Algoritmizace

Cvičení

2025

Obsah

Kapitola 1

Elementární funkce

Lineární funkce

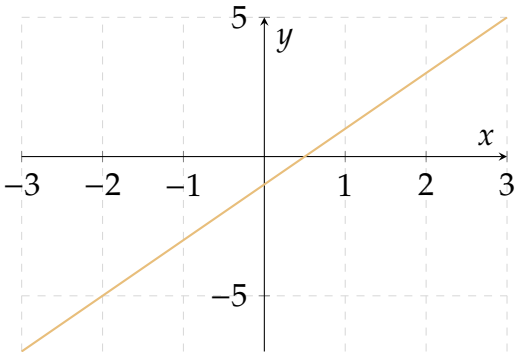
Definice

Lineární funkce má tvar

$$f(x) = ax + b, \quad a \neq 0.$$

Vlastnosti:

- definiční obor: $\mathbb{R}$
- obor hodnot: $\mathbb{R}$
- monotónnost: rostoucí pro $a > 0$, klesající pro $a < 0$
- nemá extrémy, není omezená



Kvadratická funkce

Definice

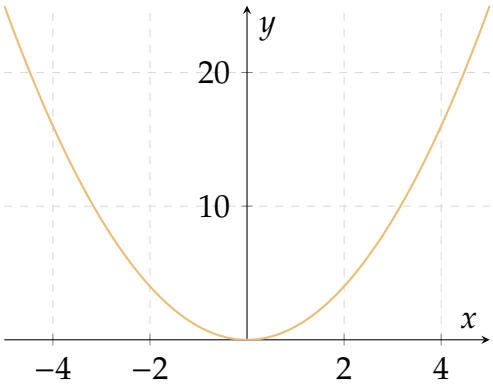
Kvadratická funkce má tvar

$$f(x) = ax^2 + bx + c, \quad a \neq 0.$$

Vlastnosti:

- definiční obor: $\mathbb{R}$

- obor hodnot (pro $f(x) = x^2$): $\langle 0, +\infty)$
- na intervalu $(-\infty, 0]$ je klesající, na $[0, +\infty)$ rostoucí
- zdola omezená



Obecně můžeme uvažovat polynomy. Kvadratická i lineární funkce jsou speciálními případy polynomických funkcí.

Goniometrické funkce

Definice

Základní goniometrické funkce jsou sinus

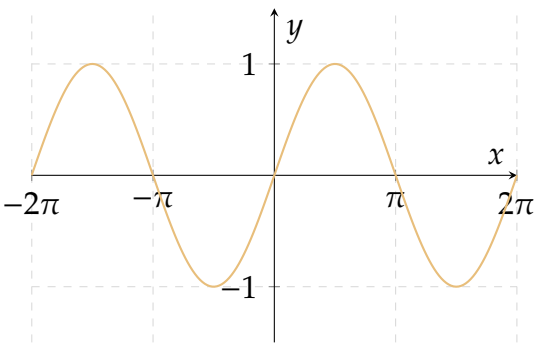
$$f(x) = \sin x,$$

a cosinus

$$f(x) = \cos x.$$

Vlastnosti sinu:

- definiční obor: $\mathbb{R}$
- obor hodnot: $\langle -1, 1 \rangle$
- periodická s periodou 2π
- není monotónní na celém oboru, ale je monotónní na dílčích intervalech



Exponenciální funkce

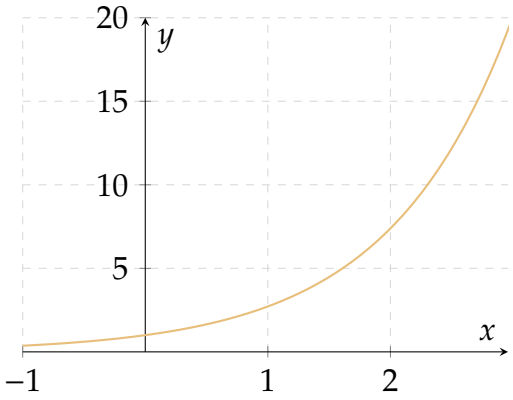
Definice

Exponenciální funkce je dána předpisem

$$f(x) = a^x, \quad a > 0,\ a \neq 1.$$

Vlastnosti:

- definiční obor: $\mathbb{R}$
- obor hodnot: $(0, \infty)$
- monotónnost: rostoucí pro $a > 1$, klesající pro $0 < a < 1$



Logaritmická funkce

Definice

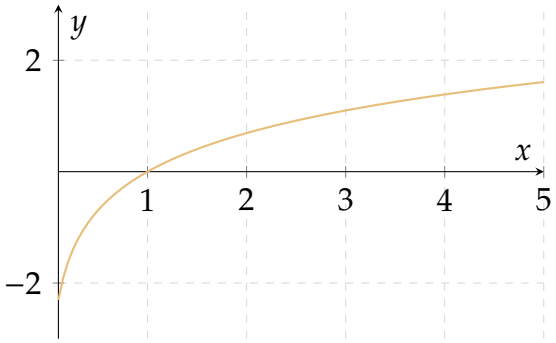
Logaritmická funkce je inverzní k exponenciální funkci:

$$f(x) = \log_a(x), \quad a > 0,\ a \neq 1.$$

Platí: $a^y = x$ právě tehdy, když $\log_a(x) = y$.

