Oblikovanje i analiza algoritama

Matej Mihelčić

Prirodoslovno-matematički fakultet, Sveučilište u Zagrebu

matmih@math.hr

27. listopada, 2023.

Rekurzivni algoritmi sortiranja - merge_sort

Kod s predavanja kolegija programiranje 2.

```
1  lista merge_sort(lista prvi){
2  /* Sortira listu Merge_Sort algoritmom. */
3       lista zadnji, prvi_2;
4  /* Test na praznu ili jednoclanu listu. */
5       if ((prvi == NULL) || (prvi->sljed == NULL))
6            return prvi;
7  /* U nastavku obrade, lista ima bar dva
8  elementa. */
9  /* Raspolovi listu. Pokazivac zadnji pokazuje
10 na zadnjeg u prvom dijelu. Pokazivac prvi_2 je
11 pomocni i sluzi za raspolavljanje liste. */
12      zadnji = prvi;
13      prvi_2 = prvi->sljed;
```

Rekurzivni algoritmi sortiranja - merge_sort

```
14 /* Pomicemo zadnjeg za JEDNO mjesto, prvi_2 za
15 DVA mjesta, dok prvi_2 ne bude na kraju liste. */
16      while ((prvi_2 != NULL) &&
17             (prvi_2->sljed != NULL)) {
18               zadnji = zadnji->sljed;
19               prvi_2 = prvi_2->sljed->sljed;
20         }
21 /* Pokazivac zadnji sad korektno pokazuje na
22 zadnjeg u prvom dijelu. Pokazivac prvi_2 zadamo
23 kao prvog u drugom dijelu (prvi iza zadnjeg) i
24 korektno zavrsavamo prvi dio. */
25     prvi_2 = zadnji->sljed;
26     zadnji->sljed = NULL;
27 /* Rekurzivno sortiranje i merge. */
28     prvi = merge(merge_sort(prvi),
29     merge_sort(prvi_2));
30    return prvi; }
```

$$T(n) = 2T(\tfrac{n}{2}) + n \Rightarrow T(n) \in \Theta(n \cdot log_2 n)$$

Rekurzivni algoritmi sortiranja - QuickSort algoritam

Kod s predavanja kolegija programiranje 2.

```c
#include <stdio.h>
/*QuickSort algoritam. x[l] je kljucni element.*/

void swap(int *a, int *b)
{
  int temp;
  temp = *a;
  *a = *b;
  *b = temp;
  return;
}
```

Rekurzivni algoritmi sortiranja - QuickSort algoritam

```
12 void quick_sort(int x[], int l, int d)
13 {
14    int i, j;
15    if (l < d) {
16        i = l + 1; j = d;
17
18       while (i <= j) {
19           while (i <= d && x[i] <= x[l]) ++i;
20           while (x[j] > x[l]) --j;
21           if (i < j) swap(&x[i], &x[j]); }
22
23      if (l < j) swap(&x[j], &x[l]);
24          quick_sort(x, l, j - 1);
25          quick_sort(x, j + 1, d);
26      }
27 return; }
```

Quick sort - analiza složenosti

Vremenska složenost u **najgorem** slučaju (sortirani niz, interno $k = 1$ neuravnotežena particija).

$$T(n) = T(0) + T(n-1) + \Theta(n) \Rightarrow T(n) \in \Theta(n^2)$$

Pretpostavke (randomizirani Quick sort)

- svi polazni poretci (permutacije) su jednako vjerojatni
- nema jednakih ključeva

Dodatna prepostavka:

- svi poretci u lijevoj i desnoj particiji su jednakovjerojatni

Quick sort - analiza složenosti

Propozicija: Očekivani broj usporedbi na nizu od n međusobno različitih elemenata je $\mathcal{O}(nlog_2 n)$.

Dokaz 1. Neka su a_i i a_j i-ti i j-ti elementi u sortiranom polju.

$$X_{ij} = \begin{cases} 1, & \text{ako se uspoređuju } a_i \text{ i } a_j \\ 0, & \text{inače} \end{cases}$$

Ukupni broj usporedbi $X = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij}$.
$E(X) = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} E(X_{ij}) = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij} \cdot P(a_i \text{ se uspoređuje s } a_j)$
$= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j-i+1} = (k = j - i) = \sum_{i=1}^{n-1} \sum_{k=1}^{n-i} \frac{2}{k+1}$
$< \sum_{i=1}^{n-1} \sum_{k=1}^{n-i} \frac{2}{k} =$ estimacija integralom $\in \sum_{i=1}^{n-1} \mathcal{O}(ln(n)) \in$
$\mathcal{O}(n \cdot ln(n)) = \mathcal{O}(\frac{1}{log_2 e} nlog_2 n) = \mathcal{O}(nlog_2 n)$

