_{i,j}, U_{i,m} + U_{m,j}\}$.

Vprašanje: Ali smo z vpisom izgubili staro vsebino $U_{i,j}$, ki se bo še potrebovala? Poglejmo, kaj se dogaja, ko računamo novo vrednost $U_{i,j}$. Izračun nove vsebine komponente $U_{i,j} = u_{i,j}^{(m)}$ rabi tri stare vsebine komponent,

$$U_{i,j} = u_{i,j}^{(m-1)} \quad U_{i,m} = u_{i,m}^{(m-1)} \quad U_{m,j} = u_{m,j}^{(m-1)},$$

spremeni pa le prvo od njih, $U_{i,j}$ (pa 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 (ki po spremembi $U_{i,j}$ ostaneta nespremenjeni).

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 v m -ti vrstici in m -tem stolpcu. Ker $U_{i,j}$ ni v nobeni od njiju, računanje novega $U_{k,\ell}$ ne potrebuje starega $U_{i,j}$. Kaj pa, če je $U_{k,\ell} (\neq U_{i,j})$ v m -ti vrstici ali m -tem stolpcu? Denimo, da je v 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 robili. Analogno dokažemo, če je $U_{k,\ell}$ v m -tem stolpcu.

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

Zdaj lahko zapišemo Floyd-Warshallov algoritem še z matriko 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.
```

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

VI. Požrešnost



18. Nalaganje zabolnikov

Definicija problema

V pristanišču čaka n zabolnikov na prevoz v Afriko. Zabolniki so enakih dimenzij, razlikujejo pa se po svojih težah, saj vsebujejo različen tovor. Zabolnike bo odpeljala ladja, a ne nujno vseh, ker teža naloženih zabolnikov ne sme preseči nosilnosti ladje. Po drugi strani pa kapetan želi odpeljati čim več zabolnikov, saj bo za prevoz vsakega prejel enako plačilo. Kapetan se sprašuje:

„Katere zabolnike naj naložim, da bo cena prevoza največja?“

Problem. Zabojnike označimo s števili $1, 2, \dots, n$, težo zabojnika i s t_i , nosilnost ladje s T , plačilo za prevoz enega zabojnika pa p . Za vsak $i = 1, 2, \dots, n$ uvedimo spremenljivko $x_i \in \{0, 1\}$, ki bo povedala, kaj naj kapetan storí z zabojnikom i :

$$x_i = \begin{cases} 1 & \text{če naj naloži } i \text{ na ladjo;} \\ 0 & \text{če naj ne naloži } i \text{ na ladjo.} \end{cases}$$

Kapetanova naloga je določiti take vrednosti spremenljivkam x_1, x_2, \dots, x_n , da bo

$$\sum_{i=1}^n x_i t_i \leq T \quad (\text{nosilnost ladje})$$

in

$$\sum_{i=1}^n p x_i \quad \text{maksimalna} \quad (\text{zaslužek})$$

Vsaka prireditev vrednosti 0 ali 1 spremenljivkam x_1, x_2, \dots, x_n , ki zadošča pogoju $\sum_{i=1}^n x_i t_i \leq T$, je *dopustna* (možna) rešitev kapetanovega problema; vsaka dopustna rešitev, ki maksimizira vsoto $\sum_{i=1}^n p x_i$, pa je *optimalna*.

Opazimo naslednjo podrobnost: Kapetan želi maksimizirati zaslužek $\sum_{i=1}^n p x_i$, a ker je p konstanta, bo zaoščalo, da maksimizira vsoto $\sum_{i=1}^n x_i$. Slednja pa pomeni *število naloženih zabojsnikov*. To nam da naslednjo zamisel algoritma.

Zamisel algoritma

Nalaganje zabojsnikov naj teče v zaporednih korakih. Pred prvim korakom preimenujmo nosilnost T ladje v P , *preostanek nosilnosti* ladje. Spremenljivka P bo pred vsakim korakom povedala, kolikšno težo zabojsnikov lahko še naložimo na ladjo. Nato v vsakem koraku izberimo enega izmed nenaloženih zabojsnikov in če njegova teža t_i ne presega preostanka P nosilnosti ladje, zabojsnik i naložimo na ladjo (torej postavimo $x_i := 1$) in zmanjšamo P za t_i . Ta postopek zagotavlja, da skupna teža naloženih zabojsnikov ne bo presegla nosilnosti T ladje.

Kakšno pa bo število naloženih zabojsnikov? Kdaj bo to število maksimalno? Intuicija nam pravi, da bo to takrat, ko bomo v vsakem koraku izbrali *najlažjega* med nenaloženimi zabojsniki, saj bomo z njim najmanj zmanjšali P , s tem pa omogočili, da se kasneje naloži več zabojsnikov.

Zamisel zapišemo bolj jedrnato s spodnjim algoritmom.