Vlastnosti:

- definiční obor: $(0, \infty)$
- obor hodnot: $\mathbb{R}$
- monotónnost: rostoucí
- nemá extrémy



Faktoriál

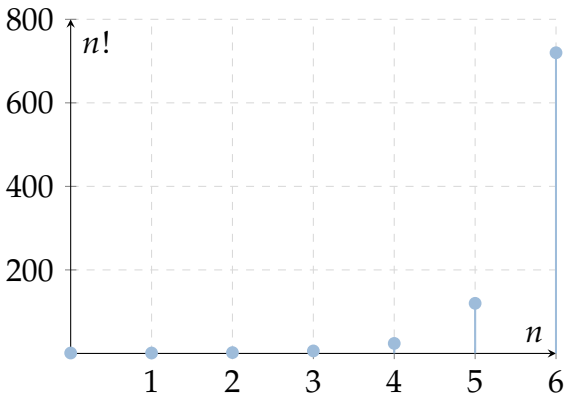
Definice

Faktoriál je definován pro $n \in \mathbb{N}_0$ předpisem

$$n! = 1 \cdot 2 \cdot 3 \cdot \cdots \cdot n, \quad 0! = 1.$$

Vlastnosti:

- definiční obor: $\mathbb{N}_0$
- obor hodnot: $\mathbb{N}$
- monotónnost: rostoucí posloupnost
- není omezený, nemá extrémy



Kapitola 2

Posloupnosti

Definice

Posloupnost je zobrazení z množiny přirozených čísel do libovolné množiny. Zápis: (a_n) – posloupnost, a_n – n-tý prvek.

Často za množinu A budeme dosazovat $\mathbb{R}$, takovým posloupnostem se říká *číselné posloupnosti*. Ty budou důležité pro tuto kapitolu. O posloupnosti $(a_n) : \mathbb{N} \to A$ můžeme uvažovat tak, že každému prvku z A lze přiřadit index.

2.1 Způsoby zadávání posloupností

2.1.1 Prvních x členů

Musí být jasné pravidlo, jak vyjádřit další členy posloupnosti.

Příklad

- triviální příklad $1, 2, 3, 4...$
- fibbonacci $0, 1, 1, 2, 3, 5, 8...$
- alternující $1, -1, 1, -1...$
- konečná $2, 4, 6, ..., 20$

2.1.2 Vzorec pro n-tý člen

Vyjádříme obecný vzorec pro n-tý prvek na základě indexu.

Příklad

- $(\frac{n}{n+1})$
- $((-1)^n n)$
- $(1 + \frac{1}{n})^n)$

2.1.3 Rekurentně

Rekurentní zadání obsahuje zpravidla 1. člen (nebo několik prvních členů) a pravidlo, jak vytvořit další člen ze členů předcházejících.

Příklad

- $(\frac{n}{n+1})$
- $((-1)^n n)$
- $(1+\frac{1}{n})^n)$

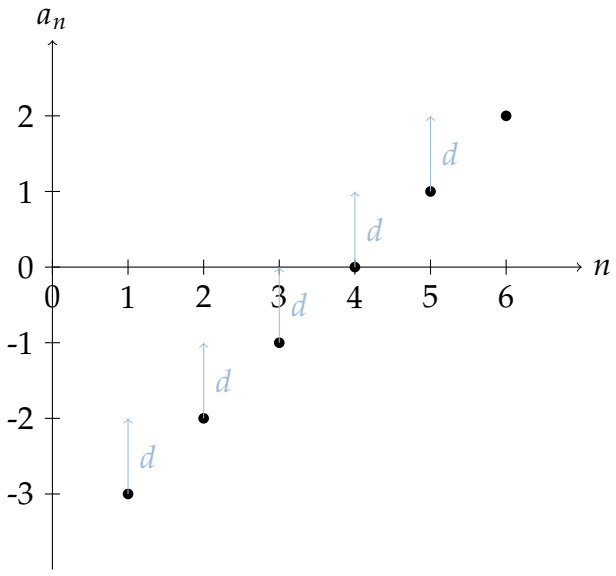
Speciálním případem rekurentního zadání jsou aritmetická a geometrická posloupnost.

Definice

Posloupnost (a_n) se nazývá **aritmetická** právě tehdy, když

$$\exists d \in \mathbb{R}\ \forall n \in \mathbb{N} \quad a_{n+1} = a_n + d$$

Číslo d se nazývá **diference** aritmetické posloupnosti.



Definice

Posloupnost (a_n) se nazývá **geometrická** právě tehdy, když

$$\exists q \in \mathbb{R}\ \forall n \in \mathbb{N} \quad a_{n+1} = a_n \cdot q$$

Číslo q se nazývá **kvocient** aritmetické posloupnosti.

2.2 Vlastnosti posloupností

Definice

Posloupnost (a_n) nazveme

- *rostoucí*, jestliže $a_{n+1} > a_n$ pro všechna $n \in \mathbb{N}$,
- *nerostoucí*, jestliže $a_{n+1} \leq a_n$ pro všechna $n \in \mathbb{N}$,
- *klesající*, jestliže $a_{n+1} < a_n$ pro všechna $n \in \mathbb{N}$,
- *neklesající*, jestliže $a_{n+1} \geq a_n$ pro všechna $n \in \mathbb{N}$.

Definice

Posloupnost $\{a_n\}$ je *omezená*, jestliže existuje reálné číslo $M > 0$ tak, že

$$|a_n| \leq M \quad \text{pro všechna } n \in \mathbb{N}.$$

Ekvivalentně, posloupnost je omezená shora, pokud $\exists K \in \mathbb{R}$ tak, že $a_n \leq K$ pro všechna n; omezená zdola, pokud $\exists L \in \mathbb{R}$ tak, že $a_n \geq L$ pro všechna n.

Definice

Pro posloupnost $\{a_n\}$ definujeme:

- *maximum* $\max\{a_n\}$, pokud existuje člen a_k, pro který platí $a_k \geq a_n$ pro všechna n,
- *minimum* $\min\{a_n\}$, pokud existuje člen a_k, pro který platí $a_k \leq a_n$ pro všechna n.