Neka je $A_{ij} = \{a_i,\ a_{i+1}, \ldots, a_j\}$
a_i je pivotni element ili a_j
$\frac{1}{j-i+1} + \frac{1}{j-i+1}$

Sortiranjem korištenjem posebnih struktura podataka

Heap sort - algoritam sortiranja koji koristi strukturu podataka **hrpu** za sortiranje niza brojeva.

1. Dodajemo redom elemente u hrpu (ovisno o smjeru sortiranja imamo element s maksimalnom/minimalnom vrijednosti u korjenu).
2. Redom izbacujemo korjen hrpe dok hrpa nije prazna.

Pošto i operacija ubacivanja i izbacivanja elemenata iz hrpe imaju složenost $\mathcal{O}(log_2 n)$, složenost algoritma **heap sort** je $\mathcal{O}(nlog_2 n)$.

Sortiranje u linearnom vremenu - Counting sort

Counting sort je algoritam sortiranja koji koristi dodatnu memoriju (**umjesto usporedbi**) za sortiranje polja.
Pretpostavke:

- elementi ulaznog niza poprimaju vrijednosti $1, \ldots, n$ za $n \in \mathbb{N}$.

Dodatna memorija:

- polje za sortirani niz (jednake veličine kao ulazni)
- pomoćno polje za spremanje rezultata (veličine $n + 1$)

Složenost: $\mathcal{O}(n + m)$, za ulazno polje od m elemenata čije vrijednosti su $\leq n$.

Sortiranje u linearnom vremenu - Counting sort

Neka je zadano ulazno polje A s 10 elemenata:

3	0	1	4	1	2	3	5	2	0

Pomoćno polje B s 6 elemenata (pošto je $n = 5$).

0	0	0	0	0	0

I polje za sortirani niz C veličine 10 elemenata.

Sortiranje u linearnom vremenu - Counting sort

Izbrojimo **broj pojavljivanja** svake vrijednosti u polju A i zapišemo u polje B.

2	2	2	2	1	1

Izračunamo **kumulativnu** (**prefiksnu**) sumu elemenata u polju B, te rezultat spremimo u isto polje. i-ti element sadrži $\sum_{j=1}^{i} B[i]$.

2	4	6	8	9	10

Sortiranje u linearnom vremenu - Counting sort

Iteriramo po polju A s desna na lijevo i postavljamo element na ispravnu poziciju (poziciju koju element mora imati u sortiranom polju) u polju C.

3	0	1	4	1	2	3	5	2	0

Promotrimo element $B[A[i]]$.

2	4	6	8	9	10

Zapišemo element $A[i]$ na poziciju $C[B[A[i]] - 1]$ (to je njegova prirodna pozicija u sortiranom polju).

	0								

Umanjimo vrijednost elementa $B[A[i]]$.

