Podatkovne strukture in algoritmi

Andrej Brodnik
UP FAMNIT

Uvod

uvod in matematične osnove

Izvajalci

predavanja: Andrej Brodnik, andrej.brodnik@upr.si

vaje: Rok Požar, rok.pozar@upr.si

e-viri:

- LMS: https://e.famnit.upr.si/

Literatura in viri

Literatura:

- **Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest in Clifford Stein: Introduction to Algorithms, McGraw-Hill**,
- Sedgewick: Algorithms in Java.
- *Open Data structures*.
- *A. Brodnik, R. Požar: Zbirka nalog*.

Opravljanje predmeta

Program:

- Prva polovica *podatkovne strukture* in druga polovica *algoritmi*.

Domače naloge:

- šest domačih nalog s po štirimi vprašanji, od katerih je eno programersko (empirično)
- programersko vprašanje na način kot pri ACM ICPC (`http://cm2prod.baylor.edu/`), oziroma UPM (`http://tekmovanja.acm.si/upm`)
- uporaba okolja za oddajo nalog *Marmoset*
- oddaja preko e-učilnice
- vsaj enkrat na leto zagovor teoretične domače naloge, kar se ocenjuje
- *vsako programersko vprašanje vsaj 20%* in skupna ocena DN *0,75 programerske + 0,25 zagovor*

Opravljanje predmeta – nadaljevanje

Izpit: lahko nadomestita kolokvija, vendar mora biti *skupna ocena pozitivna* in nobeden od kolokvijev *ne sme biti nižji od 40%*

Končna ocena:

- **30%:** sprotno delo (domače naloge)
- **60%:** končna ocena (izpit)
- **10%:** projekt

opravljeno vsaj 50%, a predavatelj lahko oceno dvigne v primeru nadpovprečne aktivnosti (npr. oddane domače naloge)

Vsebina predmeta – podatkovne strukture

- Osnove: modeliranje, matematika
- Številska drevesa
- Slovar: osnove in definicija, drevesa (osnove, iskalna, uravnotežena, B-drevesa, 2-3 drevesa, rdeče črna drevesa, rang in izbira), preskočni seznam, razpršene tabele in Bloomov filter.
- Vrste s prednostjo: osnove, dvojiška, binomska in Fibonaccijeva kopica.
- Disjunktne množice.

Vsebina predmeta – algoritmi

- Urejanje: osnove, deli in vladaj, končne množice
- Dinamično programiranje: Fibonaccijeva števila, množenje matrik, Needleman-Wunsch
- Algoritmi na grafih: sprehodi, najcenejše vpeto drevo, najkrajše poti

Algoritem

- Vsak dobro definiran računski postopek, s katerim kaj izračunamo ali rešimo kak problem, imenujemo *algoritem*.
  Algoritem vhodne podatke (*input*) spremeni v izhodne podatke (*output*).
- Na kaj moramo biti posebej pozorni:
  1. pravilnost,
  2. časovna zahtevnost (število korakov v odvisnosti od velikosti problema) in
  3. prostorska zahtevnost (v odvisnosti od velikosti problema)
- Algoritme – korake postopka – bomo opisavali v *psevdokodi*.
- Primer – urejanje.

  Vhod: Zaporedje n števil $(a_1, a_2, \ldots, a_n)$,
  Izhod: Permutacija $(a'_1, a'_2, \ldots, a'_n)$ vhodnega zaporedja, tako da velja $(a'_1 \leq a'_2 \leq \cdots \leq a'_n)$.

  Za zaporedje $(31, 41, 59, 26, 41, 58)$ naj algoritem vrne zaporedje $(26, 31, 41, 41, 58, 59)$.