2.3 Úkoly

1. Určete vzorec pro n-tý člen následujících posloupností:
   - $3, 7, 11, 15, 19\ldots$
   - $2, 6, 18, 54, 162, 486\ldots$
   - $2, 6, 12, 20, 30\ldots$
   - $1, -2, 3, -4, 5, -6\ldots$
   - $\frac{1}{1\cdot 4}, \quad \frac{3}{4\cdot 7}, \quad \frac{5}{7\cdot 10}, \quad \frac{7}{10\cdot 13}, \quad \ldots$
2. Vypočítejte prvních 5 prvků posloupností daných vzorcem:
   - $a_n = 2 \cdot 4^{n-1}$
   - $a_n = n^2 + 2n + 1$
   - $a_n = (-2)^n \cdot n$
   - $a_n = \frac{n^2+n}{2}$

Kapitola 3

Pseudokód

Ze slidů prof. Bělohlávka:

3.1 Náznak definice

Definice

První přiblížení („definice“): Algoritmus je posloupnost instrukcí pro řešení problému.
Tato „definice“ je však nepřesná (tedy vlastně ani nejde o přesnou definici):

- Co je to problém?
- Co je to instrukce?
- Co znamená „řešit problém“?

Názorně ale vystihuje podstatu pojmu algoritmus.

Budeme se zabývat zejména algoritmy, které jsou určeny pro počítač (tj. instrukce vykonává počítač). Základní způsoby popisu takových algoritmů jsou:

- **Přirozeným jazykem** (např. popis receptu).
  - \+ Snadno srozumitelné i laikům.
  - – Může být nejednoznačné nebo příliš zdlouhavé.
- **Programovacím jazykem**. Budeme používat jazyk Python (ale lze použít libovolný jazyk).
  - \+ Jednoznačné.
  - \+ Snadno lze vytvořit počítačový program.
  - – Obsahuje i nepodstatné detaily. Často příliš dlouhé.
- **Pseudokódem**. Pseudokód je jazyk podobný programovacímu jazyku, ale úspornější – neobsahuje zbytečné detaily. Použijeme pseudokód z knihy *Cormen et al.: Introduction to Algorithms, 2nd Ed., MIT Press, 2001*.

+ Snadno srozumitelný i pro ty, kteří nemají zkušenost s programováním.
+ Úsporně a přehledně popisuje algoritmy.
+ Umožňuje snadný přepis do programovacích jazyků.
– Je třeba ho převést do konkrétního programovacího jazyka při implementaci.

- **Dalšími (polo)formálními prostředky**, např. vývojovými diagramy.

3.2 Základní prvky pseudokódu (podle Cormen et al.)

Přiřazení

$x \leftarrow y$

Podmínka

```
1: if podmínka then
2:     příkazy
3: else
4:     příkazy
```

Cyklus `for`

```
1: for i ← 1 to n do
2:     příkazy
```

Cyklus `while`

```
1: while podmínka do
2:     příkazy
```

Cyklus `repeat-until`

```
1: repeat
2:     příkazy
3: until podmínka
```

Volání funkce

$y \leftarrow \textsc{Funkce}(x)$

Návratová hodnota

return hodnota

Přístup k poli

$A[i]$

3.3 Základní otázky o algoritmech

3.3.1 Správnost

Vrátí algoritmus správný výsledek pro každý přípustný vstup?

Algorithm 1 Algoritmus pro nalezení maxima pole

Require: Pole A délky n obsahující celá čísla
Ensure: Maximální prvek v poli

```
1: max ← 0
2: for i = 1 to n − 1 do
3:     if A[i] > max then
4:         max ← A[i]
5: return max
```

Co takový algoritmus? Je správný? Pokud chceme ukázat, že algoritmus správný není stačí nám najít vstup pro který algoritmus nenajde správný výstup.Zkusme tedy pole [1, 2, 3] maximum by mělo být 3. Po krokování algoritmu zjistíme, že tomu tak je. Co vstup [-1, -3, -5]? Algoritmus vrátí 0. Nula není největší prvek pole. Dokonce do něj vůbec nepatří.

Musíme si dávat pozor. Často se správnost algoritmu dokazuje aby nešlo rozporovat, že pracuje správně. Ukažme snadný důkaz správnosti následujícího algoritmu.

Algorithm 2 Lineární vyhledávání

Require: Pole A délky n, hledaná hodnota x
Ensure: Index prvku x nebo NIL, pokud se nenachází

```
1: for i = 0 to n − 1 do
2:     if A[i] = x then
3:         return i
4: return NIL
```

Důkaz

Chceme dokázat, že algoritmus vrátí správný výsledek a že vždy skončí.
Na začátku každé iterace s indexem i platí indukční předpoklad: „Prvek x není v $A[0\,..\,i-1]$.“

- Pro $i = 0$ je tvrzení triviálně pravdivé.
- Pokud algoritmus v iteraci i zjistí, že $A[i] = x$, vrátí správně index.
- Pokud $A[i] \neq x$, je tvrzení splněno i pro další iteraci.

Cyklus skončí po n iteracích. Pokud se x v poli nenachází, algoritmus vrátí NIL, což je správně.

3.3.2 Složitost

- časová složitost,

- paměťová složitost.

Časová složitost

Popisuje, jak rychle algoritmus pracuje – tedy závislost počtu vykonaných kroků na velikosti vstupu. Tuto závislost můžeme vyjádřit funkcí

$$T : \mathbb{N} \to \mathbb{N},$$

která velikosti vstupu přiřadí počet kroků provedených algoritmem.

Složitosti se budeme podrobně věnovat v další kapitole.

3.3.3 Optimalita

Je náš algoritmus nejlepší možný vzhledem ke složitosti? Neexistuje rychlejší varianta?

Algorithm 3 Největší společný dělitel — naivní verze

Require: Celá čísla m, n
Ensure: Největší společný dělitel čísel m, n
  $t \leftarrow \min(m, n)$
  while $m \bmod t \neq 0$ **or** $n \bmod t \neq 0$ **do**
    $t \leftarrow t - 1$
  return t

Správnost je zřejmá, ale algoritmus je velmi pomalý. Existuje mnohem rychlejší varianta – **Eukleidův algoritmus**.