```
procedure PozresnoNalaganjeZabojnikov(n,t,T) return table x;
begin
    for i:=1 to n do x_i := 0;      | Inicializiraj izhodno tabelo
    Uredi(t);                      | Uredi zabojnike po narascajoci tezi
    P := T;                         | Inicializiraj preostanek nosilnosti
    i := 1;
    while t_i <= P do              | Nalagaj, dokler se da
        x_i := 1;
        P := P - t_i
    endwhile;
    return x
end.
```

Kakovost rešitve

Trditev. Rešitev x , ki jo izračuna zgornji algoritmom, je *optimalna*.

Dokaz. Naj bo $y = y_1 y_2 \dots y_n$ poljubna druga dopustna rešitev. Dokažimo, da y ni boljša rešitev od x , torej da mora veljati $\sum_{j=1}^n p_j y_j \leq \sum_{i=1}^n p_i x_i$ oz. $\sum_{j=1}^n y_j \leq \sum_{i=1}^n x_i$. Iz algoritma sledi, da je $x = \underbrace{11 \dots 1}_k 0 \dots 0$, kjer je $1 \leq k \leq n$. Recimo, da bi bil y boljša rešitev od x , tj. $\sum_{i=1}^n x_i < \sum_{j=1}^n y_j$.

Sledilo bi $k < \sum_{j=1}^n y_j$, zapis $y = y_1 y_2 \dots y_n$ pa bi imel več kot k komponent 1. To bille komponente $y_{j_1}, y_{j_2}, \dots, y_{j_{k+\ell}}$, kjer $1 \leq \ell \leq n - k$. Opazimo, da bi za $i = 1, 2, \dots, k$ veljalo $i \leq j_i$ in prav tako tudi $t_i \leq t_{j_i}$. Sledilo bi $\sum_{i=1}^k t_i \leq \sum_{i=1}^k t_{j_i}$. To pomeni, da bi bilo prvih k zaboljnikov, naloženih na način y , vsaj tako težkih kot ves tovor, naložen na način x . Po predpostavki pa bi moral način y natovoriti še zaboljnike $j_{k+1}, \dots, j_{k+\ell}$. Poglejmo, ali bi mu to uspelo s prvim zaboljnikom j_{k+1} . Njegova teža je $t_{j_{k+1}}$, toda spet je $t_{k+1} \leq t_{j_{k+1}}$ (saj so t_1, t_2, \dots, t_n urejene in $k+1 \leq j_{k+1}$).

Ker nalaganje x ni moglo povečati natovorjene teže $\sum_{i=1}^k t_i$ s težo t_{k+1} , tudi nalaganje y ne bo moglo povečati (kvečjemu večje) natovorjene teže $\sum_{i=1}^k t_{j_i}$ s (kvečjemu večjo) težo $t_{j_{k+1}}$. To pa je v protislovju s predpostavko. Zato y ne more biti boljša rešitev od x . \square

18. Požrešna metoda

Požrešna metoda poskuša konstruirati rešitev z zaporedjem korakov. V vsakem koraku izbere tisto odločitev, ki se zdi med vsemi možnimi odločitvami tistega koraka najboljša glede na nek kriterij. Ta kriterij je seveda odvisen od reševanega problema, *požrešen* pa je zato, ker narekuje algoritmu, da „pograbi” najboljšo izmed trenutnih (lokalnih) možnosti, ne da bi se spreševal o (globalnih) posledicah te odločitve. Vsaka odločitev je tudi *dokončna*: ko je v nekem koraku sprejeta, je noben kasnejši korak ne more spremeniti.

Algoritmi, razviti s požrešno metodo, so sorazmerno enostavni in pogosto hitri. To ne preseneča, saj se ukvarjajo predvsem z iskanjem naslednje, lokalno optimalne odločitve. Samo s takim požrešnim, *lokalno* optimalnim vedenjem gradijo dopustno rešitev in si obetajo, da bo ta tudi *globalno* optimalna.

Pri nekaterih računskih problemih se požrešna metoda obnese in res vodi do optimalnih rešitev. Pri drugih problemih pa ne zagotavlja globalno optimalne rešitve; včasih se je namreč bolje odpovedati lokalno optimalnemu nadaljevanju, se odločiti za lokalno slabšo potezo in upati, da bo sledilo izdatno poplačilo za to vzdržnost v kakem kasnejšem koraku. Tako je tudi v življenju.

Če nam intuicija pravi, da naš požrešni algoritem vrača *optimalne* rešitve, je običajno, da poskušamo njihovo optimalnost tudi *dokazati*. Takrat ne smemo pozabiti, da nas intuicija lahko varja, in da zato morda takega dokaza ni. (Npr., če vemo, da je reševani problem NP-težek ali celo težji, naš požrešni algoritem pa polinomsko časovno omejen, potem je zelo verjetno, da nam dokaza za optimalnost rešitev ne bo uspelo sestaviti, ker te zelo verjetno niso optimalne.)

Če pa smo zadovoljni tudi s *približnimi* (suboptimalnimi) rešitvami, je lahko požrešna metoda dobra podlaga za razvoj *heurističnih algoritmov*, tj. algoritmov, pri katerih se v korist njihove hitrosti odpovemo zahtevi po optimalnosti rešitev.

Za nekatere (ne vse!) probleme je s požrešno metodo možno zasnovati celo take heuristične algoritme – imenovane *aproksimacijski algoritmi* – ki poleg polinomske časovne zahtevnosti zagotavljajo tudi navzgor *omejeno napako*, tj. omejeno odstopanje suboptimalne rešitve od optimalne rešitve.

19. Razvrščanje opravil

Definicija problema

Kako razdeliti opravila enakim procesorjem, da se bodo vsa izvedla čim hitreje? To je optimizacijski problem, ki ga definiramo takole.

Problem. *Naj bosta dani množica $V = \{v_1, v_2, \dots, v_n\}$ paroma neodvisnih opravil in množica $T = \{t_1, t_2, \dots, t_n\}$ njihovih trajanj ter $p \leq n$ enakih procesorjev. Opravila naj procesorjem dodeli funkcija $f : V \rightarrow \{1, 2, \dots, p\}$, njeno ceno $c(f)$ pa definirajmo kot čas, potreben za izvedbo vseh preslikanih opravila. Poiskati je treba funkcijo f^* , ki bo imela najmanjšo ceno.*

Reševanje problema

Problem je NP-težek. Poskusimo razviti algoritmom, ki bo namesto optimalne rešitve f^* v polinomskem času vrnil kakovostno približno (suboptimalno) rešitev.

Ločili bomo dve obliki tega problema.

- Pri prvi obliki prihajajo opravila eno za drugim, algoritmom pa mora vsako prispelo opravilo takoj dodeliti nekemu procesorju.
- Pri drugi obliki so vsa opravila že na voljo, zato jih algoritmom lahko pregleda, ustrezno pripravi in nato dodeli procesorjem.

Pri prvi obliki problema je algoritmom *sproten* (*on-line*), pri drugi pa ne (*off-line*). Opisali bomo sproten algoritmom *LS* (*List Scheduling*) in algoritmom *LPT* (*Largest Processing Time*), ki ni sproten.

Algoritem LS – List Scheduling

LS dodeli vsako prispelo opravilo dodeli procesorju, ki ima takrat najmanjše breme (tj. najmanjšo vsoto trajanj vseh opravil, ki so mu bila dotedaj dodeljena). To odločanje je *požrešno*: vedno izbere alternativo, ki je takat lokalno najboljša.