Urejanje z vstavljanjem

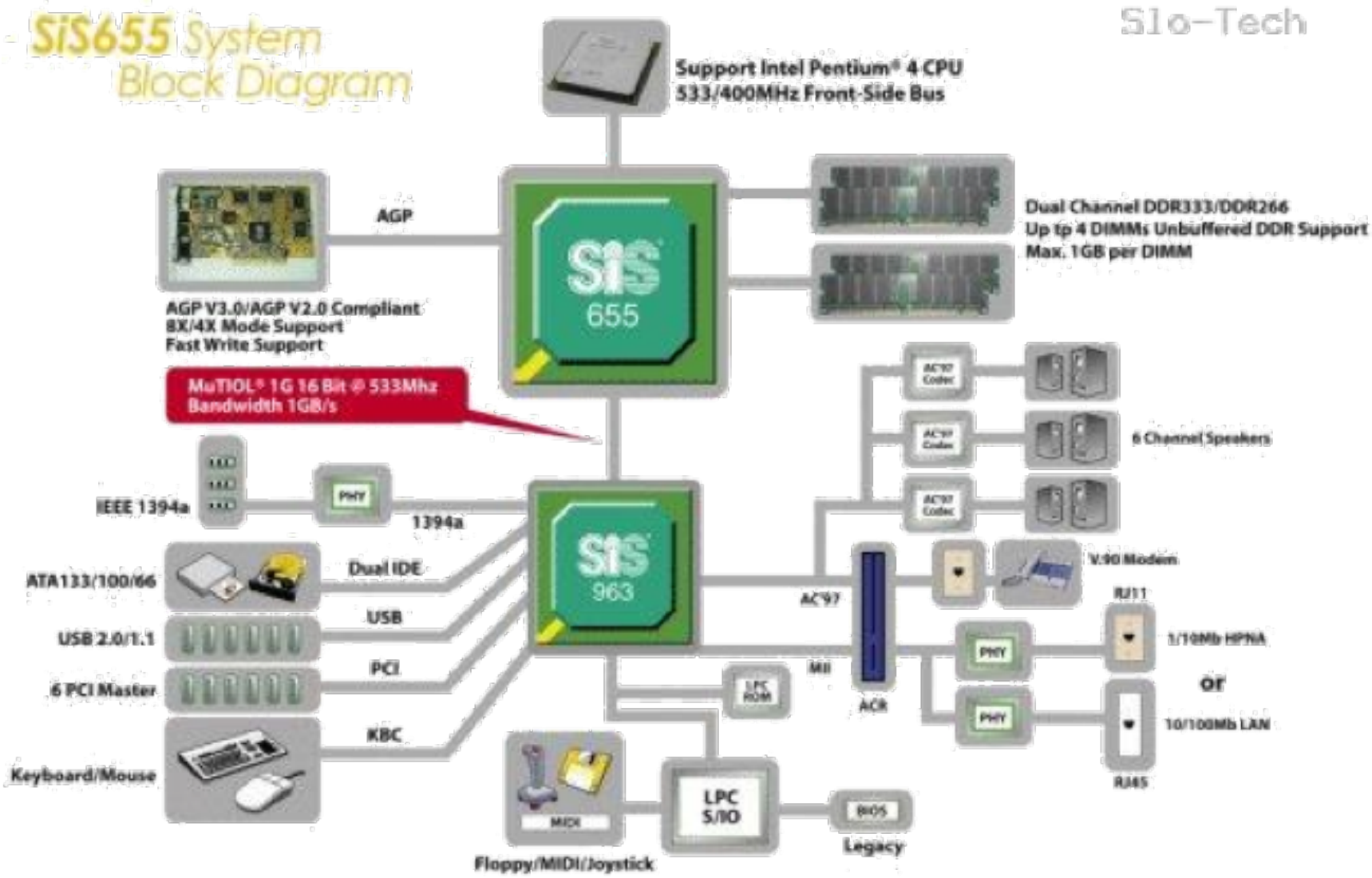
Eden izmed algoritmov za urejanje – *urejanje z vstavljanjem*:
INSERTION-SORT(A)

```
1: for j ← 2 to A.length do
2:     key ← A[j]
3:     i ← j − 1
4:     while i > 0 and A[i] > key do
5:         A[i + 1] ← A[i]
6:         i ← i − 1
7:     end while
8:     A[i + 1] ← key
9: end for
```