3.4 Úkoly

1. Navrhněte v pseudokódu algoritmus, který:
   - IN: číslo x, OUT: jeho ciferný součet
   - IN: číslo x, OUT: součet jeho dělitelů
   - IN: číslo x, OUT: rozhodne, zda je prvočíslem
   - IN: číslo x, OUT: počet jeho dělitelů
   - IN: pole čísel A, OUT: pole s prvky v opačném pořadí
   - IN: pole čísel A, OUT: nejmenší a největší prvek v jednom průchodu
   - IN: pole čísel A, OUT: součet všech prvků
   - IN: pole čísel A, OUT: počet prvků větších než průměr pole
   - IN: pole čísel A, OUT: pole A bez duplicit
   - IN: pole čísel A, OUT: pole, kde jsou všechny nuly přesunuty na konec (bez změny pořadí ostatních prvků)
   - IN: pole čísel A a číslo x, OUT: index prvního výskytu
   - IN: pole čísel A a číslo x, OUT: pole všech indexů, kde se nachází

Kapitola 4

Složitost

Analýza algoritmu - snažíme se odhadnout jak se algoritmus bude chovat. Pokud mluvíme o složitosti jde nám o to kolik algoritmu zabere času než nám dá výsledek. Nebo můžeme uvažovat paměťovou složitost – kolik paměti algoritmus bude potřebovat na výpočet.

pokud budeme dále mluvit o složitosti algoritmu. Budeme tím myslet časovou, pokud nebude explicitně řečeno jinak.

Definice

Časová složitost algoritmu A je funkce $T_A : \mathbb{N} \to \mathbb{N}$ $T_A(n)$ = počet elementárních kroků, které A vykoná pro vstup velikosti n.

Funkce $T_A(n)$ popisuje jak se algoritmus chová pro všechny vstupy. Obvykle nás zajímá nejhorší případ, nebo průměrný případ.

Nechť algoritmus A řeší problém P a nechť $I_1, I_2, \ldots, I_m$ jsou všechny možné vstupy problému P o velikosti n. Označme $t_A(I_i)$ dobu běhu algoritmu A na instanci I_i.

Definice

Časová složitost v nejhorším případě (worst-case time complexity) je definována jako maximální doba běhu algoritmu pro všechny možné instance o velikosti n:

$$T_A^{\max}(n) = max\{t_A(I) | \text{I je vstup problému P a} |I| = n\}$$

tj. graficky:

$$\left.\begin{array}{c|c} I_1 & t_A(I_1) \\ I_2 & t_A(I_2) \\ \vdots & \vdots \\ I_m & t_A(I_m) \end{array}\right\} \text{max.}$$

Definice

Časová složitost v průměrném případě (average-case time complexity) je definována jako průměrná doba běhu algoritmu:

$$T_A^{\text{avg}}(n) = \frac{t_A(I_1) + \cdots + t_A(I_m)}{m} \qquad \text{tj. graficky:} \qquad \left.\begin{array}{c|c} I_1 & t_A(I_1) \\ I_2 & t_A(I_2) \\ \vdots & \vdots \\ I_m & t_A(I_m) \end{array}\right\} \mathbb{E}$$

kde p_i je pravděpodobnost výskytu instance I_i.

Přehled běžných typů složitostí

Typ růstu	Příklad funkce	Typické algoritmy / operace
konstantní	c	přístup k prvku v poli
logaritmická	$\log n$	binární vyhledávání
lineární	n	prohledání pole, výpočet minima
lineárně–logaritmická	$n \log n$	efektivní třídění (MergeSort, QuickSort)
kvadratická	n^2	naivní třídění (BubbleSort, InsertionSort)
kubická	n^3	naivní násobení matic
exponenciální	2^n	úplné prohledávání stavového prostoru
faktoriál	$n!$	procházení všech permutací

4.1 Úkoly

1. U následujících algoritmů určete časovou složitost

Algorithm 4 Faktoriál čísla n

```
function Faktoriál(n)
    if n = 0 then
        return 1
    else
        return n · Faktoriál(n − 1)
end function
```

Algorithm 5 Eukleidův algoritmus

```
function NSD(a, b)
    while b ≠ 0 do
        r ← a mod b
        a ← b
        b ← r
    return a
end function
```

Algorithm 6 Výpočet n-tého Fibonacciho čísla

```
function FIBONACCI(n)
    if n ≤ 1 then
        return n
    prev ← 0
    curr ← 1
    for i ← 2 to n do
        new ← prev + curr
        prev ← curr
        curr ← new
    return curr
end function
```

Algorithm 7 Rychlé mocnění

```
function MOCNINA(x, n)
    if n = 0 then
        return 1
    else if n je sudé then
        y ← MOCNINA(x, n/2)
        return y ··· y
    else
        return x· MOCNINA(x, n − 1)
end function
```

4.2 Růst základních funkcí

n	$\log_2 n$	n	$n \log_2 n$	n^2	n^3	2^n	$n!$
10	3.3	10	33	100	1000	1024	3,628,800
100	6.6	100	660	10^4	10^6	$1.27 \cdot 10^{30}$	$9.33 \cdot 10^{157}$
10^3	10	10^3	10^4	10^6	10^9	$1.07 \cdot 10^{301}$	$4.02 \cdot 10^{2567}$
10^4	13	10^4	$1.3 \cdot 10^5$	10^8	10^{12}	$1.99 \cdot 10^{3010}$	$2.85 \cdot 10^{35659}$
10^5	17	10^5	$1.7 \cdot 10^6$	10^{10}	10^{15}	–	–
10^6	20	10^6	$2 \cdot 10^7$	10^{12}	10^{18}	–	–

Tabulka 4.1: Přibližné hodnoty základních funkcí.

n	$\log_2 n$	n	$n \log_2 n$	n^2	n^3	2^n	$n!$
10	0	0	0	0	0	0	0
100	0	0	0	0	0	$1.27 \cdot 10^{15}$	$9.33 \cdot 10^{142}$
10^3	0	0	0	0	0	$1.07 \cdot 10^{286}$	$4.02 \cdot 10^{2552}$
10^4	0	0	0	0	0.001	$1.99 \cdot 10^{2995}$	$2.85 \cdot 10^{35644}$
10^5	0	0	0	10^{-5}	1	–	–
10^6	0	0	0	0.001	1000	–	–

Tabulka 4.2: Doba výpočtu v sekundách na velmi rychlém počítači (10^{15} operací za sekundu).