1	4	6	8	9	10

Sortiranje u linearnom vremenu - Counting sort

$A =$

3	0	1	4	1	2	3	5	2	0

$B =$

1	4	6	8	9	10

$C =$

	0				2				

$A =$

3	0	1	4	1	2	3	5	2	0

$B =$

1	4	5	8	9	10

$C =$

	0				2				5

$A =$

3	0	1	4	1	2	3	5	2	0

$B =$

1	4	5	8	9	9

$C =$

	0				2		3		5

Sortiranje u linearnom vremenu - Counting sort

$A =$

3	0	1	4	1	2	3	5	2	0

$B =$

1	4	5	7	9	9

$C =$

	0			2	2		3		5

$A =$

3	0	1	4	1	2	3	5	2	0

$B =$

1	4	4	7	9	9

$C =$

	0		1	2	2		3		5

$A =$

3	0	1	4	1	2	3	5	2	0

$B =$

1	3	4	7	9	9

$C =$

	0		1	2	2		3	4	5

Sortiranje u linearnom vremenu - Counting sort

$A =$ | 3 | 0 | 1 | 4 | 1 | 2 | 3 | 5 | 2 | 0 |

$B =$ | 1 | 3 | 4 | 7 | 8 | 9 |

$C =$ | | 0 | 1 | 1 | 2 | 2 | | 3 | 4 | 5 |

$A =$ | 3 | 0 | 1 | 4 | 1 | 2 | 3 | 5 | 2 | 0 |

$B =$ | 1 | 2 | 4 | 7 | 8 | 9 |

$C =$ | 0 | 0 | 1 | 1 | 2 | 2 | | 3 | 4 | 5 |

$A =$ | 3 | 0 | 1 | 4 | 1 | 2 | 3 | 5 | 2 | 0 |

$B =$ | 0 | 2 | 4 | 7 | 8 | 9 |

$C =$ | 0 | 0 | 1 | 1 | 2 | 2 | 3 | 3 | 4 | 5 |

$B =$ | 0 | 2 | 4 | 6 | 8 | 9 |

Sortiranje u linearnom vremenu

Algoritmi koji sortiraju u linearnom vremenu sadrže:

- Radix sort
- Bucket sort

Oblikovanje efikasnih algoritama - dinamičko programiranje

Različite metode za konstrukciju algoritama:

- smanji pa vladaj (eng. *reduce and conquer* - problem hanojskih tornjeva)
- podijeli pa vladaj (eng. *divide and conquer* - merge sort, quick sort, binarno pretraživanje...)
- dinamičko programiranje (pronalaženje najkraćeg puta, problem ruksaka...)
- pohlepni pristup (kontinuirani problem ruksaka, minimalno razapinjuće stablo ...)
- backtracking (problem n kraljica, ...)
- branch and bound (šah, n kraljica, ...)

Oblikovanje efikasnih algoritama - dinamičko programiranje

Promatramo kombinatorne algoritme:

- problemi prebrajanja
- problemi na grafovima i sličnim diskretnim strukturama
- **problemi kombinatorne optimizacije** (najkraći put, najmanja cijena, maksimalni profit, ...)

Dinamičko programiranje primjenjujemo na probleme čije rješenje možemo interpretirati kao rezultat nekog niza odluka.

Primjeri:

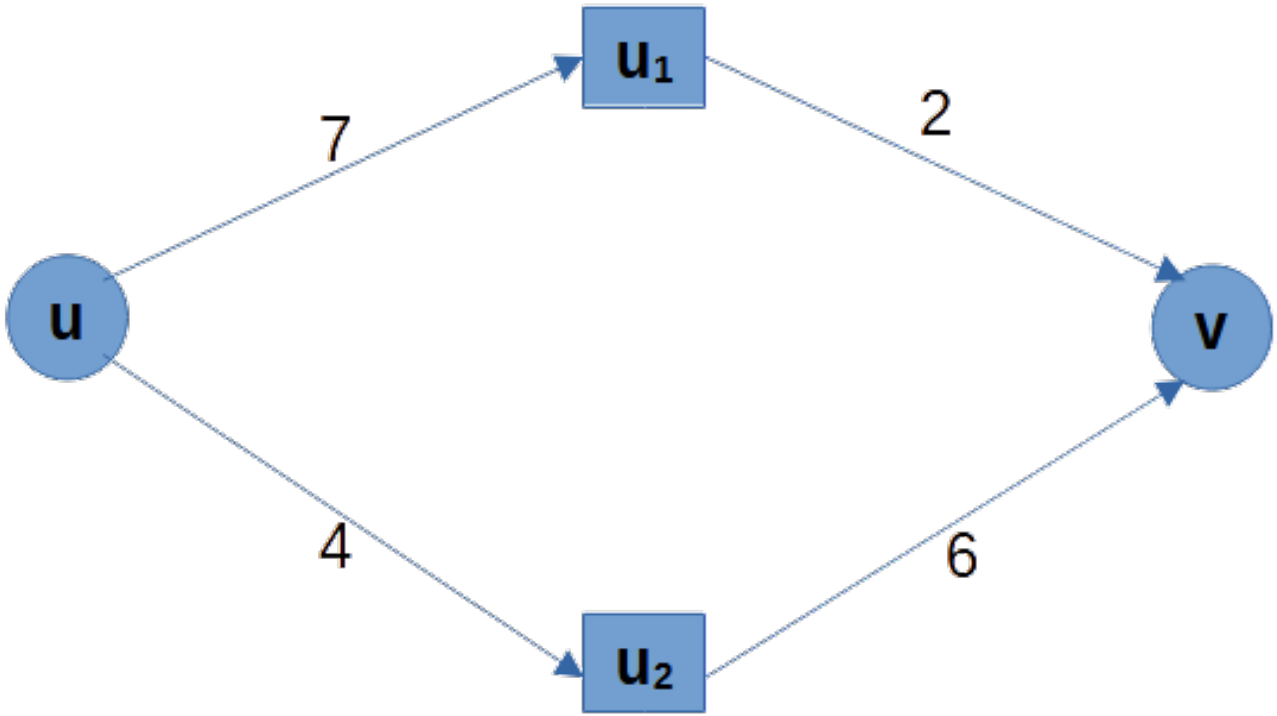
- pronalaženje najkraćeg puta
- problem ruksaka
- određivanje podvektora maksimalne sume elemenata
- ulančano množenje matrica.

Problem - najkraći put

Problem: Zadano je n gradova (označavamo ih brojevima 1 do n). Između pojedinih gradova postoje direktne veze, dok veza između nekih gradova prolazi kroz druge gradove. Dozvoljene su i jednosmjerne veze (i, j), koje mogu biti različite od (j, i).

- Dobivamo graf G čiji vrhovi $V = \{1, \ldots, n\}$ su gradovi a bridovi $E = \{(i, j), i, j \in V\}$ su direktne jednosmjerne veze između gradova. Pretpostavljamo da je $|E| = m$. Svakoj direktnoj, jednosmjernoj vezi je pridružena težina $d(i, j)$ (udaljenost, cijena, trajanje ...).
- Pretpostavljamo da je graf **povezan** (ne treba raditi provjeru). Problem povezanosti grafa se može učinkovito riješiti (BFS ili DFS uz složenost $\mathcal{O}(n + m)$).
- **Cilj**: pronaći najkraći put između dva zadana vrha u i v. Dakle, tražimo niz gradova $u_1, u_2, \ldots, u_k$ takvih da put $(u, u_1), (u_1, u_2), \ldots, (u_{k-1}, u_k), (u_k, v)$ ima najkraću udaljenost (cijenu, trajanje).

Problem - najkraći put



- Pronalaženje rješenja interpretiramo kao **niz odluka**.
- Odluke **nisu nezavisne**.

Problem - najkraći put

Koji grad bi trebalo posjetiti iz grada u?

- **Ne možemo** odlučiti **lokalno**, bez promatranja **cijelog** grafa!
- Ukoliko proširimo problem i promatramo najkraće puteve od u do **svih** vrhova u grafu, tada odluku možemo donijeti **pohlepno**.
- Promatramo: $min\{7 + d(u_1, v), 4 + d(u_2, v)\} = min\{9, 10\} = 9$
- Dakle, najkraći put je: $(u, u_1), (u_1, v)$

Problem ruksaka

Problem: Na raspolaganju nam je ruksak kapaciteta M (najveća težina koju možemo ponijeti ili volumen ruksaka). Ruksak želimo napuniti predmetima. Možemo birati između n različitih predmeta.

Ovisno o dostupnoj količini svakog predmeta i mogućnosti smještanja predmeta u ruksak (cijeli ili dijelova), postoji nekoliko varijanti problema:

1. Treba izabrati količinu x_i predmeta $i = 1, \ldots, n$ koje ćemo staviti u ruksak, tako da ne prepunimo ruksak, a da ukupni profit bude maksimalan. Količina x_i predmeta i ima težinu $w_i x_i$ i profit $p_i x_i$.

   Problem je **lako rješiv**. Promatramo koji predmet ima najveći profit po količini $\frac{p_i}{w_i}$ i tom vrstom potpuno ispunimo ruksak $x_i = \frac{M}{w_i}$.
2. Imamo dodatni uvjet: maksimalnu količinu svakog predmeta X_i. Dakle, $x_i \leq X_i$.

   Rješenje tog problema je također **jednostavno**, prvo stavimo predmet s najboljim omjerom $\frac{p_i}{w_i}$, zatim drugi po redu, itd. dok ne popunimo ruksak.

Problem ruksaka

3. Često nam treba i dodatni uvjet, da ne možemo *rezati* predmete, odnosno količine su nenegativni cijeli brojevi (Integer Knapsack).

   Rješenje navedenog problema je **puno teže**.

4. Standardna formulacija dodatno uvodi ograničenje da količine mogu imati vrijednost 0 ili 1 (0-1 Knapsack).

   Rješenje ovog problema je također teže od prva dva (ne postoji algoritam koji ga rješava polinomno **isključivo** prema broju predmeta n).

Odluke pri rješavanju problema:

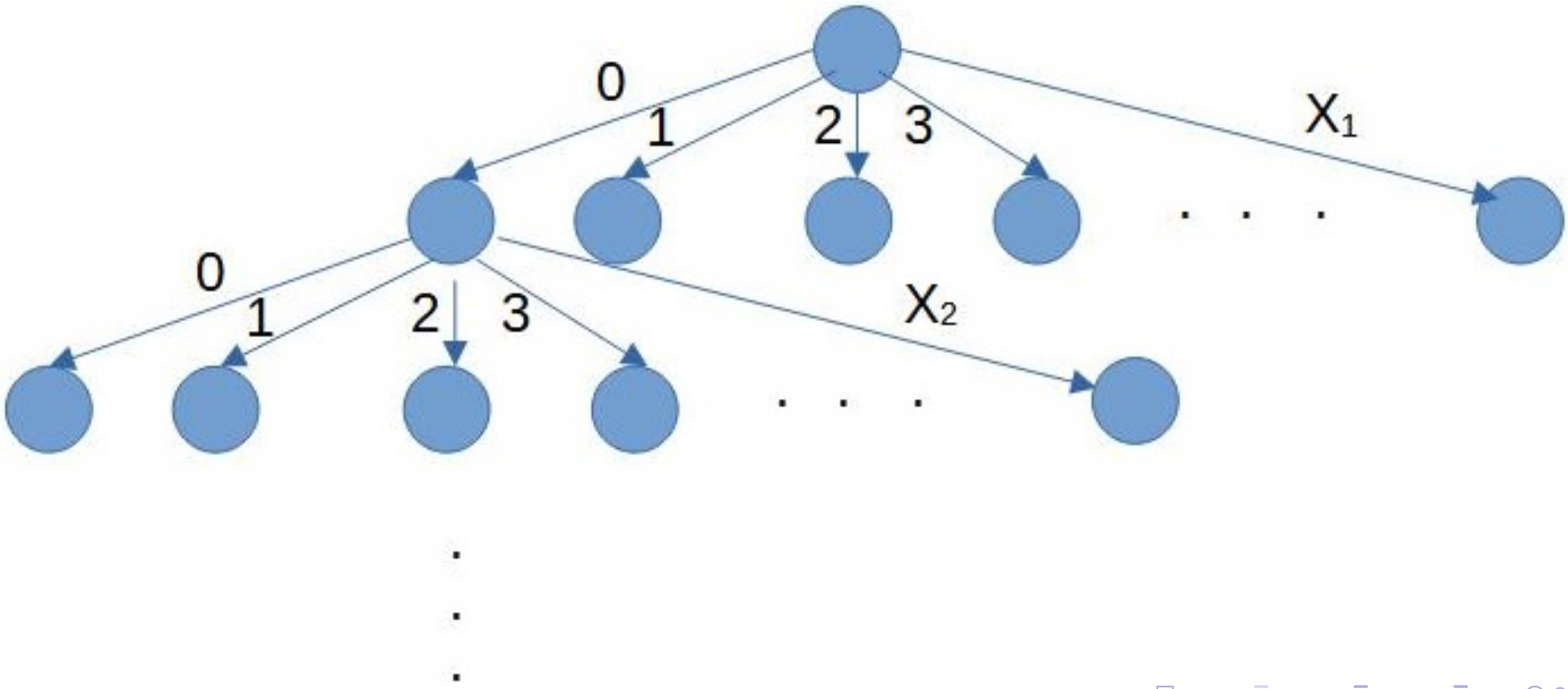
- izaberi količinu x_1
- izaberi količinu x_2
- ...

Problem ruksaka

Odnosno:

- ponesi x_1 ili ne
- ponesi x_2 ili ne
- ...

Postupak odlučivanja možemo prikazati stablom:



Problem ruksaka

Naivni algoritam bi morao pretražiti: $(X_1 + 1) \cdot (X_2 + 1) \cdot \cdots \cdot (X_n + 1)$ mogućnosti. U slučaju $0 - 1$ Knapsack problema, broj mogućnosti je 2^n.

Niti kod $0 - 1$ Knapsack problema **ne možemo** doći do rješenja promatrajući izbore **lokalno** (nezavisno od drugih izbora).

i	w_i	c_i
1	5	3
2	2	2
3	3	2

Za $M = 5$, izbor prvog elementa donosi profit 3, dok izbor elemenata 2 i 3 donosi profit 4. Međutim, za $M = 7$ bi bilo **pogrešno** ne uzeti prvi element.