```
procedure LS(V,T,p) return f;
begin
    for i:=1 to n do f(v_i) := 0;      |Inicializacija f in bremen
    for j:=1 to p do breme(j) := 0;
    for i:=1 to n do
        j := trenutno_najmanj_obremenjeni_procesor;
        f(v_i) := j; breme(j) := breme(j) + t_i
    endfor;
    return f
end.
```

Časovna zahtevnost

Tretja zanka se izvede n -krat, njeno telo pa zahteva $\mathcal{O}(p) = \mathcal{O}(n)$ časa. Zato ima *LS* časovno zahtevnost $\mathcal{O}(pn) = \mathcal{O}(n^2)$.

Kakovost približnih rešitev

Ker je problem NP-težek, algoritmom LS pa ima polinomsko časovno zahtevnost, verjamemo, da konstruirane rešitve v splošnem niso optimalne. Ali lahko ocenimo njihovo kakovost, tj. *največje možno odstopanje od optimalne rešitve?*

Cena dodelitve f je $c(f) = \max_{1 \leq j \leq p} \sum_{f(v_i)=j} t_i$. Označimo s c^* ceno $c(f^*)$ optimalne dodelitve f^* . Naslednja trditev pravi, da se bodo opravila, dodeljena z algoritmom LS , izvajala *kvečjemu enkrat dlje, kot je najmanj potrebno*.

Trditev. Za rešitev f algoritma LS velja $c(f) \leq (2 - \frac{1}{p})c^*$.

Dokaz. Celotno breme je $B = \sum_{i=1}^n t_i$. Velja $\frac{B}{p} \leq c^*$ in $\forall i : t_i \leq c^*$. Naj bo B_i breme procesorja i (i bo izvedel dodeljena opravila v času B_i). Naj bo h procesor z največjim bremenom; torej velja $c(f) = B_h$. Naj bo v_j zadnje opravilo, dodeljeno procesorju h . Ta je imel pred tem najmanjše breme med vsemi procesorji, tj. breme $B_h - t_j$. Zato imajo zdaj vsi procesorji vsaj tako breme, torej $\forall i : B_i \geq B_h - t_j$. Računajmo: $B = \sum_{i=1}^p B_i = \sum_{i=1, i \neq h}^p B_i + B_h \geq \sum_{i=1, i \neq h}^p (B_h - t_j) + B_h = \sum_{i=1}^p (B_h - t_j) + t_j = p(B_h - t_j) + t_j$. Iz prvega in zadnjega dela sledi $B_h \leq \frac{B}{p} + \frac{p-1}{p}t_j$. Zato je $c(f) = B_h \leq \frac{B}{p} + \frac{p-1}{p}t_j \leq / \text{ker } \frac{B}{p} \leq c^* \text{ in } t_j \leq c^* / \leq c^* + \frac{p-1}{p}c^* = (2 - \frac{1}{p})c^*$, torej $c(f) \leq (2 - \frac{1}{p})c^*$. \square

Algoritmom LS je hevrističen, a ker zagotavlja določeno kakovost izračunanih približnih rešitev, lahko rečemo, da je celo aproksimacijski algoritmom.

Algoritem LPT – Largest Processing Time

Zdaj pa vzemimo, da so opravila že vsa prisotna. Preden jih algoritem dodeli procesorjem, jih lahko *predprocesira*, tj. pregleda in se pripravi na dodeljevanje.

Tudi algoritem *LPT* bo hevrističen, pravilo, ki ga bo vodilo pri dodeljevanju, pa mu predpišimo enako kot *LS*: *naslednje opravilo naj dodeli procesorju, ki ima takrat najmanjše breme*. Toda, za razliko od *LS* lahko *LPT* sam izbere naslednje opravilo! Kako naj to storí?

Zamislimo si tale scenarij. Denimo, da so vsi procesorji pred dodelitvijo zadnjega opravila enako obremenjeni. Če je zadnje opravilo zelo kratko, bodo vsi končali skoraj hkrati. Če pa je to opravilo zelo dolgo, bo njegova dodelitev zelo pokvarila izenačenost bremen, ne glede na to, kateremu procesorju bo dodeljeno.

Algoritem *LS* se temu scenariju ne more ogniti, saj ne more izbirati, kdaj bo kakemu opravilu dodelil procesor. Algoritem LPT pa se scenariju lahko izogne tako, da opravila uredi po padajočih trajanjih in jih dodeljuje v tem redu. Torej: LPT naj *predprocesira* opravila tako, da jih uredi po padajočem trajanju.