Kapitola 5

Třídění

Definice

Problém třídění:
Vstup: posloupnost n čísel: $\langle a_1, a_2 \dots a_n \rangle$
Výstup: permutace $\langle b_1, b_2 \dots b_n \rangle$ tak, že platí $b_1 \leq b_2 \leq \cdots \leq b_n$
vstup i výstup obvykle reprezentujeme polem.

Definice

Algoritmus pracuje *na místě* pokud použije nanejvýš konstantní prostor mimo tříděné pole.

5.1 Insertion Sort

Idea tohoto algoritmu je podobná způsobu, jak třídíme n rozdaných karet: n karet leží na začátku na stole. Pravou rukou je bereme a vkládáme do levé ruky tak, že v levé ruce vzniká setříděná posloupnost karet (zleva od nejmenší po největší). Drží-li levá ruka k karet, pak další, $(k+1)$-ní, kartu zatřídíme tak, že ji zprava porovnáváme se setříděnými kartami a vložíme ji na správné místo.

Algorithm 8 Insertion-Sort($A[0..n-1]$)

```
1: for j ← 1 to n − 1 do
2:     t ← A[j]
3:     i ← j − 1
4:     while i ≥ 0 and A[i] > t do
5:         A[i + 1] ← A[i]
6:         i ← i − 1
7:     A[i + 1] ← t
```

5.2 Selection Sort

Idea: Najdi v $A[0..n-1]$ nejmenší prvek a vyměň ho s $A[0]$. Najdi v $A[1..n-1]$ nejmenší prvek a vyměň ho s $A[1]$. ... Najdi v $A[n-2..n-1]$ nejmenší prvek a vyměň ho s

$A[n-2]$.

Algorithm 9 Selection-Sort($A[0..n-1]$)

```
1: for j ← 0 to n − 2 do
2:     iMin ← j
3:     for i ← j + 1 to n − 1 do
4:         if A[i] < A[iMin] then
5:             iMin ← i
6:     t ← A[j]; A[j] ← A[iMin]; A[iMin] ← t
```

5.3 Úkoly

1. Nasimulujte si algoritmy selection sort a insertion sort krok po kroku (podle pseudokódu) pro nějaké malé pole
2. Zamyslete se jak budou algoritmy pracovat pro pole, které je už setřízené.
3. Naprogramujte insertion sort a selection sort v pythonu.
4. Upravte selection sort tak, aby fungoval výběrem největšího prvku.
5. Upravte insertion sort tak, aby třídíl sestupně nebo vzestupně na základě volitelného parametru.

Kapitola 6

Asymptotický růst funkcí

V předchozích kapitolách jsme počítaly kolik kroků algoritmus vykoná v nejhorším případě (časová složistos). V této kapitole se budeme věnovat časové složitosti více do hloubky. Zavedeme O-notaci. Ukážeme jak analyzovat algoritmy efektivněji bez zbytečně velké přesnosti, která nám nedává informaci navíc. Dejme tomu, že algoritmus A a B řeší problém P. Spočítáním počtu instrukcí co provedou máme: $T_A(n) = 2n + 4$ a $T_B(n) = 6n - 10$.

Vidíme, že funkce $T_A(n)$ roste rychleji než $T_B(n)$ rozdíl je ale nezávislý na n. Podívejme se tedy na hrubší metodu analýzy, která tento nedostatek odstraní.

Definice

Asymptotická hodní mez $O(g)$ pro funkci $g(n)$ je

$$O(g(n)) = \{f(n) \mid \exists c > 0 \text{ a } n_0 \in \mathbb{N} \text{ tak, že } \forall n \geq n_0 \text{ je } 0 \leq f(n) \leq cg(n))\}$$

Všimněte, že definujeme množinu funkcí, které splňují podmínku "f funkce je zhora omezená funkcí g". Pak můžeme říct $n^3 + 2n + 4 \in O(n^3)$. Tedy že funkce$n^3 + 2n + 4$ roste nanejvýš tak rychle jako $c(n^3)$. Bude se nám hodit ještě pár definic. U časových složitostí se někdy zapisuje $f(n) = O(g(n))$ namísto $f(n) \in O(g(n))$. Význam je ale stejný.

Příklad

Dokažme $2n^2 - 4 = O(n^2)$. $f(n) = 2n^2 - 4$ a $g(n) = n^2$. Podle definice hledáme C a n_0 Zvolme tedy $c = 2$, $n_0 = 2$.
Pro každé $n \geq n_0$ platí $0 \leq f(n) \leq cg(n)$.
Tato podmínka totiž znamená, že pro každé $n \geq 2$ je

$$0 \leq 2n^2 - 4 \leq 2n^2,$$

Což zřejmě platí.
A tedy i $2n^2 - 4 = O(n^2)$ platí.

Příklad

Dokažme, že neplatí $0.5n^3 = O(20n^2)$.
Je tedy $f(n) = 0.5n^3$ a $g(n) = 20n^2$.
Zvolme libovolné $c > 0$.
Požadovaná nerovnost pak je $0.5n^3 \leq 20cn^2$, což je ekvivalentní

$$0.5n \leq 20c, \text{ tj. } n \leq 40c.$$

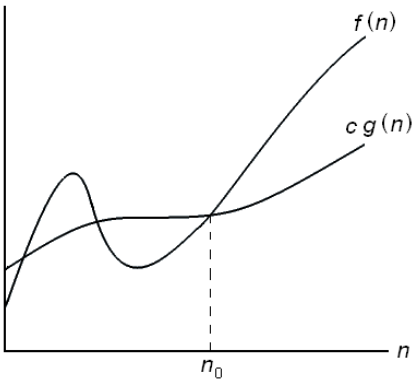
Nerovnost tedy neplatí pro žádné $n \geq 40c$. Neexistuje tedy n_0 tak, aby požadovaná nerovnost platila pro každé $n \geq n_0$.
Čísla c a n_0 požadovaná definicí tedy neexistují, proto

$$0.5n^3 = O(20n^2) \text{ neplatí.}$$

Definice

Asymptotická dolní mez $\Omega(g)$ Pro funkci $g(n)$ je

$$\Omega(g(n)) = \{f(n) \mid \exists c > 0,\ n_0 \in \mathbb{N} \forall n \geq n_0 \text{ je } 0 \leq cg(n) \leq f(n)\}.$$



Obrázek 6.1: Znázornění funkce $f(n)$ zdola omezené $cg(n)$.