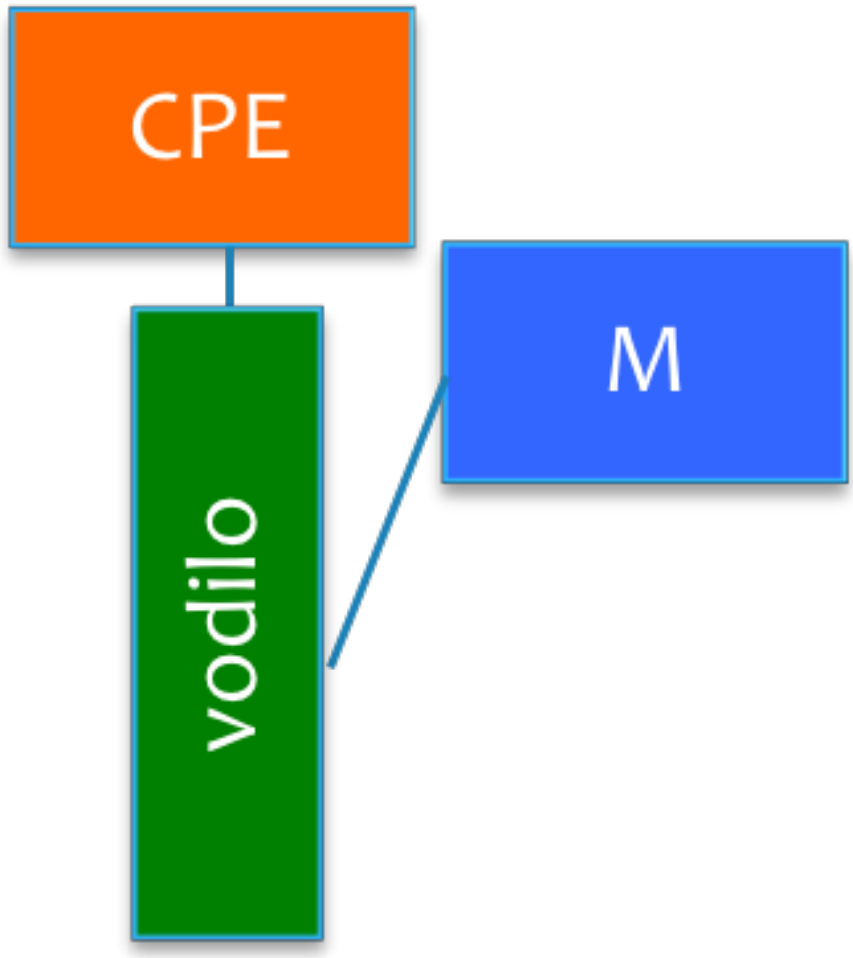
Štiri osnovna vprašanja

1. Ali pravilno deluje?
2. Kakšna je časovna zahtevnost?
3. Kakšna je prostorska zahtevnost?
4. Ali se da narediti bolje?

Kako izgleda računalnik



Von Neumannova arhitektura



Model računanja

- Za model računanja bomo vzeli *random-access machine (RAM)*, oziroma točneje *primerjalni model*.
  - Ukazi se izvajajo eden za drugim.
  - Predpostavljali bomo, da so vse operacije enako drage, čeprav to ne odraža dejanskega stanja, saj so množenja ter posebej še deljenja dražja.
- Obstajajo še drugi modeli, npr. *dostopni model*, kjer štejemo samo dostope do pomnilnika

Pravilnost delovanja

INSERTION-SORT(A)

```
1: for j ← 2 to A.length do
2:     key ← A[j]
3:     i ← j − 1
4:     while i > 0 and A[i] > key do
5:         A[i + 1] ← A[i]
6:         i ← i − 1
7:     end while
8:     A[i + 1] ← key
9: end for
```

Orodje: Invarianca – izjava o stanju spremenljivk v algoritmu, ki velja za vse ponovitve (iteracije).

Pravilnost delovanja – dokaz

Dokaz z indukcijo po zunanji zanki – po j:
Brez škode za splošnost predpostavimo, da so elementi, ki jih urejamo, med seboj različni.

Osnova: Če je samo en element (a_1), je polje že urejeno.

Hipoteza: Predpostavimo, da algoritem zna urediti elemente $(a_1, a_2, \ldots, a_{j-1})$ v $(a'_1 < a'_2 < \cdots < a'_{j-1})$.