```

procedure LPT(V,T,p) return f;
begin
    for i:=1 to n do f(v_i) := 0;           |Inicializacija f in bremen
    for j:=1 to p do breme(j) := 0;
    Uredi(V);                                |odslej je t_i >= t_{i+1}
    for i:=1 to n do
        j := trenutno_najmanj_obremenjeni_procesor;
        f(v_i) := j; breme(j) := breme(j) + t_i
    endfor;
    return f
end.

```

Časovna zahtevnost

Ureditev n opravil rabi $\Theta(n \log n)$ časa, tretja zanka pa (kot pri algoritmu LS) $\mathcal{O}(n^2)$ časa. Tudi algoritmom LPT ima časovno zahtevnost $\mathcal{O}(n^2)$.

Kakovost približnih rešitev

Kakšne rešitve vrača algoritmom *LPT* v primerjavi z optimalnimi rešitvami? Naslednja trditev pove, da se bodo opravila, dodeljena z *LPT*, izvajala *kvečjemu tretjino dlje kot je najmanj potrebno*. Zato je tudi *LPT* aproksimacijski algoritmom, ki pa zagotavlja večjo kakovost rešitev f v primerjavi z *LS*.

Trditev. Za rešitev f algoritma *LPT* velja $c(f) \leq (\frac{4}{3} - \frac{1}{3p})c^*$.

Dokaz. Dokaz je podoben kot pri algoritmu *LS*. Naj bo $B = \sum_{i=1}^n t_i$. Spet je $\frac{B}{p} \leq c^*$. Po ureditvi opravil, je v_n zadnje opravilo, ki se dodeli, njegovo trajanje pa je t_n . Dodeli se procesorju h , ki ima zdaj breme B_h , prej pa je imel breme $B_h - t_n$. Ostali procesorji imajo bremena $B_i \geq B_h - t_n$. Tudi tu dobimo $B \geq p(B_h - t_n) + t_n$ in $c(f) = B_h \leq \frac{B}{p} + \frac{p-1}{p}t_n \leq /ker \frac{B}{p} \leq c^* / \leq c^* + \frac{p-1}{p}t_n (*)$.

Za t_n sta dve možnosti:

(a) če je $t_n \leq \frac{c^*}{3}$, dobimo z nadaljevanjem $(*)$ tole: $c(f) \leq c^* + \frac{p-1}{p}t_n \leq c^* + \frac{p-1}{p}\frac{c^*}{3} = (\frac{4}{3} - \frac{1}{3p})c^*$.

(b) če je $t_n > \frac{c^*}{3}$, opazimo, da dobi pri *optimalni* dodelitvi f^* vsak procesor največ dve opravili. (če bi f^* kakemu procesorju dodelila več kot dve opravili, bi bilo njegovo skupno breme vsa $3t_n$, kar bi bilo več kot c^* in f^* ne bi bila optimalna!) Zato jev primeru (b) največ $2p$ opravil: $n \leq 2p$. Vsako dodelitev, ki procesorjem dodeli po *največ dve* opravili, pa lahko preoblikujemo v dodelitev

$$\begin{array}{ccccccc} v_1 & \cdots & v_{2p-n} & v_{2p-(n-1)} & \cdots & v_{p-1} & v_p \\ & & v_n & \cdots & v_{p+2} & & v_{p+1} \end{array}$$

ne da bi se njena cena poslabšala! V primeru (b) to velja tudi za *optimalno* dodelitev, zato je preoblikovana dodelitev takrat tudi optimalna. Toda dodelitev f tudi ni slabša od optimalne! To pa pomeni, da je tudi f optimalna in da je v primeru (b) njena cena $c(f) = c^*$. Če (a) in (b) združimo, gotovo velja slabša od obeh ocen, zato velja $c(f) \leq (\frac{4}{3} - \frac{1}{3p})c^*$. \square

20. Trgovski potnik

Definicija problema

Danih je nekaj mest, ki so med seboj vsa neposredno povezana s cestami. Trgovski potnik želi narediti najkrajši obhod, v katerem bo obiskal vsako mesto natanko enkrat in se vrnil v začetno mesto. To je optimizacijski problem TRGOVSKI POTNIK, ki ga strogo definiramo takole:

Problem. *Dani sta množica $M = \{m_1, m_2, \dots, m_n\}$ mest, $n \geq 2$, in matrika $R = (r_{i,j})_{n \times n}$ razdalj $r_{i,j} \in \mathbb{R}^+$ od m_i do m_j . Obhod mest M je permutacija $p = (m_{i_1}, m_{i_2}, \dots, m_{i_n})$, njegova dolžina pa $d(p) = \sum_{k=1}^{n-1} r_{i_k, i_{k+1}} + r_{i_n, i_1}$. Cilj je poiskati optimalni obhod, tj. permutacijo p^* z najmanjšo dolžino.*

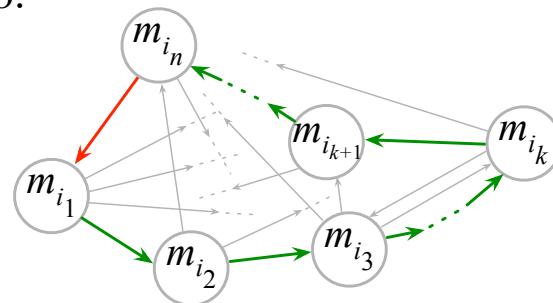
Opomba. V definiciji nismo zahtevali niti *simetričnosti* razdalj ($r_{i,j} = r_{j,i}$) niti *trikotniške relacije* med njimi ($r_{i,j} \leq r_{i,k} + r_{k,j}$), saj v splošnem cestna omrežja teh lastnosti nimajo. Dogovorimo se tudi, da je $r_{i,j} = \infty$, če iz mesta m_i ni neposredne povezave do mesta m_j .

Požrešni algoritem

Sestavimo algoritem za problem trgovskega potnika s požrešno metodo. Bistvo algoritma je, da potnik vsakokrat nadaljuje pot k *najbližjemu neobiskanemu* mestu, razen takrat, ko se *mora* vrniti v začetno mesto. Poglejmo podrobnosti.

Naj bo začetno mesto m_{i_1} . Potnikova pot je na začetku trivialna, $p = (m_{i_1})$. Naj bo po $k - 1$ korakih prehojena pot $p = (m_{i_1}, m_{i_2}, \dots, m_{i_k})$. Med doslej neobiskanimi sosedji mesta m_{i_k} potnik poišče tisto, ki je mestu m_{i_k} najbliže in ga obišče, torej vključi v dotedaj prehojeno pot. Ko obišče še zadnje mesto, je opravljena pot $p = (m_{i_1}, m_{i_2}, \dots, m_{i_n})$. Zdaj se mora le še odpraviti domov v začetno mesto m_{i_1} , torej pot skleniti v obhod.

Požrešnost algoritma je v tem, da se trgovski potnik vedno odloči za *najbližje* neobiskano sosednje mesto.