Úkol

1) Dokažme $n^2 = \Omega(2n^2 - 4)$. Je tedy $f(n) = n^2$ a $g(n) = 2n^2 - 4$. Zvolme $c = 1/2$ a $n_0 = 1$. Pak pro $n \geq n_0$ platí $cg(n) = n^2 - 2 \leq n^2$. Důkaz je hotov.

2) Dokažme $2n^2 - 4 = \Omega(n^2)$. Je tedy $f(n) = 2n^2 - 4$ a $g(n) = n^2$. Zvolme $c = 1$ a $n_0 = 3$. Pak pro $n \geq n_0$ platí $cg(n) = n^2 \leq 2n^2 - 4$. Důkaz je hotov.

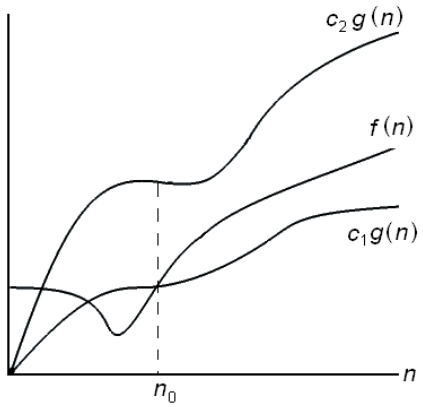
3) Dokažme, že pro každou funkci $f(n)$ a libovolnou $k > 0$ je $f(n) = \Omega(kf(n))$. Zvolme $c = 1/k$ a $n_0 = 1$. Podmínky definice pak zřejmě platí.

Definice

Asymptotická oboustranná (těsná) mez Pro funkci $g(n)$ je

$$\Theta(g(n)) = \{f(n) \mid \exists c_1, c_2 > 0,\ n_0 \in \mathbb{N} \forall n \geq n_0 \text{ je } 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)\}.$$

Alternativně platí, že $f(n) \in \Theta(g(n))$ právě když $f(n) \in O(g(n))$ a $f(n) \in \Omega(g(n))$



Obrázek 6.2: Znázornění funkce $f(n)$ mezi $c_1 g(n)$ a $c_2 g(n)$.

6.1 Vlastnosti

Tranzitivita odhadů

Pokud $f = O(g)$ a $g = O(h)$, pak $f = O(h)$.
Pokud $f = \Omega(g)$ a $g = \Omega(h)$, pak $f = \Omega(h)$.
Pokud $f = \Theta(g)$ a $g = \Theta(h)$, pak $f = \Theta(h)$.
Pokud $f = o(g)$ a $g = o(h)$, pak $f = o(h)$.
Pokud $f = \omega(g)$ a $g = \omega(h)$, pak $f = \omega(h)$.

Reflexitita a symetrie odhadů

Reflexivita:
$f = O(f)$.
$f = \Omega(f)$.
$f = \Theta(f)$.
Symetrie:
$f = \Theta(g)$ právě když $g = \Theta(f)$.
$f = O(g)$ právě když $g = \Omega(f)$.
$f = o(g)$ právě když $g = \omega(f)$.

6.2 Úkoly

1. Rozhodněte, zda platí následující:
   - $3n^2 + 5n + 1 = \Theta(n^2)$
   - $2n^3 - 10 = O(n^3)$
   - $12 \log n + 3n = \Theta(n \log n)$
   - $\frac{n^2+4n-5}{n-1} = O(n)$
2. Nechť $f(n)$ a $g(n)$ jsou nezáporné funkce. Pomocí definice Θ notace ukažte, že platí: $max(f(n), g(n)) = \Theta(f(n) + g(n))$
3. Ukažte, že pro každou konstantu a a b, kde $b > 0$ platí: $(n + a)^b = \Theta(n^b)$
4. Tvrzení "Časová složitost algoritmu A je nejméně $O(n^2)$". Vysvětlete proč takové tvrzení je nesmyslné.
5. Dokažte alternativní definici Θ-notace.

Kapitola 7

Třídění 2

7.1 Bubble Sort

Algorithm 10 Bubble Sort(A)

```
1: for j ← 1 to n − 1 do
2:     for i ← n − 1 downto j + 1 do
3:         if A[i] < A[i − 1] then
4:             swap(A[i], A[i − 1])
```

7.2 Quick Sort

Intuice: namísto řešení složitého problému najednou jej rozdělíme na menší (jednodušší) části, které vyřešíme samostatně a jejich výsledky spojíme. Této technice se říká rozděl a panuj. Přímo vybízí k rekurzívní implementaci.

1. **Rozděl**: vstupní pole a rozdělíme na dvě poloviny. Dělící index se nazývá pivot.

2. **Panuj**: rekurzivně volej quicksort na menší části pole.