Korak: Sedaj imamo polje elementov $(a_1, a_2, \ldots, a_j)$. Prvih $j-1$ elementov po predpostavki znamo urediti. v $(a'_1 < a'_2 < \cdots < a'_{j-1})$. Za zadnji element se notranja while izvaja dokler je `key` manjši. Ustavi se pri i-tem elementu. Ko vstavimo ključ na to mesto, kar se zgodi v koraku 8, so elementi $(a'_1 < a'_2 < \cdots < a'_i < \texttt{key})$ urejeni. Poleg tega velja, zaradi oblike WHILE zanke, da ostajajo elementi $(a'_{i+1} < a'_{i+2} < \cdots < a'_{j-i})$ urejeni, hkrati pa velja $(\texttt{key} < a'_{i+1}, a'_{i+2}, \ldots, a'_{j-i})$. Zatorej $(a'_1 < a'_2 < \cdots < a'_i < \texttt{key} < a'_{i+1} < a'_{i+2} < \cdots < a'_{j-i})$.
$\mathcal{QED}$

Časovna zahtevnost

- Naj bo $T_{\mathcal{A}}(n)$ število izvedenih osnovnih operacij ali ≫korakov≪ algoritma $\mathcal{A}$, pri vhodnih podatkih velikosti n. Funkciji $T_{\mathcal{A}}(n)$ rečemo *časovna zahtevnost ali čas izvajanja algoritma* $\mathcal{A}$.
- Predpostavljali bomo, da za izvajanje posameznega koraka potrebujemo konstantno časa, t.j. izvajanje i-te vrstice v vzame c_i časa.
- Izraz *velikost podatkov* je odvisen od problema, ki ga rešujemo.
  - V primeru urejanja je to kar *število elementov*, ki jih moramo urediti.
  - Pri množenju celih števil je to *število bitov* (števk), ki jih potrebujemo za dvojiški zapis vhodnih števil.

Časovna zahtevnost *urejanja z vstavljanjem*

INSERTION-SORT(A)	čas	#
1: **for** $j \leftarrow 2$ **to** A.length **do**	c_1	n
2: $key \leftarrow A[j]$	c_2	$n-1$
3: $i \leftarrow j-1$	c_3	$n-1$
4: **while** $i > 0$ and $A[i] > key$ **do**	c_4	$\sum_{j=2}^{n} \mathbf{t_j}$
5: $A[i+1] \leftarrow A[i]$	c_5	$\sum_{j=2}^{n} (\mathbf{t_j}-1)$
6: $i \leftarrow i-1$	c_6	$\sum_{j=2}^{n} (\mathbf{t_j}-1)$
7: **end while**		
8: $A[i+1] \leftarrow key$	c_8	$n-1$
9: **end for**		

(n je dolžina niza A, $\mathbf{t_j}$ naj bo število ponovitev zanke **while** pri danem j.)

Časovna zahtevnost *urejanja z vstavljanjem* – 2

Naj bo $T(n)$ čas izvajanja algoritma INSERTION-SORT. Velja

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_3(n-1) + c_4 \sum_{j=2}^{n} \mathbf{t_j} + c_5 \sum_{j=2}^{n} (\mathbf{t_j} - 1) \\ &\quad + c_6 \sum_{j=2}^{n} (\mathbf{t_j} - 1) + c_8(n-1) \end{aligned}$$

Funkcija $T(n)$ je odvisna od podatkov (od t_j, ta števila pa so odvisna od podatkov).
Izračunajmo $T(n)$ v dveh primerih: v *najboljšem* (podatki so že urejeni) in v *najslabšem primeru* (podatki so urejeni v obratnem vrstnem redu).

Časovna analiza – najboljši primer

V najboljšem primeru (podatki so že urejeni) velja $\mathbf{t_j} = 1$:

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_3(n-1) + c_4(n-1) + c_8(n-1) \\ &= (c_1 + c_2 + c_3 + c_4 + c_8)n - (c_2 + c_3 + c_4 + c_8) \\ &= an + b \end{aligned}$$

Torej je $T(n)$ linearna funkcija. Pravimo, da ima algoritem *v najboljšem primeru* **linearno časovno zahtevnost**.

Časovna analiza – najslabši primer

V najslabšem primeru (podatki so urejeni obratno) velja $\mathbf{t_j} = j$:

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_3(n-1) + c_4\left(\frac{n(n+1)}{2} - 1\right) + c_5\frac{n(n+1)}{2} + \\ &\quad c_6\frac{n(n+1)}{2} + c_8(n-1) \\ &= \left(\frac{c_4}{2} + \frac{c_5}{2} + \frac{c_6}{2}\right)n^2 + \left(c_1 + c_2 + c_3 + \frac{c_4}{2} + \frac{c_5}{2} + \frac{c_6}{2} + c_8\right)n - \\ &\quad (c_2 + c_3 + c_4 + c_8) \end{aligned}$$

Torej je $T(n)$ kvadratna funkcija. Pravimo, da ima algoritem *v najslabšem primeru* **kvadratno časovno zahtevnost**.

Ko rečemo, da *ima algoritem . . . časovno zahtevnost*, mislimo na časovno zahtevnost v NAJSLABŠEM PRIMERU.

Prostorska analiza

Algoritem potrebuje:

- spremenljivko $A[]$, v kateri se hranijo števila, ki jih urejamo: njena velikost je n – brez tega ne gre
- spremenljivki i in j, ki se uporabljata v zankah: njuna velikost je 2; spremenljivka `key`: size $\mathbf{1}$ – to je dodaten prostor

Ali se da bolje

Dve možni smeri iskanja odgovora:

1. poskusimo najti boljši algoritem – običajnejša, inženirska metoda
2. se vprašamo, koliko korakov v našem modelu nujno potrebuje KATERIKOLI algoritem za rešitev problema – težja a boljši rezultat, saj nam popiše ZAHTEVNOST PROBLEMA

Ocena zahtevnosti problema

RAZMISLEK:

- če imamo n elementov, obstaja $n!$ permutacij le-teh
- katerikoli algoritem za urejanje mora znati pretvoriti katerokoli permutacijo elementov v tisto pravo, urejeno permutacijo: z drugimi besedami, mora znati POISKATI vse permutacije
- vsa možna računanja algoritma si lahko predstavljamo kot drevo, kjer so:
  - listi z najdenimi permutacijami – jih je (vsaj) $n!$
  - notranja vozlišča so primerjave, ki jih naredi algoritem ter se nato odloči za nadaljevanje v levo ali desno poddrevo

  pri danem $A[]$ izvajanje algoritma sledi eni poti do lista
- višina drevesa primerjav je najkrajši »čas«, ki je potreben za urejanje in je $\lg n!$
- ker je (Stirling)

$$n! \approx \sqrt{2\pi n}\left(\frac{n}{e}\right)^n \cdot C$$

potem $\lg n! \approx n \lg n + n \cdot C_1$.

Ocena velikosti funkcije

Do sedaj smo srečali:

- časovna zahtevnost našega algoritma je: $a_2 n^2 + a_1 n + a_0$
- prostorska zahtevnost našega algoritma je: $n + 2$
- zahtevnost našega problema je: $n \lg n + n \cdot C_1$

Vrednost/velikost funkcij je odvisna od velikosti konstant, ki pa so odvisne od konkretnega računalnika, na katerem izvajamo naš algoritem.

Želimo dobiti mero, ki bo neodvisna od konkretnega računalnika in bo (na primer)

$$a_2 n^2 + a_1 n + a_0 \approx F(n^2)$$

ker je to največji člen. Definirali bomo DRUŽINO funkcij.

Ocena velikosti funkcij in *veliki O*

Za dano funkcijo $g(n)$ označimo z $O(g(n))$ *množico* funkcij

$$O(g(n)) = \left\{ f(n) \mid \begin{array}{l} \text{obstajata pozitivni konstanti } c \text{ in } n_0, \\ \text{tako da } f(n) \leq cg(n) \text{ za vse } n \geq n_0. \end{array} \right.$$

Če je $h(n) = O(g(n))$ potem rečemo, da $g(n)$ *asimptotično od zgoraj omejuje* $h(n)$.

Opomba: predpostavljamo, da so vse funkcije nenegativne.

Pravzaprav bi morali zapisati $h(n) \in O(g(n))$, toda držali se bomo ustaljene (zlo)rabe!

Ocena velikosti funkcij in *veliki O* – primer

- $3n = O(n)$, $n = O(n^2)$.
- INSERTION-SORT za urejanje potrebuje $O(n^2)$ *primerjav*, kjer je $n = |A[]|$.

INSERTION-SORT(A)

```
1: for j ← 2 to A.length do
2:     key ← A[j]
3:     i ← j − 1
4:     while i > 0 and A[i] > key do
5:         A[i + 1] ← A[i]
6:         i ← i − 1
7:     end while
8:     A[i + 1] ← key
9: end for
```

Ocena velikosti funkcij in *velika* Ω

Za dano funkcijo $g(n)$ označimo z $\Omega(g(n))$ *množico* funkcij

$$\Omega(g(n)) = \left\{ f(n) \mid \begin{array}{l} \text{obstajata pozitivni konstanti } c \text{ in } n_0, \\ \text{tako da } f(n) \geq cg(n) \text{ za vse } n \geq n_0. \end{array} \right.$$

Če je $h(n) = \Omega(g(n))$ potem rečemo, da je $g(n)$ *asimptotično od spodaj omejuje* $h(n)$.

Časovna zahtevnost *problema* urejanja je $\Omega(n \log n)$; t.j. *katerikoli* algoritem za urejanje potrebuje *vsaj* $\gg n \log n \ll$ primerjav

Ocena velikosti funkcij in *velika* Θ

Za dano funkcijo $g(n)$ označimo z $\Theta(g(n))$ *množico* funkcij

$$\Theta(g(n)) = \left\{ f(n) \mid \begin{array}{l} \text{obstajajo pozitivne konstante } c_1,\ c_2 \text{ in } n_0, \\ \text{tako da } f(n) \geq c_1 g(n) \text{ in } f(n) \leq c_2 g(n) \\ \text{za vse } n \geq n_0. \end{array} \right.$$

kar je enakovredno

$$(f(n) = O(g(n))) \ \ \& \ \ (f(n) = \Omega(g(n))) \equiv (f(n) = \Theta(g(n)))$$

Dokažite to!

Če je $h(n) = \Theta(g(n))$ potem rečemo, da je $h(n)$ *asimptotično omejena z* $g(n)$.

Ocena velikosti funkcij in *velika* Θ – primer

- $\frac{1}{2}n^2 - 3n = \Theta(n^2)$
- Ko bomo spoznali algoritme za urejanje, ki imajo časovno zahtevnost $O(n \log n)$, bomo zapisali, da imajo časovno zahtevnost v resnici $\Theta(n \log n)$.

Zakaj?

Ocena velikosti funkcij in *mali o*

Za dano funkcijo $g(n)$ označimo z $o(g(n))$ množico funkcij

$$o(g(n)) = \left\{ f(n) \mid \begin{array}{l} \text{za poljubno konstanto } c > 0 \text{ obstaja} \\ \text{konstanta } n_0 > 0, \text{ tako da } f(n) < cg(n) \text{ za vse } n \geq n_0. \end{array} \right.$$

Intuitivni pomen: če je $f(n) = o(g(n))$, potem je $f(n)$ vedno manjša od $g(n)$, ko n večamo čez vse meje.
Enakovredna definicija:

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} = 0 \ .$$

Primer: $2n = o(n^2)$, $2n^2 \neq o(n^2)$, $7n = o(n^{1+\epsilon})$.

Ocena velikosti funkcij in *mali* ω

Za dano funkcijo $g(n)$ označimo z $\omega(g(n))$ množico funkcij

$$\omega(g(n)) = \left\{ f(n) \mid \begin{array}{l} \text{za poljubno konstanto } c > 0 \text{ obstaja} \\ \text{konstanta } n_0 > 0, \text{ tako da } f(n) > cg(n) \text{ za vse } n \geq n_0. \end{array} \right.$$

Intuitivni pomen: če je $f(n) = \omega(g(n))$, potem je $f(n)$ vedno večja od $g(n)$, ko n večamo čez vse meje
Enakovredna definicija:

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} = \infty \ .$$

Primer: $n^2/2 = \omega(n)$, $n^2/2 \neq \omega(n^2)$, $7n = \omega(n^{1-\epsilon})$.

Ocena velikosti funkcij – intuitivni pomen

$$
\begin{array}{lll}
f(n) = & \omega(g(n)) & \approx f(n) > g(n) \\
f(n) = & \Omega(g(n)) & \approx f(n) \geq g(n) \\
f(n) = & \Theta(g(n)) & \approx f(n) = g(n) \\
f(n) = & O(g(n)) & \approx f(n) \leq g(n) \\
f(n) = & o(g(n)) & \approx f(n) < g(n)
\end{array}
$$

Štiri osnovna vprašanja o urejanju z vstavljanjem

1. Ali pravilno deluje?
   Da, smo dokazali z indukcijo.
2. Kakšna je časovna zahtevnost?
   Z izračunom smo pokazali, da je $O(n^2)$.
3. Kakšna je prostorska zahtevnost?
   Ker razen polja potrebujemo samo še dve spremenljivki (i *in* j*) in* `key` *je* $O(n)$*, oziroma* $O(1)$ *dodatnega prostora.*
4. Ali se da narediti bolje?
   Morda. Vemo, da potrebujemo vsaj $\Omega(n \log n)$ *primerjav kar zadeva časa. Prostor* $\Theta(n)$ *je najboljši, ker polje moramo vedno imeti.*

Še en algoritem

X-FUNCTION(p, r)

1: **if** $p < r$ **then**
2: $\quad q \leftarrow \lfloor (p + r)/2 \rfloor$
3: $\quad$ X-FUNCTION(p, q)
4: $\quad$ X-FUNCTION$(q + 1, r)$
5: $\quad$ Y-FUNCTION(p, q, r)
6: **end if**

Če je $r - p = n$ časovna zahtevnost funkcije Y-FUNCTION(p, q, r), kakšna je potem časovna zahtevnost $T(n)$ funkcije X-FUNCTION$(1, n)$?

Očitno velja:

$$T(n) = \begin{cases} \Theta(1) & n = 1, \\ 2T(n/2) + \Theta(n) & n > 1. \end{cases}$$

In $T(n)$ je potem koliko?

Glavni izrek – splošno pravilo

Izrek. Naj bosta a, b konstanti, $a \geq 1$, $b > 1$, naj bo $f(n)$ funkcija in naj bo $T(n)$ funkcija definirana na nenegativnih celih številih, ki zadošča enačbi

$$T(n) = aT(n/b) + f(n) \ ,$$

kjer interpretiramo n/b kot $\lfloor n/b \rfloor$ ali $\lceil n/b \rceil$.
Potem velja:

1. Če je $f(n) = O(n^{\log_b a - \epsilon})$ za nek $\epsilon > 0$, potem $T(n) = \Theta(n^{\log_b a})$.
2. Če je $f(n) = \Theta(n^{\log_b a})$, potem $T(n) = \Theta(n^{\log_b a} \log n)$.
3. Če je $f(n) = \Omega(n^{\log_b a + \epsilon})$ za nek $\epsilon > 0$ in, če velja $af(n/b) \leq cf(n)$ za neko konstanto $c < 1$ in vse dovolj velike n, potem $T(n) = \Theta(f(n))$.

Glej Cormen, *Master theorem*.

Analiza X-Function

X-Function(p, r)

1: **if** $p < r$ **then**
2: $\quad q \leftarrow \lfloor (p + r)/2 \rfloor$
3: $\quad$ X-Function(p, q)
4: $\quad$ X-Function$(q + 1, r)$
5: $\quad$ Y-Function(p, q, r)
6: **end if**

$$
\begin{aligned}
T(n) &= \begin{cases} \Theta(1) & n = 1, \\ 2T(n/2) + cn & n > 1. \end{cases} \\
&= aT(n/b) + f(n) \qquad f(n) = \Theta(n^{\log_b a}) \\
&= 2T(n/2) + n \qquad a = 2, b = 2, \Theta(n^{\log_2 2}) = \Theta(n) \\
&= \Theta(n \log n)
\end{aligned}
$$