```

procedure PozresniTP(M,R) return p;
begin
    p := (m_i_1); |zacetno mesto
    for k:=1 to n-1 do
        (m_i_k,m_i_{k+1}) := najkrajsa povezava iz m_i_k v neobiskani m;
        p := p U {m_i_{k+1}}
    endfor
end.

```

Časovna zahtevnost

Zanka se izvede $(n - 1)$ -krat, jedro pa zahteva $\mathcal{O}(n)$ časa, da najde najbližjega neobiskanega soseda. Algoritem ima polinomsko časovno zahtevnost, $\mathcal{O}(n^2)$.

Bralec se verjetno veseli: TRGOVSKI POTNIK je NP-težek optimizacijski problem (saj nanj lahko prevedemo nek NP-poln problem), zato po splošnem prepričanju zanj ni polinomskega determinističnega algoritma. Smo rešili problem P=?NP in lahko pričakujemo Turingovo nagrado? Žal ne; upoštevati je treba tudi *kakovost* približnih (suboptimalnih) rešitev, ki jih vrača naš požrešni algoritem.

Kakovost približnih rešitev

Kakšna je dolžina $d(p)$ konstruiranega obhoda p ? Pri obhodu se potnik vedno odloči za *najbližje* sosednje neobiskano mesto. Ta slepa požrešnost pa se mu lahko maščuje v zadnjem koraku, ko se *mora* vrniti domov po (rdeči) povezavi (m_{i_n}, m_{i_1}) – kajti razdalja r_{i_n, i_1} je lahko *poljubno velika*, pač odvisno od primerka (M, R) problema TRGOVSKI POTNIK. S tem uvidom lahko dokažemo tole trditev:

Trditev. Za vsak $c > 1$ obstaja primerek (M, R) problema TRGOVSKI POTNIK, da za njegovo rešitev p , dobljeno z opisanim požrešnim algoritmom, velja

$$\frac{d(p)}{d(p^*)} > c.$$

Po domače: Obhodi, ki jih dobimo s požrešnim algoritmom, so lahko *poljubno mnogokrat* daljši od najkrajših obhodov.

Sklep: Požresna metoda ni vselej uspešna.

VII.

Približnost



Aproksimacijski algoritmi

Ko se srečamo z NP-težkim optimizacijskim problemom, običajno opustimo iskanje natančnega polinomskega algoritma, saj po splošni domnevi ne obstaja, in se raje posvetimo iskanju algoritmov, ki v korist hitrosti žrtvujejo natančnost.

To pomeni, da bomo zadovoljni tudi s *približno* (suboptimalno) rešitvijo, če jo le lahko izračunamo v sprejemljivem, polinomskem času. Algoritmi, pri katerih se v korist njihove hitrosti odpovemo zahtevi po optimalnosti rešitev, so *hevristični*. Za nekatere NP-težke probleme obstajajo celo takšni hevristični algoritmi – imenovani *aproksimacijski algoritmi* – ki poleg polinomske hitrosti zagotavljajo tudi *omejeno napako*, tj. omejeno odstopanje dobljenih rešitev od optimalnih rešitev.

21. Metrični trgovski potnik

Definicija problema

Pri problemu TRGOVSKI POTNIK razdalja od mesta m_i do mesta m_j v splošnem ni bila enaka razdalji od m_j do m_i ; v splošnem je torej veljalo $r_{i,j} \neq r_{j,i}$. Če pa so med mesti vse ceste dvosmerne, je od enega mesta do drugega enako daleč kot nazaj, torej je $r_{i,j} = r_{j,i}$ za vsak par i, j . Matrika razdalj $R = (r_{i,j})$ je takrat *simetrična*. Včasih lahko predpostavimo celo, da je iz m_i v m_j preko m_k kvečjemu dlje kot pa iz m_i neposredno v m_j . Za razdalje med mesti takrat velja *trikotniška neenakost* $r_{i,j} \leq r_{i,k} + r_{k,j}$ za vsako trojico i, j, k . Kadar veljata obe predpostavki, se novi problem imenuje METRIČNI TRGOVSKI POTNIK.

Problem. *Dani sta množica $M = \{m_1, m_2, \dots, m_n\}$ mest, $n \geq 2$, in matrika $R = (r_{i,j})_{n \times n}$ razdalj $r_{i,j} \in \mathbb{R}^+$ od m_i do m_j . Za poljubne i, j, k veljata $r_{i,j} = r_{j,i}$ in $r_{i,j} \leq r_{i,k} + r_{k,j}$. Obhod mest M je permutacija $p = (m_{i_1}, m_{i_2}, \dots, m_{i_n})$, njegova dolžina pa $d(p) = \sum_{k=1}^{n-1} r_{i_k, i_{k+1}} + r_{i_n, i_1}$. Cilj je poiskati optimalni obhod, tj. permutacijo p^* z najmanjšo dolžino.*

Opomba. Spet se dogovorimo, da je $r_{i,j} = \infty$, če med mestoma m_i in m_j ni neposredne povezave.

„Prvi“ aproksimacijski algoritem

1. V grafu $G(V, E, d)$ konstruiraj minimalno vpeto drevo T .

Trditev. $w(T) < d(p^*)$.

Dokaz. Če izbrišemo povezavo v p^* , dobimo vpeto drevo T' ; njegova teža $w(T')$ je vsaj $w(T)$; zato $w(T) \leq w(T') < d(p^*)$. \square

2. Iz začetnega vozlišča konstruiraj obhod S po drevesu T , ki vsako povezavo drevesa T prečka dvakrat.

Tedaj: $d(S) = 2w(T) < 2d(p^*)$.

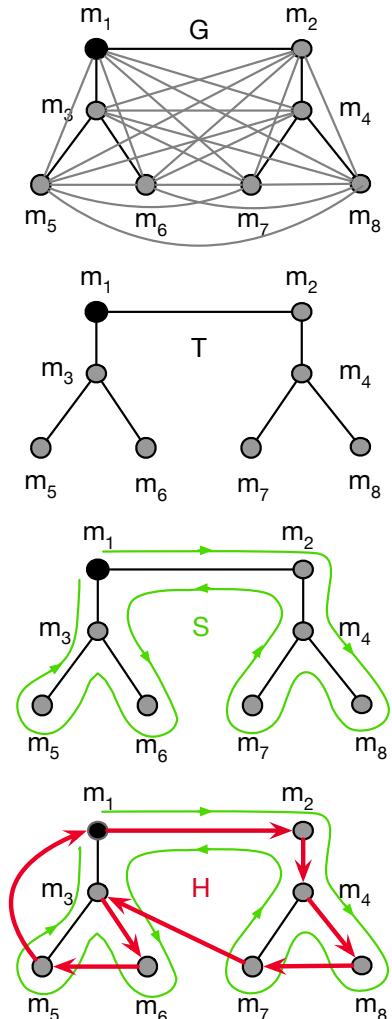
(S ni dopustna rešitev, ker S ni Hamiltonov obhod.)

3. Iz obhoda S konstruiraj Hamiltonov obhod H po grafu G , pri čemer preskoči vsako že obiskano vozlišče.

(H je dopustna rešitev)

Izrek: $d(H) \leq d(S) < 2d(p^*)$.

Algoritem je *polinomske* časovne zahtevnosti.
Vrne Hamiltonov obhod, ki je *kvečjemu enkrat daljši od najkrajšega Hamiltonovega obhoda*.



Christofidesov aproksimacijski algoritem

1. V grafu $G(V, E, d)$ poiščemo minimalno vpeto drevo T .

Vemo: V vsakem grafu je sodo mnogo vozlišč lihih stopenj.

Pišimo: $V_{\text{odd}} :=$ množica vseh vozlišč lihe stopnje drevesa T .
Torej je $|V_{\text{odd}}| = 2k$ za neki naravni k .

Primer: Na sliki je $|V_{\text{odd}}| = |\{m_3, m_4, m_5, m_6, m_7, m_8\}| = 6$.

Definicija. Ujemanje v V_{odd} je vsaka razdelitev M množice V_{odd} v disjunktne pare $\{m_{i_1}, m_{j_1}\}, \{m_{i_2}, m_{j_2}\}, \dots, \{m_{i_k}, m_{j_k}\}$.
Teža ujemanja je vsota razdalj, definiranih s pari v M .

Primer: Možno ujemanje je $M = \{\{m_3, m_4\}, \{m_5, m_6\}, \{m_7, m_8\}\}$.

Drugo ujemanje $M = \{\{m_3, m_7\}, \{m_4, m_5\}, \{m_6, m_8\}\}$.

V množici V_{odd} je $|V_{\text{odd}}|(|V_{\text{odd}}|-1)/2$ ujemanj. (Dokaži!)

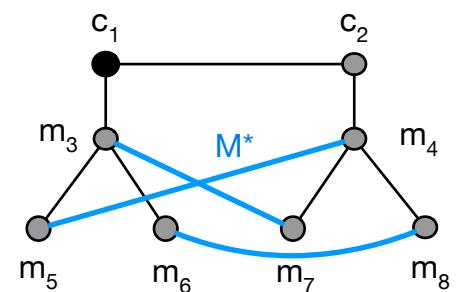
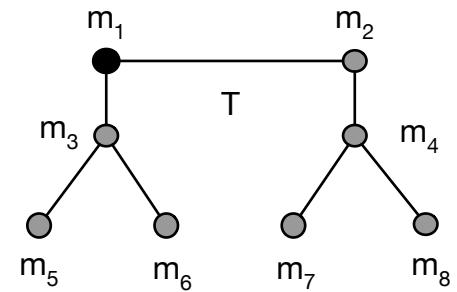
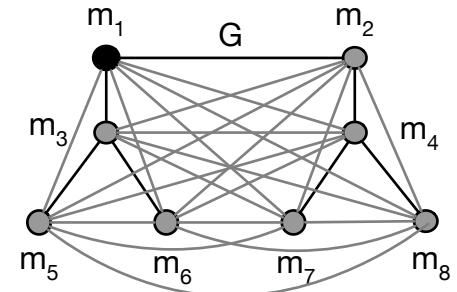
Najmanjše ujemanje M^* v V_{odd} je M z najmanjšo težo.

Vemo: M^* lahko snajdemo v polinomskem času glede na $|V_{\text{odd}}|$.

2. V množici V_{odd} poiščemo najmanjše ujemanje M^* .

Naj bo $M^* = \{\{m_1, m_2\}, \{m_3, m_4\}, \dots, \{m_{2k-1}, m_{2k}\}\}$.

(Zaradi enostavnosti smo preindeksirali mesta.)



(nadaljevanje)

3. Drevesu T dodamo povezave iz M^* in dobimo graf $T \cup M^*$.

Dokažemo: Vsaka točka v $T \cup M^*$ ima sodo stopnjo.

Vemo: Če so vse točke grafa sodih stopenj, je graf Eulerjev.

(ima Eulerjev obhod, tj. obhod, ki prečka vsako povezavo $2x$)
Torej je $T + M^*$ Eulerjev graf.

4. V grafu $T \cup M^*$ sestavimo Eulerjev obhod S .

Toda: S ni Hamiltonov obhod (zato ni dopustna rešitev).

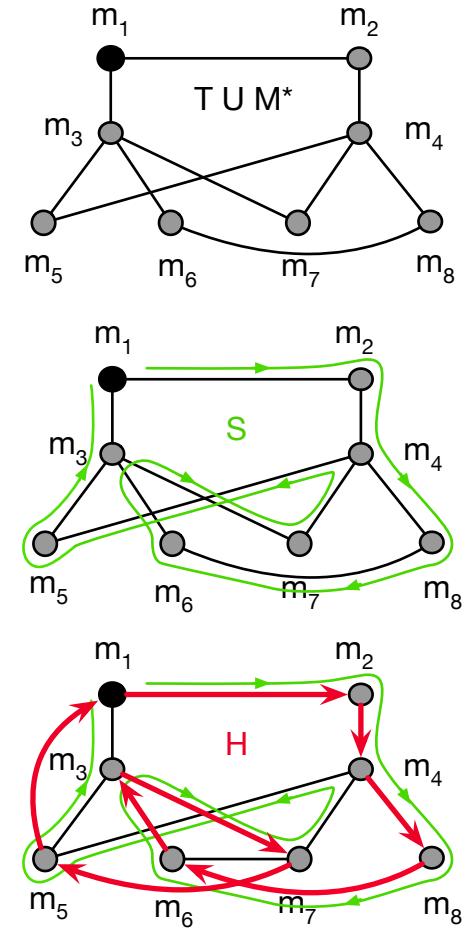
5. Iz Eulerjevega obhoda S sestavimo Hamiltonov obhod H grafu G tako, da preskočimo vsako že obiskano vozlišče na S .

Izrek. $d(H) < 1.5 d(p^*)$.

Algoritem ima polinomsko časovno zahtevnost

Vrne Hamiltonov obhod, ki je *kvečjemu pol daljši*.

od najkrajšega Hamiltonovega obhoda.



VIII. Naključnost



Naključnostni algoritmi

Ko se srečamo z NP-težkim ali kakim drugim računsko zahtevnim problemom, lahko namesto iskanja aproksimacijskega algoritma (ki v polinomskem času vrne kakovostno približno rešitev) iščemo *naključnostni algoritem*, ki v *polinomskem* času vrne rešitev, ki je z *veliko gotovostjo* pravilna (čeprav je verjetnost, da je napačna, večja od 0). Naključnostni algoritmi so uporabni tudi pri reševanju NP-polnih ali kakih drugih računsko zahtevnih odločitvenih problemov.

22. Razpoznavanje praštevil

Problem. *Ali je naravno število n praštevilo?*

Naivni algoritem

Prva zamisel za običajen, *deterministični* algoritem za ta problem je enostavna: preveri, ali je n deljivo s kakim od števil med 2 in $\sqrt[2]{n}$:

```
procedure Prastevilo(n) return DA/NE;  
begin  
    p := DA;                      //Zacasni odgovor  
    for i:=2 to  $\lfloor \sqrt{n} \rfloor$  do  
        if i deli n then  
            p := NE; exit  
        endif  
    endfor;  
    return(p)  
end.
```

Na prvi pogled se zdi, da je časovna zahtevnost tega algoritma reda $\mathcal{O}(\sqrt{n})$, saj se deljivost števila n preveri kvečjemu ($\sqrt[2]{n}-1$)-krat. Toda ta razmislek je napačen. Namreč, velikost danega primerka tega problema ni n , pač pa $s = \lceil \log_2 n \rceil = \Theta(\log n)$. (Velikost primerka problema je velikost prostora, ki ga potrebuje koda primerka.)

Ker je

$$\sqrt[2]{n} = \sqrt[2]{2^{\log_2 n}} \leqslant \sqrt[2]{2^s} = 2^{\frac{1}{2}s} = \Theta(2^{\frac{1}{2}s}),$$

je resnična časovna zahtevnost problema, izražena z s , eksponentna!

Miller-Rabinov algoritem

Do leta 2002 je bil med praktično in teoretično najučinkovitejšimi algoritmi za razpoznavanje praštevil t.i. *Miller-Rabinov algoritem*. Poglejmo, kako ga dobimo.

Začnimo z znanim *Malim Fermatovim izrekom*:

Izrek. (Mali Fermatov izrek) če je n praštevilo in w poljubno celo število, ki ni deljivo z n , potem je $w^{n-1} - 1$ mnogokratnik števila n .

Omejimo se na števila $w \in \{1, 2, \dots, n-1\}$. Potem iz izreka sledi

$$n \text{ je praštevilo} \implies \forall w (1 \leq w < n) : w^{n-1} \equiv 1 \pmod{n}.$$

Obrnimo zgornjo implikacijo, da dobimo ekvivalentni zapis

$$\exists w (1 \leq w < n) : w^{n-1} \not\equiv 1 \pmod{n} \implies n \text{ je sestavljen.}$$

Uvedimo okrajšavo $F(w, n) \stackrel{\text{def}}{=} w^{n-1} \not\equiv 1 \pmod{n}$. Zgornja implikacija se skrči v

$$\exists w (1 \leq w < n) : F(w, n) \implies n \text{ je sestavljen}$$

Število w , za katerega velja $F(w, n)$, je t.i. *Fermatova priča* za sestavljenost števila n .

Ugotovili smo: Če najdemo $w \in \{1, 2, \dots, n - 1\}$, za katerega velja $F(w, n)$, je n sestavljen število. Iskanje takega w bo naloga naslednjega algoritma:

```
procedure Sestavljen(n) return DA/NE;
begin
    s := NE;                                |Zacasni odgovor
    for w:=1 to n do
        if F(w,n) then
            s := DA; exit
        endif
    endfor;
    return(s)
end.
```

Če algoritem *najde* Fermatovo pričo w , je n gotovo sestavljen število: odgovoru DA lahko verjamemo. Kaj pa če *ne najde* Fermatove priče? Ali to pomeni, da n ni sestavljen, tj. ali smemo verjeti odgovoru NE? Odgovor je presenetljiv: *Odgovoru NE ne smemo zaupati*, kajti *obstajajo sestavljenia števila, ki nimajo Fermatovih prič!* Npr., 561 je sestavljen, a brez Fermatovih prič. To so t.i. *Carmichaelova števila*. Za 561 so taka še 1105, 1729, 2465, 2821, 6601, 8911, od leta 1994 pa vemo, da jih je neskončno mnogo.

Sklep. *Odgovoru DA prejšnjega algoritma lahko zaupamo, odgovoru NE pa ne smemo zaupati* (ker algoritem vrne lažni NE, če je n Carmichaelovo število).

Zamisel. Zakaj pa ne bi preverili odgovora NE tako, da bi preverili, ali je n Carmichaelovo število? (Če bi se izkazalo, da je Carmichaelovo število, bi izvedeli, da je NE lažen.) Ta zamisel je slaba, ker moramo pri preverjanju, ali je n Carmichaelovo število, preveriti tudi, ali je n sestavljeno – to pa je problem, ki ga pravkar rešujemo!

Miller in Rabin sta preizkušala razne zamisli, kako se izogniti lažnim odgovorom. Tule je ena od ključnih zamisli, ki sta jo uporabila vsak na svoj način.

Zamisel. Zamenjajmo predikat $F(w, n)$ z *ostrejšim* predikatom $W(w, n)$, takim, da bo namesto implikacije

$$\exists w (1 \leq w < n) : F(w, n) \implies n \text{ je sestavljen}$$

veljala *ekvivalenca*

$$\exists w (1 \leq w < n) : W(w, n) \iff n \text{ je sestavljen.}$$

Če tak predikat $W(w, n)$ obstaja in ga bomo uspeli najti, potem Carmichaelova števila ne bodo več povzročala težav. Zakaj? Če bomo v prejšnjem algoritmu $F(w, n)$ zamenjali z $W(w, n)$, bomo dobili skoraj identičen algoritem

```
procedure Sestavljen(n) return DA/NE;
begin
    s := NE;
    for w:=1 to n do
        if W(w,n) then
            s := DA; exit
        endif
    endfor;
    return(s)
end.
```

|Zacasni odgovor
|Lahko tudi w:=2 ...

ki pa bo dajal *verodostojna* odgovora DA/NE, da bomo lahko *vsakemu od njiju zaupali*.

Ali tak predikat $W(w, n)$ sploh obstaja? Da, tako nam zagotavlja tale izrek.

Trditev.

$$W(w, n) = w^{n-1} \not\equiv 1 \pmod{n} \vee \exists i (2^i | (n-1) \wedge 1 < \gcd(w^{(n-1)/2^i} - 1, n) < n)$$

Dokaz. Če velja prvi člen, tj. $w^{n-1} \not\equiv 1 \pmod{n}$, je w Fermatova priča za sestavljenost n .

Drugi člen, tj. $\exists i(\dots)$, izraža, kdaj je n deljiv z netrivialnim deliteljem ($\neq 1, n$). Prvi ali drugi člen veljata natanko tedaj, ko je n sestavljen število. \square

Število w , za katerega velja $W(w, n)$, imenujemo *priča za sestavljenost* števila n .

Torej za ta predikat $W(w, n)$ velja ekvivalenca

$$\exists w (1 \leq w < n) : W(w, n) \iff n \text{ je sestavljen.}$$

Da bi izvedeli, ali je n sestavljeni število ali je praštevilo, moramo ugotoviti, ali je predikat $\exists w (1 \leq w < n) : W(w, n)$ resničen ali neresničen. Vprašanje pa je, ali se da to ugotoviti hitro, v *polinomskem* času glede na $s = \lceil \log n \rceil$? Miller je prišel leta 1975 do zanimivega odkritja (ki pa še ni zadovoljiv odgovor na vprašanje): sestavil je deterministični algoritem, ki na podlagi zgornje ekvivalence razpoznavajo sestavljeni števila (oz. praštevila) v času reda $\mathcal{O}(s^4)$, vendar pa se ta algoritem naslanja na znano *Razširjeno Riemannovo Hipotezo (ERH)*, ki je še danes *nedokazana*. (če ERH v resnici ne velja, potem Millerjev algoritem ni pravilen.)

Ali se da v polinomskem času izračunati vrednost izraza $\exists w (w > 0) : W(w, n)$ brez predpostavke, da velja ERH? Kako ugotoviti, ali obstaja $w \in \{1, 2, \dots, n\}$, pri katerem je $W(w, n)$ resničen? Tu je Rabin v razmišljjanju vključil *naključnost*. Kako? Denimo, da *naključno* izberemo $w \in \{1, 2, \dots, n\}$ in z njim izračunamo $W(w, n)$. če je n v resnici sestavljeni, ima vsaj eno pričo za svojo sestavljenost, zato je verjetnost $P_W(n)$, da bo naključno izbrani w priča, *pozitivna*; $P_W(n) > 0$. Ta verjetnost bo odvisna od števila prič, ki jih ima sestavljeni n . Rabin pa je dokazal, da velja tole.

Izrek. Če je $n > 4$ sestavljeni število, ima vsaj $\frac{3}{4}(n-1)$ prič za svojo sestavljenost.

Dokaz. Je dolg in zapleten. Radovedni bralec si ga lahko ogleda v članku Michael O. Rabin, Probabilistic Algorithm for Testing Primality. *Journal of Number Theory*, **12**:128-138(1980). \square

Zato je verjetnost, da bo naključni w priča za sestavljenost n , vsaj $\frac{3}{4}$;

$$P_W(n) \geq \frac{3}{4}.$$

Zdaj pa poglejmo tale algoritem:

```
procedure Sestavljen(n) return DA/NE;  
begin  
    s := NE;  
    w := Nakljucno_izberi(1,n);  
    if W(w,n)  
        then s := DA  
    endif;  
    return(s)  
end.
```

|Zacasni odgovor
|Izberi w iz {1,2,...,n}

Če ta algoritem vrne odgovor DA, smo lahko gotovi, da je n sestavljen število, saj je bil izbrani w priča za sestavljenost števila n . Zato *odgovoru DA lahko zaupamo*.

Kaj pa če algoritem vrne odgovor NE? Potem očitno velja ena od dveh možnosti:

- v $\{1, 2, \dots, n\}$ ni nobene priče za sestavljenost n ;
(Zato je bil odgovor NE pravilen.)
- v $\{1, 2, \dots, n\}$ je vsaj ena priča za sestavljenost n , a nobena ni bila izbrana.
(Zato je bil odgovor NE lažen.)

Torej je odgovor NE lahko pravilen ali lažen, toda ne vemo, kaj od tega je v resnici. Zato *odgovoru NE ne smemo zaupati*.

Kakovost odgovora

Kako bi se znebili negotovosti, ki obremenjuje odgovor NE? Zaženimo zgornji algoritmom k -krat (npr. $k = 100$). V vsakem zagonu bo algoritmom naključno izbral w in na njegovi podlagi vrnil neki odgovor DA/NE. Zdaj pa se vprašajmo:

Kakšna je verjetnost dogodka $L(k)$, da algoritmom v vseh k zagonih vrne lažni NE?

Označimo iskano verjetnost s $P_L(k)$. Algoritmom vrne lažni NE le, če je n sestavljeno, hkrati pa izbor števila w zgreši vse priče za sestavljenost n . Ker pa ima po zgornjem izreku sestavljeni število n vsaj $\frac{3}{4}n$ prič, je verjetnost, da nobena od njih ne bo izbrana, kvečjemu $\frac{1}{4}$. Zaporedne izvedbe algoritma pa so neodvisni dogodki, zato je verjetnost, da se to zgodi v vseh k zagonih algoritma, manjša od $(\frac{1}{4})^k$, torej

$$P_L(k) < (\frac{1}{4})^k = (\frac{1}{2})^{2k}.$$

Zdaj vidimo: z večanjem števila k zagonov algoritma lahko poljubno zmanjšamo verjetnost $P_L(k)$, da je NE lažen. Večkrat ko bo algoritmom zaporedoma odgovoril NE („ n ni sestavljen“), večja bo verjetnost $1 - P_L(k)$, da je odgovor NE pravilen.

Primer. Naj bo $k = 100$. Verjetnost, da bo algoritmom 100-krat zgrešil vse priče je $P_L(100) < (\frac{1}{2})^{200} \approx 6.2 \times 10^{-61}$. Torej, če za dani n zaženemo algoritmom 100-krat in vsakokrat dobimo odgovor NE („ n ni sestavljen“), je verjetnost, da je n praštevilo, enaka $1 - P_L(100) = 0.\underbrace{999\dots}_{60 \text{ devetic}} 99937$. \square

Časovna zahtevnost

V navedenem članku je Rabin pokazal, da je časovna zahtevnost ene izvedbe algoritma reda $\mathcal{O}(s)$, kjer je $s = \lceil \log_2 n \rceil$, torej *polinomska*. Zato je časovna zahtevnost k izvedb algoritma reda $k \cdot \mathcal{O}(s)$, kar je pri konstantnih k še vedno polinomska zahtevnost.

Miller-Rabinov algoritom `Sestavljeno_MR(n,k)` se v psevdokodi glasi takole:

```
procedure Sestavljeno_MR(n,k) return (DA/NE, real);
begin
    i:=1;
    repeat
        s := NE;
        w := Nakljucno_izberi(1,n);
        if W(w,n)
            then s := DA
        endif;
        i++
    until (s=DA or i=k);
    if s=DA
        then return (DA, 1)           | n je z verj. 1 sestavljeno
        else return (NE, 1-P_L(k))   | n je z verj. 1-P_L(k) prastevilo
    endif
end.
```

Sklep. Miller-Rabinov algoritmom razpozna sestavljeno število v polinomskem času z verjetnostjo napake 0, prastevilo pa v polinomskem času s pozitivno, a poljubno majhno verjetnostjo napake.