Algorithm 11 Quick-Sort(A, p, r)

```
1:  procedure QUICK-SORT(A, p, r)
2:      if p < r then
3:          q ← PARTITION(A, p, r)
4:          QUICK-SORT(A, p, q - 1)
5:          QUICK-SORT(A, q + 1, r)
6:  procedure PARTITION(A, p, r)
7:      x ← A[r]                                   ▷ pivot
8:      i ← p − 1
9:      for j ← p to r − 1 do
10:         if A[j] ≤ x then
11:             i ← i + 1
12:             swap(A[i], A[j])
13:     swap(A[i + 1], A[r])
14:     return i + 1
```

7.3 Složitost quicksortu

Nejhorší případ:

- nastává tehdy, když volba pivotu rozdělí pole **nerovnoměrně**
- pivot je vždy **nejmenší nebo největší prvek** v dané části pole
- v takovém případě má jedno podpole délku $n - 1$ a druhé délku 0
- počet instrukcí roste kvadraticky s velikostí pole, což vede na časovou složitost $O(n^2)$

Počty porovnání v partition (nejčastěji vykonávaná instrukce):

1. volání: $n - 1$ porovnání (všechny prvky s pivotem)
2. volání: $n - 2$ porovnání
3. volání: $n - 3$ porovnání
4. ...
5. (poslední) 1 porovnání

Celkem:

$$1 + 2 + 3 + \cdots + (n - 1) = \frac{n(n-1)}{2} = \Theta(n^2)$$

Časová složitost v nejlepším a průměrném případě

Intuice: Quick Sort je velmi efektivní, pokud pivoty nejsou voleny extrémně nevhodně.

Nejlepší případ:

- nastává, když pivot rozdělí pole na dvě **přibližně stejně velké** části
- každé rekurzivní volání tedy pracuje s polovinou prvků
- funkce partition provede $O(n)$ operací pro každou úroveň rekurze
- hloubka rekurze je $\log_2 n$, proto dostáváme výslednou složitost $\Theta(n \log n)$

Průměrný případ:

- pivoty obvykle nedělí pole zcela rovnoměrně, ale přesto rozdělí pole „dostatečně dobře“
- strom rekurzivních volání má podobnou hloubku jako v nejlepším případě
- proto i složitost v průměrném je lineárně-logaritmická $\Theta(n \log n)$

7.4 Merge Sort

Problém: Máme dvě již setříděné posloupnosti čísel a chceme je sloučit do jedné. *Pozorování*: protože jsou obě části setříděné, víme, že nejmenší prvek celé dvojice se nachází vždy na začátku jedné z nich. Stačí tedy porovnávat pouze první prvky obou posloupností a menší z nich zapsat do výsledné.

Tento postup opakujeme, dokud nevyčerpáme jeden z obou vstupů, zbytek druhého vstupu je již setříděný, proto jej stačí přepsat do výsledku. *Důsledek*: celé sloučení lze provést v jediném průchodu oběma poli, tj. v čase úměrném délce většího z nich.

Algorithm 12 Merge-Sort

```
 1: procedure MERGE-SORT(A, p, r)
 2:     if p < r then
 3:         q ← ⌊(p + r)/2⌋
 4:         MERGE-SORT(A, p, q)
 5:         MERGE-SORT(A, q+1, r)
 6:         MERGE(A, p, q, r)
 7: procedure MERGE(A, p, q, r)
 8:     create arrays L and R from A[p ... q] and A[q + 1 ... r]
 9:     append sentinels to L and R
10:     i ← 1; j ← 1
11:     for k ← p to r do
12:         if L[i] ≤ R[j] then
13:             A[k] ← L[i]
14:             i ← i + 1
15:         else
16:             A[k] ← R[j]
17:             j ← j + 1
```

7.5 Úkoly

1. Implementujte Bubble Sort.
2. Upravte Bubble tak, aby používal tzv. Cocktail Shaker sort. (známí z přednášky) Tedy každý sudý průchod skenuje zleva doprava a každý lichý zprava doleva.
3. Upravte Bubble sort tak, aby měl na vstupu interval $[i, j]$, kde i a j jsou indexy vstupního pole. Algoritmus by pak měl třídit pouze tuto část pole.
4. Implementujte Quick Sort.
5. Upravte Quick Sort tak, aby používal Hybridní třízení, tedy aby počítal rekurzivní volání a pokud překročí hranici $MaxDepth$, přejde v dané části pole na nějaký jednodužší sort(select, bubble, insert).
6. Upravte Quick Sort tak, aby používal 3-cestný partition. Pole rozdělí na 3 části, prvky menší, než pivot, rovny pivotu a větší než pivot.
7. Implementujte Merge Sort.
8. Upravte Merge Sort na tzv. k-cestný merge sort. Algoritmus na vstupu dostane číslo k. Vstupní pole pak bude dělit na k částí. (Vyzkoušejte nejprve pro $k = 3$, $k = 4$, a poté pro obecné k.)

Kapitola 8

Třídění pomocí haldy

8.1 Halda

Halda je datová struktura. Konkrétně nás zajímá binární halda.

Definice

Halda je Binární strom splňující podmínku

- pro každý uzel x platí, že hodnota v rodiči x je menší než hodnota v x. pak mluvíme o Min-haldě.
- pro každý uzel x platí, že hodnota v rodiči x je větší, než hodnota v x. Tady jde o Max-haldu.

Haldu můžeme pohodlně reprezentovat pomocí pole a následujících operací. Z indexu i prvku lze snadno určit (spočítat) index Parent(i) jeho rodiče, index Left(i) jeho levého potomka i index Right(i) jeho pravého potomka:

Algorithm 13 Parent(i)

1: **return** $\left\lfloor \frac{i-1}{2} \right\rfloor$

Algorithm 14 Left(i)

1: **return** $2i + 1$

Algorithm 15 Right(i)

1: **return** $2i + 2$

Nyní definice haldy pomocí pole

Definice

Pole $A[0..n-1]$ se nazývá *max-halda*, pokud pro každý $i = 1, \ldots, n-1$ platí, že $A[i] \leq A[\text{Parent}(i)]$ (této nerovnosti říkáme vlastnost max-haldy). A duálně: $A[0..n-1]$ se nazývá *min-halda*, pokud pro každý $i = 1, \ldots, n-1$ platí, že

$A[i] \geq A[\text{Parent}(i)]$.

Pokud budeme dále používat Halda bude tím myšlena Max halda.

8.2 Konstrukce haldy

Předpokládá se, že části pole A, které odpovídají stromu s kořenem $A[\text{Left}(i)]$ (tj. stromu, jehož kořen je prvek s indexem $\text{Left}(i)$) i stromu s kořenem $A[\text{Right}(i)]$ tvoří haldy. Cílem je zařadit správně prvek $A[i]$ tak, aby část pole odpovídající stromu s kořenem $A[i]$ tvořila haldu.

Algorithm 16 Max-Heapify(A, i)

```
1: l ← Left(i)
2: r ← Right(i)
3: if l ≤ heap-size(A) and A[l] > A[i] then
4:     largest ← l
5: else
6:     largest ← i
7: if r ≤ heap-size(A) and A[r] > A[largest] then
8:     largest ← r
9: if largest ≠ i then
10:    swap(A[i], A[largest])
11:    Max-Heapify(A, largest)
```

Prvky vstupního pole, které tvoří v odpovídajícím stromu listy, tvoří jednoprvkové max-haldy (nemají totiž potomky). To jsou právě prvky $A[\lfloor n/2 \rfloor], \ldots, A[n-1]$ Build-Max-Heap prochází ostatní prvky od $A[\lfloor n/2 \rfloor - 1]$ až po $A[0]$ a každý zařadí funkcí Max-Heapify.

Algorithm 17 Build-Max-Heap($A[0..n-1]$)

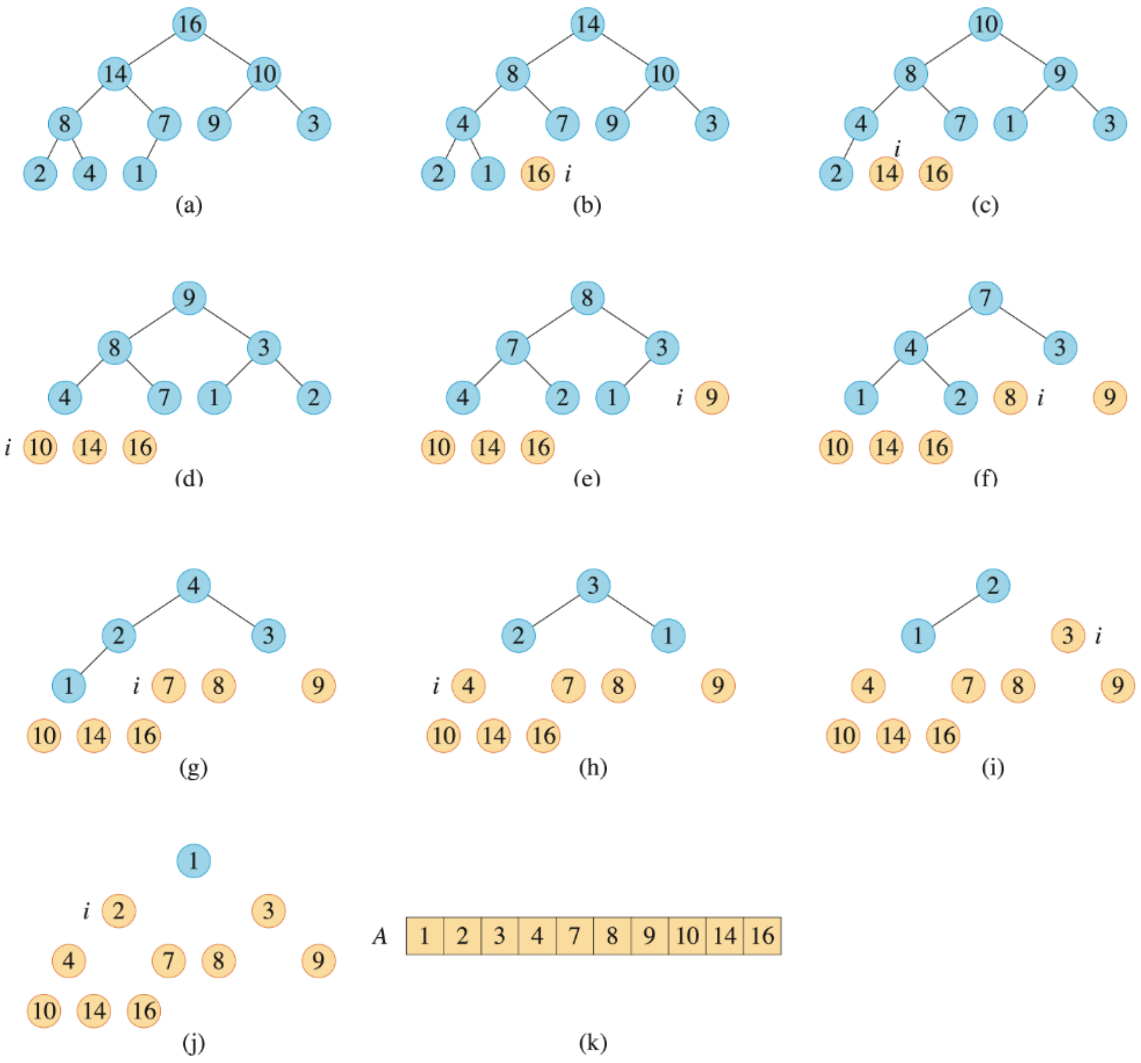
```
1: heap-size(A) ← n
2: for i ← ⌊n/2⌋ − 1 downto 0 do
3:     Max-Heapify(A, i)
```

8.3 Heap sort

Intuice: V každé iteraci vezmeme z haldy největší prvek a vyměníme jej s prvkem na konci pole. Tím porušíme podmínku max haldy, tu pak opravíme pomocí max-heapify.

Algorithm 18 Heap-sort($A[0..n-1]$)

```
1: Build-Max-Heap(A)
2: for i ← n − 1 downto 1 do
3:     swap(A[0], A[i])
4:     heapsize(A) ← heapsize(A) − 1
5:     Max-Heapify(A, 0)
```



8.4 Úkoly

1. Implementujte Heapsort
2. Upravte Heapsort aby na vstupu měl krom pole A i funkci funkce má na vstupu dvě hodnoty stejného typu a vrátí hodnotu typu bool. To bude funkce podle které heapsort třídí. Hlavička funkce by mohla vypadat takto: def sort(a: int, b: int) -> bool: