# Algoritmy I – prezentace k přednáškám

doc. Mgr. Jiří Dvorský, Ph.D.

Katedra informatiky
Fakulta elektrotechniky a informatiky
VŠB – TU Ostrava

Prezentace ke dni 12. února 2020





## Celková osnova přednášek

0. Úvodní přednáška
1. Pojem algoritmus
   - Algoritmický problém
   - Pojem algoritmu
   - Vlastnosti algoritmu
   - Prostředky pro zápis algoritmů
   - Algoritmus – formálnější přístup
   - Shrnutí
2. Základy C++
   - Počítač
     - Hardware – základní struktura počítače
     - Software
     - Osobní počítače
   - Programovací jazyk
   - Jazyk C++
     - Vznik spustitelného programu
   - Základní stavební bloky C++



## Celková osnova přednášek (pokrač.)

   - Komentáře v C++
   - Datové typy v C++
   - Proměné
3. Řídící struktury jazyka C++
4. Funkce v jazyce C++
5. Složitost algoritmů
   - Motivace
   - Analýza složitosti algoritmů
6. Pole v jazyce C++
   - Výpočty lineární algebry
     - Vektorové výpočty
     - Výpočty se čtvercovými maticemi
   - Game of Life
     - Popis hry
     - Zadání úlohy
     - Analýza řešení
     - Implementace



## Celková osnova přednášek (pokrač.)

   - Refaktorování
7. Rekurze
   - Co si představit pod pojmem rekurze?
   - Faktoriál
   - Fibonacciho posloupnost
8. Využití rekurze pro řešení problémů
   - Backtracking
   - Jezdcova procházka
9. Vyhledávání
   - Definice problému
   - Sekvenční vyhledávání
   - Vyhledávání půlením intervalu
10. Třídění
   - RadixSort
11. Základní třídící algoritmy

## Celková osnova přednášek (pokrač.)

- Asociativní třídicí algoritmy
- Třídění výběrem – SelectSort
- Třídění vkládáním – InsertSort
- Bublinové třídění – BubbleSort

12. Pokročilé třídící algoritmy
    - QuickSort – Třídění rozdělováním
    - Třídění haldou – HeapSort

13. Hledání nejkratší trasy aneb Sex, chlast a C++
    - Zadání úlohy – chlast
    - Metoda řešení – sex
    - Implementace – C++
    - Výsledky



# Úvodní přednáška

doc. Mgr. Jiří Dvorský, Ph.D.

Katedra informatiky
Fakulta elektrotechniky a informatiky
VŠB – TU Ostrava





## O předmětu Algoritmy I

**Upozornění**

Všechny aktuální informace k předmětu naleznete na

`http://www.cs.vsb.cz/dvorsky/`



# Děkuji za pozornost

# Pojem algoritmus

doc. Mgr. Jiří Dvorský, Ph.D.

Katedra informatiky
Fakulta elektrotechniky a informatiky
VŠB – TU Ostrava





## Algoritmický problém versus algoritmus

- vstupní data
- vstupní podmínka – co vše musí splňovat vstupní data
- výstupní data
- výstupní podmínka – co vše musí splňovat výstupní data

**Příklad**

Algoritmický problém – „Vyvrtání díry do betonového panelu“

- vstupní data – betonový panel
- vstupní podmínka – specifikace panelu
- výstupní data – díra v panelu
- výstupní podmínka – průměr a hloubka vyvrtané díry.



## Algoritmický problém versus algoritmus (pokrač.)

- Algoritmický problém reprezentuje zadání úlohy a algoritmus představuje způsob, jak tento problém řešit.
- Zřejmě může existovat více algoritmů řešících tentýž algoritmický problém.
- **Algoritmus** budeme chápat jako postup, jak vstupní data transformovat na výstupní.
- K našim účelům bude stačit tato intuitivní definice algoritmu

**Příklad**

Možné algoritmy řešící problém „Vyvrtání díry do betonového panelu“:

1. dloubání kávovou lžičkou,
2. ruční vrtačka
3. příklepová vrtačka
4. vrtací kladivo s SDS upínáním...



## Algoritmický problém versus algoritmus (pokrač.)

**Poznámky**

- Sestavit správný algoritmus řešení problému nemusí být vždy snadnou činností.
- Logická chyba v algoritmu může vést k nesprávným výsledkům.
- Pro řešení jedné a té samé úlohy může existovat více různých algoritmů, tyto algoritmy se můžou lišit v množství spotřebovaného času a paměťového prostoru.
- Algoritmus popisuje **konstruktivní** řešení problému – řešení problému je pomocí algoritmu **konstruováno** ze vstupních dat! Stejný postup používá například geometrie.

## Nekonstruktivní řešení problému aneb Na tohle není algoritmus!

**Příklad**

Naším úkolem je najít dvě iracionální čísla $x$ a $y$ taková aby platilo, že $x^y$ je číslo racionální.

**Řešení**

Zvolíme $x = \sqrt{2}$ a $y = \sqrt{2}$. Je-li $x^y$ číslo racionální jsme hotovi, není-li $x^y$ číslo racionální, zvolíme například $x = \sqrt{2}^{\sqrt{2}}$ a $y = \sqrt{2}$. Potom dostaneme

$$x^y = \left(\sqrt{2}\right)^{\sqrt{2}^{\sqrt{2}}} = \left(\sqrt{2}\right)^{\sqrt{2}\sqrt{2}} = \left(\sqrt{2}\right)^2 = 2$$

Je zřejmé, že řešením je buď dvojice čísel $x = \sqrt{2}$ a $y = \sqrt{2}$ nebo dvojice čísel $x = \sqrt{2}^{\sqrt{2}}$ a $y = \sqrt{2}$, přičemž nejsme z uvedeného řešení schopni říci která dvojice je vlastně řešením našeho problému.



## Nekonstruktivní řešení problému aneb Na tohle není algoritmus! (pokrač.)

**V čem spočívá problém?**

1. není jasné zda je $\sqrt{2}^{\sqrt{2}}$ číslo iracionální nebo ne,
2. není jasné jak volit kandidáty na řešení – iracionální čísla jsou nespočetná.

**Spočetnost množiny čísel**

- O množině řekneme, že je spočetná pokud ji lze zobrazit na množinu přirozených čísel.
- Přirozená čísla lze „probírat, vyjmenovat, jedno po druhém“.
- Stejně to lze provést s čísly celými a racionálními (zlomky).
- Nelze to provést s čísly iracionálními, reálnými a komplexními.
- Po 1 následuje nutně 2, po $2,5$ následuje co? $2,51$ nebo $2,501$ nebo $2,5001$? Jaké číslo následuje po $\sqrt{2}$?



## O původu slova algoritmus

- Slovo „algoritmus“ je odvozeno od perského matematika jménem *Abú Abd Alláh Muhammad Ibn Músá al-Chórezmí Abú Dža'far* česky
  *Otec Abdulláha, Mohamed, syn Mojžíšův, pocházející z města Chwarizm*.
- Město Chwarizm, někdy též Chórézm, najdete v dnešním Uzbekistánu pod jménem Chiva.
- Jeho mateřským jazykem byla perština, ale knihy psal arabsky.
- Jméno Al-Chórezmí bylo ve středověku latinizované na Al-Gorizmí, později na Algorismí.
- Zhruba v roce 825 napsal knihu Kitáb al-džám'a wa-l-tafríq bil-hisáb al-hindi, kterou v roce 1145 přeložil Robert z Chesteru do latiny jako Algorithmi de numero indorum.



## O původu slova algoritmus (pokrač.)

- V této knize popsal indický poziční číselný systém, výpočty s ním a číslo 0.
- Slovo algorismus, odvozeno od jeho jména, původně znamenalo „provádění aritmetiky pomocí arabských číslic“; abacisté počítali pomocí abaku, algoristé pomocí algorismů.
- Zavedl algebru jako samostatnou matematickou disciplínu, slovo algebra pochází z arabského al-džabr.
- Dále zavedl pojem neznámá a označovat ji tradičně písmenem $x$.

## Algoritmus

**Algoritmus** je předpis, který se skládá z kroků a který zabezpečí, že na základě vstupních dat jsou poskytnuta požadovaná data výstupní. Navíc každý algoritmus musí mít následující vlastnosti:

- konečnost,
- hromadnost,
- jednoznačnost,
- opakovatelnost,
- rezultativnost,
- elementárnost.



## Vlastnosti algoritmu

**Konečnost**

- Požadovaný výsledek musí být poskytnut v „rozumném“ čase.
- Pokud by výpočet trval na nejrychlejším počítači např. jeden milion let, těžko bychom mohli hovořit o algoritmu řešení, nemluvě o výpočtu, který by neskončil vůbec.
- Za rozumný lze považovat čas, kdy nám výsledek výpočtu k něčemu bude.

**Hromadnost**

- Vstupní data nejsou v popisu algoritmu reprezentována konkrétními hodnotami, ale spíše množinami, ze kterých lze data vybrat (např. při třídění přirozených čísel bude vstup konečnou podmnožinou množiny všech přirozených čísel).
- Při popisu algoritmu v programovacím jazyce se to projeví tím, že vstupy do algoritmu jsou označeny symbolickými jmény.



## Vlastnosti algoritmu (pokrač.)

**Jednoznačnost**

- Každý předpis je složen z kroků, které na sebe navazují.
- Každý krok můžeme charakterizovat jako přechod z jednoho stavu algoritmu do jiného, přičemž každý stav je určen zpracovávanými daty.
- Tím, jak data v jednotlivých stavech algoritmu vypadají, musí být jednoznačně určeno, který krok následuje.
- Například v řešení trojúhelníka může nastat situace, kdy vychází na základě vstupních dat jedno nebo dvě řešení. Situace je tedy nejednoznačná, řešení musí být jednoznačné, tzn. v předpisu se s touto možností musí počítat a musí v něm být návod, jak ji řešit.



## Vlastnosti algoritmu (pokrač.)

**Opakovatelnost**

- Při použití stejných vstupních údajů musí algoritmus dospět vždy k témuž výsledku.
- Algoritmus si tedy můžeme představit jako mlýnek na maso – pokud do něj vložíme stejné suroviny a zatočíme klikou, dostaneme vždy stejný výsledek jako při předchozím mletí.

**Rezultativnost**

- Algoritmus musí vydat aspoň jeden výstup, který je správný tj. vyhovuje výstupní podmínce.

**Elementárnost**

- Algoritmus se skládá z konečného počtu jednoduchých (elementárních) kroků.

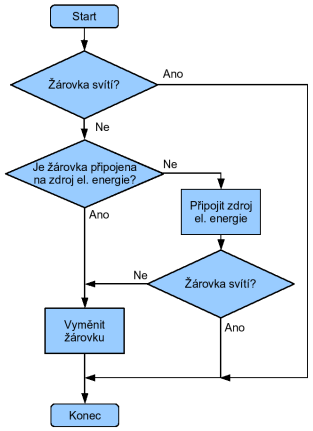
## Prostředky pro zápis algoritmů

Pro zápis algoritmů můžeme využít:

- přirozený jazyk – slovní popis například receptu, „Vraž do toho kopu vajec a dvě libry másla. . . “
- grafický zápis – vývojový diagram, strukturogram,
- pseudokód nebo
- programovací jazyk.



## Vývojový diagram




Vývojový diagram [2]

- grafické znázornění jednotlivých kroků algoritmu,
- obrazce různého tvaru, navzájem propojené pomocí šipek,
- obrazce reprezentují jednotlivé kroky:
  - obdélník s popisem – dílčí krok zpracování,
  - kosočtverec – větvení postupu v závislosti na splnění podmínky,
  - obdélník se zaoblenými rohy – počátek nebo ukončení zpracování,
- šipky určují směr zpracování
- Frank Gilbreth, 1921, dokumentace procesů v průmyslu.



## Pseudokód

```
1 while i < n do
2 |   a while loop including a repeat–until loop;
3 |   repeat
4 |   |   if x ∈ A then
5 |   |   |   we do that;
6 |   |   end
7 |   until this end condition;
8 end
```

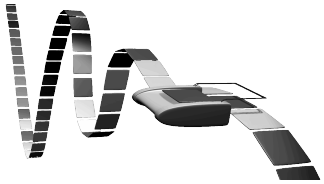
- kompaktní a neformální způsob zápisu počítačového algoritmu,
- používá strukturní konvence programovacích jazyků,
- nezahrnuje detailní syntaxi specifické pro konkrétní programovací jazyk,
- pro srozumitelnost je částečně doplněn popisy podrobností v přirozeném jazyce nebo kompaktně vyjádřeným matematickým zápisem.



## Turingův stroj

**Turingův stroj**

- teoretický model počítače popsaný matematikem Alanem Turingem,
- procesorová jednotka, tvořená konečným automatem,
- program ve tvaru pravidel přechodové funkce,
- a pravostranně nekonečné pásky pro zápis mezivýsledků.



Turingův stroj [1]

Neformálně řečeno je Turingův stroj primitivní počítač s co nejjednoduššími instrukcemi a jednou pamětí, páskou.

## Turingův stroj (pokrač.)

Výpočet na Turignově stroji:

1. na pásku zapíšeme vstupní posloupnost symbolů $w$
2. Turingův stroj začne zpracovávat vstup a na pásce budeme očekávat výstupní symboly.
3. Turingův stroj ukončí výpočet ve třech možných stavech:
   - 3.1 vstup $w$ odpovídá vstupním podmínkám, vstup byl přijat a na pásce je výsledek, **přijato**,
   - 3.2 vstup $w$ neodpovídá vstupním podmínkám, vstup nebyl přijat, **nepřijato**
   - 3.3 Turingův stroj se **zacyklí** a nikdy se nezastaví, **nelze rozhodnout**

Turingův stroj tedy **neumí řešit všechny problémy**.



## Church-Turingova teze

**Church-Turingova teze**

Ke každému algoritmu existuje ekvivalentní Turingův stroj.

- Jinak řečeno, vše co lze vypočítat Turingovým strojem (počítačem), lze vypočítat i algoritmem a naopak.
- Naše počítače jsou tedy stejně „schopné“ jako kterékoliv jiné počítače, které by bylo možné sestrojit „na druhém konci vesmíru“.
- Nebereme ale v úvahu rychlost počítače a velikost paměti, což v praxi hraje významnou roli.
- Kvůli neformální definici pojmu algoritmus nemůže být tato teze nikdy dokázána.
- Lze ji ale vyvrátit, podaří-li se sestrojit algoritmus, který bude umět řešit problémy, které Turingův stroj řešit neumí.



## Church-Turingova teze (pokrač.)

- Jelikož každý počítačový program lze přeložit do jazyka Turingova stroje a obvykle i naopak, lze tezi ekvivalentně formulovat pro kterýkoli běžně používaný programovací jazyk, který je turingovsky úplný tj. ekvivalentní Turingovu stroji.
- Z toho plyne, že všechny programovací jazyky mají stejnou vyjadřovací sílu, lze v nich naprogramovat stejné algoritmy.
- Otázkou zůstává, proč existuje takové množství programovacích jazyků? Důvod: důvod je praktický, programovací jazyky jsou uzpůsobeny pro jistý okruh problémů. V jednom jazyce se lépe programuje tato úloha, v jiném jazyce jiná.



## Shrnutí

- Algoritmický problém
- Algoritmus
  - konečnost,
  - hromadnost,
  - jednoznačnost,
  - opakovatelnost,
  - rezultativnost,
  - elementárnost.
- Turingův stroj
- Church-Turingova teze

## Literatura


[1] Turingův stroj. In: *Wikipedia: the free encyclopedia* [online]. San Francisco (CA): Wikimedia Foundation, 2001- [cit. 2017-09-26]. Dostupné z: `https://cs.wikipedia.org/wiki/Turing%C5%AFv_stroj#/media/File:Maquina.png`

[2] Vývojový diagram. In: *Wikipedia: the free encyclopedia* [online]. San Francisco (CA): Wikimedia Foundation, 2001- [cit. 2017-09-25]. Dostupné z: `https://commons.wikimedia.org/w/index.php?curid=9024401`




## Děkuji za pozornost



# Základy C++


doc. Mgr. Jiří Dvorský, Ph.D.

Katedra informatiky
Fakulta elektrotechniky a informatiky
VŠB – TU Ostrava






## Počítač

**Počítač**

- zařízení schopné provádět výpočty a rozhodnutí založená na logice.

**Hardware (HW)**

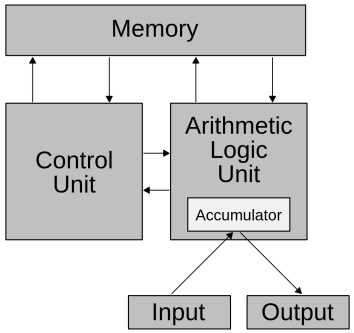
- technická část počítače, zařízení z nichž je počítač sestaven,
- např. klávesnice, monitor, pevný disk, paměť, procesor, sběrnice ...

**Software (SW)**

- programové vybavení počítače, „oživuje“ hardware,
- mnoho úrovní software:
  - ovladače zařízení,
  - jádro operačního systému,
  - moduly operačního systému (správa uživatelů, správa disků atd.),
  - grafické uživatelské rozhraní,
  - uživatelské aplikace.

## Základní struktura počítače

- vstup,
- výstup,
- paměť,
- řadič a
- aritmeticko logická jednotka (ALU).




**Poznámky**

- O takto koncipovaném počítačí říkáme, že má **von Neumannovskou architekturu**.
- Řadič a ALU tvoří dohromady to, čemu se běžně říká procesor.



## Základní struktura počítače

**Vstup**

- zařízení pomocí kterého se dostávají informace do počítače,
- klávesnice, myš, scanner, počítačová síť ...

**Výstup**

- předává informace z počítače jeho periferiím – výsledky výpočtů a řídící informace.

**Aritmeticko logická jednotka (ALU)**

- centrální výkonná část počítače,
- provádí veškeré výpočty, logické operace atd.
- Charles Babbage nazýval ALU slovem „mlýnice“

**Řadič**

- řídící část počítače,
- řídí a koordinuje ostatní části počítače.



## Základní struktura počítače

**Primární paměť**

- vnitřní paměť počítače,
- Random Access Memory (RAM),
- velice rychlá, relativně malá kapacita,
- obsahuje zpracovávané programy, jejich data,
- data ze vstupu,
- data čekající na výstup.

**Sekundární paměť**

- vnější paměti počítače,
- relativně pomalé, velká kapacita,
- obsahuje všechny programy počítače,
- data pro tyto programy.



## Program versus proces

**Program**

- posloupnost instrukcí které řídí počítač při zpracování dat,
- **statický**.

**Proces**

- program, který je v daném okamžiku již vykonáván počítačem,
- **dynamický**

## Operační systém (OS)

- první počítače OS postrádaly, procesy si zajišťovaly vše samy,
- snaha usnadnit a sjednotit práci s počítačem, periferiemi,
- poskytnout podporu pro běžně prováděné činnosti – čtení dat ze vstupu, tisk na tiskárně atd.,
- zvýšit výkon počítače,
- jednoúlohový vs. víceúlohový OS,
- jednouživatelský vs. víceuživatelský OS,
- dnešní OS: UNIX/Linux, Windows, Mac OS X Mountain Lion atd.,
- Pozor! Grafické uživatelské rozhraní (GUI), které vidíte na obrazovce je jen „špičkou“ ledovce. OS je rozsáhlá infrasturktura skrytá za tímto rozhranním.



## Osobní počítače

- v průběhu 70. let 20. století používá Apple pro své počítače označení osobní,
- v roce 1981 uvedlo IBM osobní počítač, PC s procesorem Intel 8088,
- IBM zvolilo cestu otevřeného standardu, podíleli se různí výrobci HW a SW.
- typicky procesory Intel či AMD, operační systém zprvu DOS, následně Windows a Linux.



## Programovací jazyk

- prostředek pro zápis algoritmů ve formě použitelné na počítači,
- jde o formální jazyk – formálně tj. matematicky popsaná syntaxe a sémantika jazyka,
- jinak řečeno jde o jazyk, který má svou gramatiku, slova, věty, věty mají svůj význam,
- různé programovací jazyky jsou určeny pro různé účely.



## Programovací jazyky

**Nižší programovací jazyky**

- jazyk plně přizpůsobený počítači,
- jazyk symbolických adres, strojový kód, částečně VHDL.

**Vyšší programovací jazyky**

- jazyk lépe pochopitelnější pro člověka, vyšší míra abstrakce,
- syntaxe podobná primitivní angličtině, užití např. běžných značek pro matematické operace,
- jednomu příkazu ve vyšším jazyce odpovídá obecně mnoho instrukcí ve strojovém jazyce,
- většina současně používaných jazyků.

## Nižší programovací jazyky

**Strojový kód**

- jediný jazyk, kterému počítač (procesor) rozumí,
- sekvence 0 a 1,
- strojově závislý,
- pro člověka „nestravitelný“.

**Příklad**

```
0010 1101
1101 1000
1001 1000
0001 0010
0010 0000
```



## Nižší programovací jazyky (pokrač.)

**Jazyk symbolických adres**

- nesprávně česky nazýván „assembler“, assembler – překladač do strojového kódu,
- assembly language,
- anglické zkratky názvů instrukcí,
- pro člověka „stravitelnější“,

**Příklad**

```
MOV reg0, Vyska
MOV reg1, Sirka
MUL reg0, reg1
MOV reg0, Plocha
```



## Kompilované programovací jazyky

- program je nejprve kompletně přeložen překladačem do strojového kódu počítače,
- teprve po bezchybném překladu je možné přeložený tvar programu spustit,
- výhody:
  - výsledný spustitelný tvar programu může být přesně optimalizován na konkrétní HW a OS,
  - větší rychlost běhu takového programu, lepší využití prostředků HW.
- nevýhody:
  - závislost na konkrétním HW a OS,
  - pro každý HW a OS nutno zvlášť přeložit,
  - menší pružnost při odstraňování chyb v programech, nelze snadno měnit kód programu za jeho běhu.
- typické překládané jazyky – C, C++, Pascal (Delphi), Fortran.

**Poznámky**



## Kompilované programovací jazyky (pokrač.)

- překladač se někdy označuje jako kompilátor (angl. compiler) a překlad jako kompilování (angl. to compile),
- překladač je program, který překládá program z jednoho jazyka do jiného jazyka, typicky například z C++ do strojového kódu procesoru,
- překladač se při překladuz řídí gramatikou zdrojového jazyka, de-facto provádí větný rozbor, tak jak jej znáte z výuky češtiny,
- překladače nemusí překládat jen „programy“, například tato prezentace je napsána v jazyku LaTeX a překladem textu v tomto počítačovém jazyce získáme PDF dokument, který právě čtete.

## Interpretované programovací jazyky – klasický přístup

- speciální prostředí pro běh programu,
- toto prostředí čte zdrojový kód programu a jakmile načte celý příkaz, okamžitě jej vykoná a pokračuje na další příkaz,
- výhody:
  - nezávislost programů na HW a OS, závislý je pouze interpret,
  - dynamičtější manipulace s programem, např. za běhu programu lze opravovat zdrojový kód, protože tento kód je interpretem opakovaně načítán.
- nevýhody:
  - obtížná optimalizace pro daný HW a OS,
  - horší využití prostředků HW a OS (na počítači musí běžet i interpret).
- typický interpretovaný jazyk je BASIC ve své klasické podobě, například na starých 8-bitových počítačích
- pozor – jazyk Visual Basic .NET nebo Java jsou uspořádány jinak!



## Interpretované programovací jazyky – virtuální stroj

- členění na kompilované a interpretované není absolutní, tyto přístupy lze kombinovat,
- jazyk Java a jeho Java Virtual Machine (JVM):
  - JVM představuje virtuální procesor se svým specifickým strojovým jazykem,
  - zdrojový kód v jazyku Java je kompilován do strojového kódu JVM (výsledkem jsou class a jar soubory),
  - při spuštění JVM začne jednotlivé instrukce uložené např. v jar souboru interpretovat.
- obdobně se chová i Visual Basic .NET či Python,
- Just in Time (JIT) kompilace:
  - zdrojový kód programu je teprve při spuštění kompilován do strojového kódu počítače a okamžitě spuštěn,
  - přeložený tvar programu se nikde neukládá, program je vždy kompilován znovu.



## Vyšší programovací jazyky

Lze dělit podle **programovacího paradigmatu**, čili stylu programování, na:

**Procedurální jazyky tj. imperativní jazyky**

- procedurální paradigma – program popisuje krok za krokem co se má provést,
- strukturované – C, BASIC,
- objektově orientované – Smalltalk, Java.

**Neprocedurální jazyky tj. deklarativní jazyky**

- neprocedurální paradigmata – popisujeme spíše výsledek, ne jak ho dostáhnout,
- funkcionální – Lisp, Haskell, Scheme,
- logické – Prolog.



## Příklady vyšších programovacích jazyků

- FORTRAN (FORmula TRANslation) – první vyšší jazyk z roku 1957, vývoj trval tři roky, dodnes používán pro numerické výpočty,
- COBOL (COmmon Business-Oriented Language) – 1959, jazyk pro „hromadné zpracování dat“,
- Pascal – 1970, strukturované programování, výuka programování.
- Basic (Beginner's All-purpose Symbolic Instruction Code) – 1963, nyní převážně MS Visual Basic

**Příklad**

Plocha = Vyska * Sirka

## Co je C++?

**C++ je:**

- vyšší programovací jazyk,
- určen pro všeobecné použití,
- určen pro profesionální programátory,
- zachovává zpětnou kompatibilitu s jazykem C.
- C++ je jazyk typicky překládaný do strojového kódu.

**C++ dnes**

V dnešní době plní jazyk C++ úlohu moderního assembleru.



## Historie jazyka C

- autor – Dennis Ritchie, Bell Laboratories, v letech 1968 až 1973,
- vychází z jazyků BCPL a B,
- silná vazba na OS UNIX, jádro OS UNIX práno v C,
- Brian Kernighan a Dennis Ritchie vydávají v roce 1978 knihu *The C Programming Language*, jazyk označováno jako „K & R C“
- 1989 – ANSI standard
- 1990 – ANSI a ISO norma ANSI/ISO 9899: 1990
- 2000 – C99, ISO 9899:1999



## Jazyk C++

- 1979 – Bjarne Stroustrup (Bell Laboratories) vyvíjí jazyk „C with classes“
- 1983 – název změněn na C++
- 1985 – vychází kniha *The C++ Programming Language*, zatím se nejedná o standard,
- 1998 – norma ISO/IEC 14882:1998
- 2011 – norma ISO/IEC 14882:2011, 12. srpna 2011,
- 2014 – současný standard označovaný C++14, podporovaný gcc, VS2017 atd.
- 2017 – připravovaný standard, aktuálně ve formě draftu.
- rozšíření jazyka C o OOP, plus další úpravy,
- jazyk není čistě objektový.



## Vznik spustitelného programu – základní pojmy

- **zdrojový kód** – napsán programátorem, textový soubor obsahující konstrukce C++ o kterých programátor doufá, že budou vykonávat to co požaduje.
- **hlavičkový soubor** – deklarace funkcí, tříd, konstant atd. Buď napsán programátorem nebo dodán s překladačem.
- **projekt** – soubor popisující, ze kterých zdrojových kódů a knihoven vznikne výsledný program.
- **kód s relativními adresami** – anglicky object file, zdrojové kódy jsou přeloženy do instrukcí procesoru, ale nejsou dořešeny všechny vazby.
- **spustitelný kód** – soubor s binárním zápisem instrukcí, přímo vykonavatelný procesorem.

## První program

### Hello World

```
1 #include <iostream>
2
3 using namespace std;
4
5 // This is single line comment.
6 void main()
7 {
8    cout << "Hello World" << endl;
9 }
```

### Čeština ve zdrojovém kódu

Nedoporučuji psát do C++ cokoliv česky s diakritikou, ani komentáře.



## Vznik spustitelného programu

Vznik spustitelného programu ze zdrojového kódu prochází těmito fázemi:

1. **Edit** – editace zdrojového kódu,
2. **Preprocess** – rozvinutí maker, vložení hlavičkových souborů,
3. **Compile** – vlastní překlad, výsledkem je modul s relativními adresami nebo alternativně program v assembleru,
4. **Link** – sestavení výsledného programu, vstupem jsou moduly s relativními adresami, plus knihovny; výsledkem spustitelný program,
5. **Load** – OS zavede spustitelný kód do primární paměti,
6. **Execute** – vlastní vykonávání programu.



## Vznik spustitelného programu

Ke vzniku spustitelného programu se využívají následující programy:

1. **Edit** – textový editor,
2. **Preprocess** – preprocesor,
3. **Compile** – překladač, kompilátor,
4. **Link** – linker (sestavovač, spojovač?),
5. **Load** – operační systém,
6. **Execute** – operační systém.



## Ladění

- Proces odstraňování chyb v programu se nazývá **ladění**.
- K ladění slouží program zvaný **debugger**, který
  - umožňuje sledovat vykonávání programu,
  - vypisovat hodnoty proměnných,
  - zobrazuje zásobník volání funkcí,
  - umožňuje nastavovat tzv. breakpointy, tj. místa, kde se má přerušit vykonávání programu
  - krokování programu po jednom příkazu.

## Základní stavební bloky C++

Data – údaje, které bude program zpracovávat,

Kód – instrukce programu prostřednictvím jsou zapisovány algoritmy,

Komentáře – pomocné informace pro programátora umístěné ve zdrojovém kódu.



## Komentáře v C++

Existují dva druhy komentářů:

- jednořádkový – začíná sekvencí `//` a končí s koncem řádku
- víceřádkový – omezen sekvencemi `/*` a `*/`

```
1 // This is single line comment.
2 void main()
3 {
4   cout << "Hello World" << endl; // Single line comment.
5   /*
6   Multi line comment
7   suitable for long texts.
8   */
9 }
```



## Komentáře v C++ (pokrač.)

**Komentáře**

- při překladu jsou kompilátorem ignorovány,
- určeny pouze pro programátora,
- zvyšují čitelnost kódu,
- slouží k zaznamenání informací, které nejsou přímo patrné ze zdrojového kódu – autorství, verze, datum revize atd.
- tzv. dokumentační komentáře lze využít pro zápis programátorské dokumentace přímo k příslušným částem kódu, generování dokumentace externím SW (např. Doxygen),
- platí zásada, že **všeho moc škodí**,
- jakmile je nutno v komentáři „dovysvětlit“ nesrozumitelný úsek kódu, je tento kód špatný!
- stejně tak je zcela nevhodné komentovat každý řádek kódu!



## Datové typy v C++

Datový typ definuje dvě vlastnosti:

- rozsah povolených dat, čili jaké hodnoty jsou povolené a jaké ne,
- povolené operace, čili co s těmi hodnotami lze provádět.

**Berte je vážně!**

Datové typy jsou jedním z prostředků pro psaní kvalitních programů.

## Datové typy v C++

Datový typy v C++ dělíme na:

- základní
  - celočíselné
  - s plovoucí řádovou čárkou
  - logické
- adresní
  - ukazatelé
  - reference
- strukturované
  - pole
  - struktura
  - třída



## Celočíselné datové typy v C++

- `char` – jeden byte
- `short` – větší nebo roven `char`
- `int` – větší nebo roven `short`
- `long` – větší nebo roven `int`
- `long long` – větší nebo roven `long`

Všechny typy jsou automaticky se znaménkem, neznaménková verze se uvádí klíčovým slovem `unsigned`, např. `unsigned int`.

**Obvyklé rozsahy celočíselných datových typů**

| Typ | Počet bitů | Rozsah hodnot |
|---|---|---|
| `char` | 8 | $2^7 \dots 2^7 - 1$ |
| `short` | 16 | $\check{}2^{15} \dots 2^{15} - 1$ |
| `int` | 32 | $\check{}2^{31} \dots 2^{31} - 1$ |
| `long` | 32 | $\check{}2^{31} \dots 2^{31} - 1$ |
| `long long` | 64 | $\check{}2^{63} \dots 2^{63} - 1$ |



## Celočíselné datové typy v C++ (pokrač.)

**Číselné soustavy používané v C++**

- desítková – základ soustavy je číslo 10, číslice 0 až 9, např. 158
- osmičková – základ soustavy je číslo 8, číslice 0 až 7, např. 0236
- šesnáctková – základ soustavy je číslo 16, číslice 0 až 9, písmena A až F, kde A je deset, B jedenáct atd., např. 0x9E



## Celočíselné datové typy v C++ (pokrač.)

**Poznámky**

- Nejběžnější se pochopitelně desítková soustava.
- Binární soustava se nepoužívá ve zdrojovém kódu přímo nepoužívá (dlouhý zápis čísel).
- Osmičková soustava je spíše historický přežitek.
- Šestnáctková soustava se používá pro manipulaci s bity – vždy 4 bity na jednu šestnáctkovou číslici.
- Šestnáctková soustava se také často nazývá jako **hexadecimální** soustava.

## Číselné soustavy – příklad

Číslo 158 můžeme vyjádřit v jednotlivých číselných soustavách takto:

- binárně

$$10011110_2 = 1\times2^7+0\times2^6+0\times2^5+1\times2^4+1\times2^3+1\times2^2+1\times2^1+0\times2^0$$

- osmičkově

$$236_8 = 2 \times 8^2 + 3 \times 8^1 + 6 \times 8^0$$

- desítkově

$$158_{10} = 1 \times 10^2 + 5 \times 10^1 + 8 \times 10^0$$

- hexadecimálně

$$9E_{16} = 9 \times 16^1 + 14 \times 16^0$$



## Datové typy s plovoucí řádovou čárkou v C++

| Datový typ | Počet bitů | Rozsah hodnot | Platné číslice |
|---|---|---|---|
| `float` | 32 | $\pm3,4 \times 10^{\pm38}$ | 7 |
| `double` | 64 | $\pm1,7 \times 10^{\pm308}$ | 15 |
| `long double` | ve Visual Studio 2013 shodné s `double` | | |

**Reálná čísla vs. čísla s plovoucí řádovou čárkou**

Čísla s plovoucí řádovou čárkou nejsou totožná s reálnými čísly z matematiky – problém s periodickými čísly, iracionálními čísly!



## Logický datový typ v C++

- jazyk C++ zavádí pro logické hodnoty datový typ `bool`,
- konstanty `true` pravda a `false` nepravda,
- platí že `true` $\neq$ `false`,
- neexistuje žádná souvislost mezi `bool` a `int`!

**Logické hodnoty v ANSI C**

Jazyk ANSI C používá pro uložení logických hodnot typ `int` včetně svérázné definice hodnot true a false. V jazyce C++ důrazně doporučuji nepoužívat tento zastaralý přístup a doporučuji používat typ `bool`.



## Logický datový typ v C++

**Platné vztahy**

`true` $\neq$ `false`

$\neg$`true` = `false`

$\neg$`false` = `true`

## Proměné

Data v programu jsou reprezentována pomocí proměnných.

### Proměnná

- identifikátor,
- datový typ,
- hodnota.



## Identifikátor

- obecný název pro pojmenování čehokoliv v programu co lze pojmenovat,
- konečná neprázdná posloupnost písmen, číslic, znaků podtržítko, která nesmí začínat číslicí,
- rozlišují se malá a velká písmena.
- délka identifikátoru není teoreticky omezena

### Příklad

```
a A a1 ab _a _a1 a_b VeryLongIdentifier
```

### Jak správně volit identifikátory?

- Buď česky nebo anglicky, ne mixovat! `NextMove` nebo `DalsiTah` ne `NextTah`.
- Raději anglicky.
- Identifikátor musí popisovat co obsahuje! `NumberOfStudents` nebo `PocetStudentu` ne `A28`.



## Děkuji za pozornost



# Řídící struktury jazyka C++

doc. Mgr. Jiří Dvorský, Ph.D.

Katedra informatiky
Fakulta elektrotechniky a informatiky
VŠB – TU Ostrava



## Děkuji za pozornost



## Funkce v jazyce C++

doc. Mgr. Jiří Dvorský, Ph.D.

Katedra informatiky
Fakulta elektrotechniky a informatiky
VŠB – TU Ostrava





## Děkuji za pozornost



## Složitost algoritmů

doc. Mgr. Jiří Dvorský, Ph.D.

Katedra informatiky
Fakulta elektrotechniky a informatiky
VŠB – TU Ostrava



## Motivace – kvalita řešení problémů

1. Máme navrhnout algoritmus potažmo software pro řešení problému.
2. Navrhneme algoritmus a implementujeme jej.
3. Máme zájem řešit problémy efektivně.
4. Naskýtá se otázka, jak kvalitní řešení problému jsme vlastně navrhli. Nejlepší? Nejhorší? Průměrné? Jak tedy měřit kvalitu řešení?
5. Kvalitu řešení budeme měřit pomocí tzv. **složitosti**. Složitost budeme chápat jako míru náročnosti našeho řešení na zdroje počítače.
6. Složitost problému budeme definovat jako složitost nejlepšího algoritmu řešícího daný problém.

**Příklad**

Problém – Vyvrtání díry do betonového panelu

Řešení – kávová lžička, ruční vrtačka, příklepová vrtačka, **vrtací kladivo s SDS upínáním vrtáku**



## Motivace

1. **Algoritmus** je dosud chápán jako „mlýnek na maso“ – vložíme vstupní data, zatočíme klikou a dostaneme požadovaný výstup.
2. Je jasné, že různé mlýnky na maso se můžou kvalitativně lišit.
3. Máme zájem o vývoj kvalitních algoritmů a datových struktur.
4. Musíme tedy najít nástroj, který umožní měřit kvalitu algoritmů.

Nástroj pro měření kvality algoritmů se nazývá **složitost algoritmů**.

Pro měření složitosti algoritmů se okamžitě nabízí dvě měřítka:

- **prostor** – paměť potřebná pro výpočet a
- **čas** – doba běhu programu.



## Motivace

1. V programech budeme používat i různé **datové struktury** – zásobníky, fronty, seznamy, množiny, tabulky atd.
2. Někdy si prostě nevystačíme s „proměnnou typu `int`“.
3. Datové struktury můžeme zjednodušeně chápat jako „skladiště“ informací.
4. Kvalitu datových struktur budeme měřit kvalitou operací pracujících s danou datovou strukturou, např. složitost nalezení prvku v tabulce, vložení prvku do tabulky, smazání prvku v tabulce atd.
5. Tímto získáme jednotný nástroj na měření kvality algoritmů a datových struktur.



## Prostorová složitost algoritmu

- rozsah použité pracovní paměti (RAM či diskový prostor),
- lze ji jednoduše měřit v bajtech a násobcích,
- měření je poměrně přímočaré – lze provést jakousi inventuru programu,
- využívá stejné matematické postupy jako složitost časová.

## Časová složitost algoritmu

- Doba běhu programu závisí typicky na mnoha faktorech. Jak tedy provést měření?
- Máme-li program řešící danou úlohu, můžeme na něm testovat jak dlouho program poběží pro různé velikosti vstupu a různé vstupy stejné velikosti.
- Můžeme sestrojit graf, jedna osa velikost vstupu, druhá osa doba běhu programu.
- Pozor nemusí to být funkce, pro jednu hodnotu na ose $x$ může existovat více hodnot na ose $y$!
- Pro tutéž velikost vstupu může existovat více výsledků – nelze brát do úvahy jen prostou velikost vstupu.
- Například třídění posloupnosti čísel. Někdy stačí jen zkontrolovat, že posloupnost je již setříděná, jindy je potřeba ji celou „přeskládat“.
- Aby měl takový test smysl, musí být počet testů statisticky významný. Nelze dělat závěry z jednoho testu.



## Časová složitost algoritmu

**Obecně se dá říci:**

- že doba běhu se bude zvyšovat s rostoucím velikostí vstupu a
- doba běhu bude záviset na HW a SW platformě, na které byly testy prováděny.

**Úskalí experimentální analýzy časové složitosti programu**

- Experimenty lze většinou provádět na omezeném vzorku dat (třídění 100 prvků, počet možných vstupů je 100!).
- Vybrat statisticky významný vzorek tak, aby poskytoval hodnověrné výsledky může být velice obtížné.
- Je velice problematické srovnávat měření na počítači A s operačním systémem B s počítačem C, který pracuje s operačním systémem D.
- A je také nutné algoritmus reálně implementovat a experimenty provést. Kdo to zaplatí?!



## Časová složitost algoritmu

Metoda měření časové složitosti algoritmů:

- Musí brát do úvahy všechny možné vstupy.
- Lze srovnávat složitost dvou algoritmů nezávisle na HW a SW platformě
- analýzu algoritmu lze provést studiem formálního popisu algoritmu bez nutnosti provést detailní implementaci a provádět experimentální výpočty pomocí takto implementovaného programu



## Časová složitost algoritmu

Po formální stránce směřujeme k tomu, že každému algoritmu přiřadíme funkci $f(n)$, která udává dobu běhu daného algoritmu vzhledem k velikosti vstupních dat $n$.
Doba běhu algoritmu – předpokládáme, že máme k dispozici hypotetický počítač, který přímo vykonává formálně popsané algoritmy.
Výsledkem našeho zkoumání je tvrzení typu
Algoritmus A pracuje s časovou složitostí úměrnou $n^2$.
Toto tvrzení znamená, že pokud bychom algoritmus implementovali a provedli měření doby běhu, tak pro žádný vstup nepřekročí doba běhu tohoto programu čas $cn^2$, kde $c$ je konstanta závisející na HW a SW platformě.

## Analýza složitosti algoritmů

Potřebné nástroje:

- měřítko (metrika) pro měření doby běhu algoritmu,
- model počítače, který bude vykonávat analyzované algoritmy,
- jazyk pro popis algoritmů,
- teoretický aparát pro zápis charakteristické doby běhu algoritmu.



## Měření času

- jak změřit čas nezávisle na HW a SW platformě
- jak změřit čas teoreticky, bez implementace programu
- rychlost počítačů se postupně zvyšuje – zvyšuje se i rychlost (kvalita) algoritmů?
- dobu běhu algoritmu tedy nemůžeme měřit v obvyklých časových jednotkách
- dobu běhu algoritmu budeme měřit počtem provedených instrukcí



## Primitivní operace

- nezávislé na programovacím jazyku
- předpokládáme vykonání v konstantním čase
- na různých HW a SW platformách budou časy vykonaní primitivních operací podobné
- získáváme odhad doby běhu programu na základě **počtu** primitivních operací
- přiřazení hodnoty do proměnné
- volání funkce
- provedení aritmetické operace
- porovnání dvou čísel
- indexování pole
- získání reference na objekt
- vrácení výsledku funkce



## RAM stroj – Random Access Machine

- zjednodušený model počítače model počítače
- obsahuje jednoduchý procesor s jedním registrem (akumulátor)
- potenciálně neomezená paměť rozsahu $0 \ldots n$
- základní aritmetické operace
- přímé a nepřímé (pomocí obsahu akumulátoru) adresování paměti
- RAM stroj je ekvivalentní Turingovu stroji
- `http://en.wikipedia.org/wiki/RAM_machine`

## Jazyk pro popis algoritmů

### Nalezení maxima z posloupnosti čísel, zápis v C++

```
1 int Maximum(int[] a, int n)
2 {
3    int max = a[0];
4    for(int i = 0; i < n; i++)
5    {
6       if (a[i] > max)
7       {
8          max = a[i];
9       }
10   }
11   return max;
12 }
```



## Počítání primitivních operací

1. inicializace proměnné max hodnotou `A[0]`,
2. dvě operace na začátku cyklu je proměnná `i` inicializována na 1,
3. jedna operace – před každým provedením cyklu je provedeno porovnání `i < n`, jedna operace $n$-krát,
4. tělo cyklu je provedeno $n - 1$ krát,
5. pokaždé je provedeno porovnání `max < A[i]`, což jsou dvě operace (indexování a porovnání)
6. pokud podmínka platí je provedeno přiřazení, což jsou dvě operace (indexování a přiřazení)
7. na konci těla cyklu se provede inkrementace proměnné `i`, což jsou dvě operace (přičtení a přiřazení)
8. celkově se v těle cyklu vykoná $4(n-1)$ až $6(n-1)$ operací
9. vrácení výsledku odpovídá jedné operaci



## Počítání primitivních operací

Celkový počet primitivních operací je v nejlepším případě

$$2 + 1 + n + 4(n-1) + 1 = 5n$$

v nejhorším případě je to

$$2 + 1 + n + 6(n-1) + 1 = 7n - 2$$

Nejlepší případ nastane pokud `A[0]` je maximem. Nejhorší případ nastane pokud jsou prvky v poli `A` setříděny vzestupně.



## Nejlepší, nejhorší, průměrný případ

Z hlediska časové složitosti lze uvažovat tři případy:

- nejlepší – není zajímavý, nic o podstatě algoritmu nevypovídá
- průměrný – zajímavý, ale jde velice těžko analyzovat, je nutné brát do úvahy pravděpodobnosti výskytu jednotlivých možných vstupů
- nejhorší – není potřeba prát do úvahy pravděpodobnost jak v předchozím případu, analyzuje se jednodušeji, v případě malé složitosti nejhoršího případu, je to pro nás dobrá zpráva

Budeme uvažovat vždy složitost v nejhorším případě, až na některé výjimky.

## Notace „Velké O“

analýza komplexnějších algoritmů než je vyhledání maxima může být velice složitá
musíme se spokojit s odhadem
odhad by se od skutečné složitosti neměl lišit více než o nějakou multiplikativní konstantu
snaha detekovat hlavní faktory, části kódu, které ovlivňují složitost algoritmu
je nutný matematický aparát, který nám umožní skládat složitosti jednotlivých částí kódu a zahrnovat je pod nějaký jednotící celek



## Notace „Velké O“

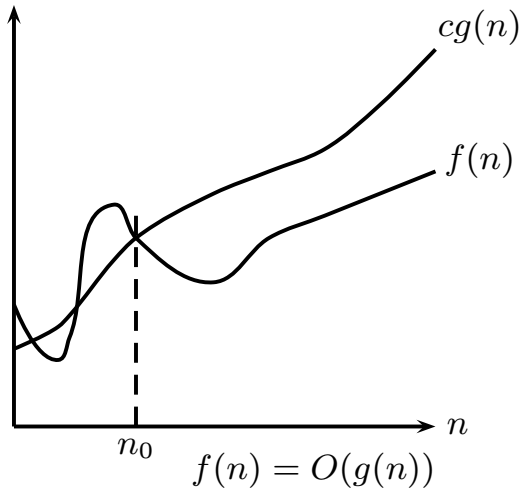
**Definice**

„Velké O“ Mějme funkce $f(n)$ a $g(n)$, kde $f(n), g(n) : N \to R$. Říkáme, že funkce $f(n)$ je $O(g(n))$, jestliže existuje reálná konstanta $c$, $c > 0$ a přirozené číslo $n_0 > 1$ takové, že $f(n) \leq cg(n)$ pro všechna $n \geq n_0$. Můžeme také říci, že f(n) je řádu g(n).

Tato definice nám umožňuje porovnávat funkce. Můžeme říci že jedna funkce „je menší než“ funkce jiná až na konstantní faktor (konstanta $c$ z definice). Funkce lze takto porovnávat **asymptoticky**, čili kdy $n$ roste k nekonečnu (požadavek pro $n \geq n_0$).



## Notace „Velké O“






## Notace „Velké O“ – příklad

**Příklad**

Výraz $7n - 2$ je $O(n)$.

**Řešení**

Podle definice „Velké O“ notace potřebujeme konstantu $c > 0$ a přirozené číslo $n_0 > 1$ takové, že $7n - 2 \leq cn$ pro všechna $n \geq n_0$. Je zřejmé, že lze vzít $c = 7$ a $n_0 = 1$.

**Poznámka**

Řešení je dokonce nekonečně mnoho, protože definici velkého O vyhovuje vzít za konstantu $c$ libovolné reálné číslo větší nebo rovno 7 a za $n_0$ lze zvolit libovolné přirozené číslo větší nebo rovné 1.

## Notace „Velké O“ – příklad

Časová složitost algoritmu vyhledání maxima v poli $n$ čísel je $O(n)$.
Řešení: Víme že v nejhorším případě je pro vyhledání maxima potřeba $7n - 2$ primitivních operací a víme také že výraz $7n - 2$ je $O(n)$. Tudíž algoritmus vyhledání maxima v poli $n$ čísel má složitost $O(n)$.



## Význam asymptotického přístupu

Předpokládejme dále, že elementární operace vykonávané programy trvají $1ms$ a spočítejme, jak rozsáhlá data mohou jednotlivé programy zpracovat za sekundu, za minutu a za hodinu.

| algortimus | složitost | 1s | 1min | 1hod |
|---|---|---|---|---|
| $t_1(n)$ | $n$ | 1000 | $6 \cdot 10^4$ | $3,6 \cdot 10^6$ |
| $t_2(n)$ | $n \log n$ | 140 | 4895 | $2,0 \cdot 10^5$ |
| $t_3(n)$ | $n^2$ | 31 | 244 | 1897 |
| $t_4(n)$ | $n^3$ | 10 | 39 | 153 |
| $t_5(n)$ | $2^n$ | 9 | 15 | 21 |



## Význam asymptotického přístupu

10-násobné zrychlení (nebo zvětšení doby)
lineární složitost – 10-násobným zvětšením rozsahu zpracovávaných dat
kvadratická složitost – 3-násobným zvětšením rozsahu zpracovávaných dat
exponenciální složitost – zvětšení rozsahu dat zhruba **o** $3,3$.
Dosažení rozsahu dat například $n = 50$ u algoritmu $t_5$ zrychlováním (nebo prodlužováním) výpočtu už vůbec nepřichází v úvahu.



## Typické třídy asymptotických složitostí

| | |
|---|---|
| $1$ | konstantní |
| $\log n$ | logaritmická |
| $\log^2 n$ | |
| $\sqrt{n}$ | |
| $n$ | lineární |
| $n \log n$ | lineárně logaritmická |
| $n^2$ | kvadratická |
| $n^3$ | kubická |
| $n^k$ | obecně polynomiální |
| $2^n$ | exponenciální |

Třídy jsou uspořádány vzestupně.

# Děkuji za pozornost



# Pole v jazyce C++

doc. Mgr. Jiří Dvorský, Ph.D.

Katedra informatiky
Fakulta elektrotechniky a informatiky
VŠB – TU Ostrava





## Zadání úlohy

Implementujte funkce pro základní vektorové a maticové výpočty:

- vektory a metice reprezentujte poli s prvky typu `double`,
- dimenze vektorů bude dána konstantou $N$,
- implementujte nulování, součet, skalární součin, délku a odchylku vektorů,
- implementujte součet a násobení matic, vytvoření nulové a jednotkové matice,
- implementujte výstup vektorů a matic na standardní výstup.



## Obecné deklarace

```
1 #include <math.h>
2 #include <iostream>
3
4 using namespace std;
5
6 const int N = 5;
```

## Zobrazení vektoru na standardní výstup

```
1 void Print(double v[], const int n)
2 {
3    for(int i = 0; i < n; i++)
4    {
5       cout << v[i] << "\t";
6    }
7    cout << endl;
8 }
```

Řešení: pomocí cyklu postupně vypisujeme všechny prvky daného vektoru.



## Nulování vektoru

```
1 void Clear(double v[], const int n)
2 {
3    for(int i = 0; i < n; i++)
4    {
5       v[i] = 0;
6    }
7 }
```

Řešení: pomocí cyklu postupně přiřazujeme všem prvkům daného vektoru hodnotu 0.



## Součet vektorů

Součet vektorů je definován jako

$$w_i = u_i + v_i, \forall i = 0, \ldots, n-1$$

```
1 void Add(double u[], double v[], double w[], const int n)
2 {
3    for(int i = 0; i < n; i++)
4    {
5       w[i] = u[i] + v[i];
6    }
7 }
```

Řešení:

- pomocí cyklu postupně sčítáme odpovídající si prvky vektoru $\vec{u}$ a $\vec{v}$,
- součet přiřazujeme do odpovídajícího prvku vektoru $\vec{w}$.



## Délka vektoru

Délku vektoru $\vec{u}$ definujeme jako

$$|\vec{u}| = \sqrt{\sum_{i=0}^{n-1} u_i^2} = u_0^2 + u_1^2 + \cdots + u_{n-1}^2$$

Řešení:

- pomocí cyklu postupně umocňujeme prvky vektoru $\vec{u}$ na druhou – tady stačí použít obyčejné násobení,
- výpočet součtu – jednotlivé druhé mocniny musíme průběžně přičítat do pomocné proměnné,
- součet (suma) tedy vzniká postupně, průběžně jsou do pomocné proměnné „přihazovány“ další a další mezivýsledky,
- nesmíme zapomenout pomocnou proměnnou vynulovat!
- nakonec pomocnou proměnnou odmocníme a vrátíme ji jako výsledek výpočtu.

## Délka vektoru (pokrač.)

```
1 double Length(double u[], const int n)
2 {
3    double sum = 0;
4    for(int i = 0; i < n; i++)
5    {
6       sum += u[i] * u[i];
7    }
8    return sqrt(sum);
9 }
```



## Skalární součin dvou vektorů

Skalární součin vektorů $\vec{u}$ a $\vec{v}$ definujeme jako

$$\vec{u} \cdot \vec{v} = \sum_{i=0}^{n-1} u_i v_i$$

Řešení:

- pomocí cyklu postupně násobíme odpovídající si prvky vektoru $\vec{u}$ a $\vec{v}$,
- výpočet součtu – jednotlivé součiny musíme průběžně přičítat do pomocné proměnné,
- nesmíme zapomenout pomocnou proměnnou vynulovat!
- nakonec pomocnou proměnnou vrátíme jako výsledek výpočtu.



## Skalární součin dvou vektorů (pokrač.)

```
1 double DotProduct(double u[], double v[], const int n)
2 {
3    double sum = 0;
4    for(int i = 0; i < n; i++)
5    {
6       sum += u[i] * v[i];
7    }
8 }
```



## Odchylka dvou vektorů

Odchylku dvou vektorů $\vec{u}$ a $\vec{v}$ (úhel sevřený temito vektory) definujeme jako

$$\cos\varphi = \frac{\vec{u} \cdot \vec{v}}{|\vec{u}||\vec{v}|}$$

Řešení:

- budeme počítat jen $\cos\varphi$, jde tedy o výpočet zlomku,
- v čitateli je skalární součin, ve jmenovateli je součin délek vektorů,
- dvě možnosti implementace:
  1. implementace funkce na „zelené louce“, vše znovu nebo
  2. využijeme již hotové funkce – preferované řešení.

```
1 double VectorsAngle(double u[], double v[], const int n)
2 {
3    return DotProduct(u, v, n) / (Length(u, n) * Length (v, n));
4 }
```

## Zobrazení matice na standardní výstup

Řešení:

- pomocí dvou cyklů postupně vypisujeme všechny prvky dané matice,
- první cyklus iteruje přes řádky matice, druhý přes sloupce matice.

```
1 void Print(double A[][N], const int n)
2 {
3    for(int i = 0; i < n; i++)
4    {
5       for(int j = 0; j < n; j++)
6       {
7          cout << A[i][j] << "\t";
8       }
9       cout << endl;
10   }
11   cout << endl;
12 }
```



## Zobrazení matice na standardní výstup (pokrač.)

**Doporučení**

Nikdy nepoužívejte pro proměnné v cyklu při iterování přes řádky a sloupce jména `x` a `y`! Vždy tradiční `i` a `j`, či `r` a `c`. Jména promměných `x` a `y` budí dojem souřadnic – tyto proměnné mají ale přesně opačný význam!



## Nulová matice

```
1 void Clear(double A[][N], const int n)
2 {
3    for(int i = 0; i < n; i++)
4    {
5       for(int j = 0; j < n; j++)
6       {
7          A[i][j] = 0;
8       }
9    }
10 }
```

Řešení: pomocí dvou cyklů postupně přiřazujeme všem prvkům dané matice hodnotu 0.



## Jednotková matice

```
1 void Unity(double A[][N], const int n)
2 {
3    for(int i = 0; i < n; i++)
4    {
5       for(int j = 0; j < n; j++)
6       {
7          A[i][j] = i == j ? 1 : 0;
8       }
9    }
10 }
```

Řešení: pomocí dvou cyklů postupně přiřazujeme všem prvkům na diagonále dané matice hodnotu 1, jinak 0.

## Součet dvou matic

Součet matic je definován jako

$$C = A + B$$

v zápisu pomocí prvků matice bude součet definován jako

$$c_{i,j} = a_{i,j} + b_{i,j}$$

Řešení:

- pomocí dvou cyklů postupně sčítáme odpovídající si prvky matic $A$ a $B$,
- součet přiřazujeme do odpovídajícího prvku matice $C$.



## Součet dvou matic (pokrač.)

```
1 void Add(double A[][N], double B[][N], double C[][N], const int n)
2 {
3    for(int i = 0; i < n; i++)
4    {
5       for(int j = 0; j < n; j++)
6       {
7          C[i][j] = A[i][j] + B[i][j];
8       }
9    }
10 }
```



## Násobení dvou matic

Násobení dvou matic je definováno po prvcích takto

$$c_{i,j} = \sum_{k=0}^{n-1} a_{i,k} b_{k,j}$$

pro všechny indexy $i$ a $j$. Řešení:

- podíváme-li se na vzorec, je zřejmé, že se jedná o skalární součin řádku $a_i$ matice $A$ se sloupcem $bj$ matice $B$,
- implementace bude vyžadovat cyklus a pomocnou proměnnou,
- dále tento skalární součin musíme provést pro všechny řádky a všechny sloupce,
- budeme tedy potřebovat další dva cykly.



## Násobení dvou matic (pokrač.)

```
1 void Multiply(double A[][N], double B[][N], double C[][N], const int n)
2 {
3    for(int i = 0; i < n; i++)
4    {
5       for(int j = 0; j < n; j++)
6       {
7          double sum = 0;
8          for(int k = 0; k < n; k++)
9          {
10            sum += A[i][k] * B[k][j];
11         }
12         C[i][j] = sum;
13      }
14   }
15 }
```

## Obecný poznatek o sumách

Z předchozích ukázek vzorců a odpovídajících funkcí je patrný obecný závěr, který můžeme neformálně vyjádřit následovně:
**Napíše-li matematik ve vzorci sumu, informatik ji čte jako cyklus.**

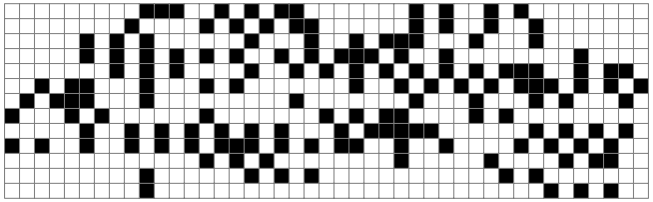


## Game of Life

- autorem je britský matematik John Horton Conway
- publikováno v říjnu 1970
- dvoustavový, dvourozměrný celulární automat, který svým chováním připomíná vývoj společenství živých organismů
- ukázka systému, kde z jednoduchých pravidel vzniká komplexní chování - objevují se zde tzv. emergentní struktury



## Popis hry



- hrací plochou je 2D čtvercová mřížka
- políčka mřížky se nazývají buňky, každá buňka je buď živá (zobrazena černě) nebo mrtvá (zobrazena bíle), buňky mimo mřížku jsou mrtvé
- každá buňka sousedí s dalšími buňkami, se kterými se dotýká hranou (ortogonálně) či vrcholem (diagonálně) – Moorovo okolí
- kolonie buněk v mřížce přežívá z generace na generaci
- stav buňky v nové generaci závisí na jejím současném stavu a počtu živých sousedů



## Pravidla hry

Hra má následující pravidla:

1. Každá živá buňka s méně než dvěma živými sousedy nebo s více než třemi živými sousedy zemře. V příští generaci bude označena jako mrtvá.
2. Každá živá buňka se dvěma nebo třemi živými sousedy zůstává v příští generaci žít.
3. Každá mrtvá buňka s právě třemi živými sousedy v příští generaci obživne.

**Poznámky**

- Stav všech buněk se mění naráz, paralelně.
- Řešení na sekvenčním počítači – ze současné generace vypočteme generaci novou, kterou poté nahradíme starou generaci.

## Ukázka výpočtu nové generace buněk

Stávající generace buněk

| | | | | | |
|---|---|---|---|---|---|
| 0 | 0 | 1 | 1 | 1 | 0 |
| 1 | 1 | 2 | **1** | 2 | 1 |
| 2 | **1** | 3 | 3 | **2** | 2 |
| 2 | **2** | 3 | 3 | **1** | 2 |
| 1 | 2 | **1** | 2 | 1 | 1 |
| 0 | 1 | 1 | 1 | 0 | 0 |

Nová generace buněk

| | | | | | |
|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 1 | 2 | 3 | 2 | 1 |
| 1 | 3 | **4** | **4** | **2** | 1 |
| 1 | **2** | **4** | **4** | 3 | 1 |
| 1 | 2 | 3 | 2 | 1 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 |



## Animace života kolonie

Animace jsou v poznámkách k prezentaci vynechány.



## Vzory vznikající v Game of Life

Přestože pravidla hry jsou velice jednoduchá, vznikají ve hře složité vzory:

Zátiší (Still life) – stabilní konfigurace buněk.

Oscilátory (Oscillators) – nestabilní konečný vzor, který periodicky přechází mezi několika konfiguracemi.

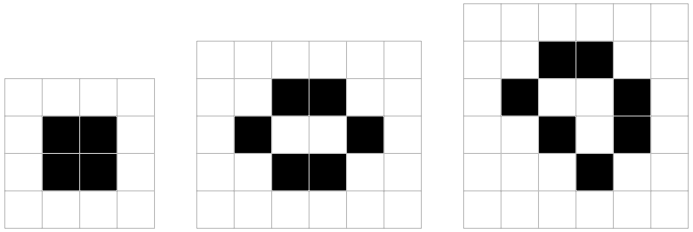
Děla (Guns) – vzor rostoucí do nekonečna

Metuzalémové (Methuselahs) – relativně malý vzor, jehož stabilizace trvá naopak velmi velký počet generací.

Výčet není konečný...



## Zátiší (Still life)



Blok (Block) Včelín (Beehive) Bochník (Loaf)

## Oscilátory (Oscillators)

Blikač (Blinker), perioda 2

Animace jsou v poznámkách k prezentaci vynechány.



## Oscilátory (Oscillators) (pokrač.)

Ropucha (Toad), perioda 2

Animace jsou v poznámkách k prezentaci vynechány.



## Oscilátory (Oscillators) (pokrač.)

Pulsar, perioda 3

Animace jsou v poznámkách k prezentaci vynechány.



## Oscilátory (Oscillators) (pokrač.)

Pentadecathlon, perioda 15

Animace jsou v poznámkách k prezentaci vynechány.

## Děla – Gosper glider gun

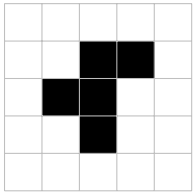
Animace jsou v poznámkách k prezentaci vynechány.

- Dělo každých 30 generací vypustí kluzák.
- Kluzák se skládá ze 5 buněk, má 4 konfigurace, pohybuje se rychlostí $\frac{1}{4}c$, kde $c$ je rychlost světla v tomto „vesmíru“ – jedna buňka za jednu generaci.



## Metuzalémové – R-pentomino

**Počáteční konfigurace**



Stabilizace kolonie
po 1103 generacích

Animace jsou v poznámkách k prezentaci vynechány.



## Chaotické chování

Animace jsou v poznámkách k prezentaci vynechány.

- Infikujeme oscilátor Pulsar jednou buňkou navíc, označena červeně.
- Drobná změna na počátku způsobila diametrálně odlišné chování jinak periodického vzoru.
- Některé „infekce“, ale odlišné chování nezpůsobí. Například buňka uprostřed vzoru by bez následků hned v druhé generaci zanikla.



## Zadání úlohy

**Zadání úlohy**

Vytvořte program, který ve čtvercové mřížce dané velikosti pro náhodně vygenerovanou kolonii s daným počtem buněk vypočítá a vypíše na standardní výstup daný počet následujících generací.

**Parametry úlohy**

- $N$ – velikost mřížky, mřížka bude mít rozměry $N \times N$,
- $p$ – počet živých buněk v počáteční kolonii a
- $g$ – požadovaný počet generací.

Pro demonstraci můžeme například použít tyto konkrétní hodnoty: $N = 10$, $p = 50$ a $g = 10$.

## Nástin řešení

1. Vygeneruj počáteční náhodnou kolonii buněk.
2. Zobraz původní kolonii buněk na standardní výstup.
3. Opakuj pro daný počet generací:
   - 3.1 Vypočti novou generaci kolonie buněk, tj. pro každou buňku v současné kolonii proveď:
     - 3.1.1 zjisti počet živých buněk v jejím okolí,
     - 3.1.2 aplikuj pravidlo pro přežití, zánik nebo zrození buňky a
     - 3.1.3 nový stav buňky zaznamenej do nové kolonie.
   - 3.2 Přepiš současnou kolonii buněk nově vypočtenou.
   - 3.3 Zobraz současnou kolonii buněk na standardní výstup.
4. Konec



## Nástin řešení – otázky a odpovědi

**Otázky:**

1. Jak reprezentovat kolonii buněk?
2. Jak vygenerovat náhodnou kolonii buněk?
3. Zobrazení kolonie na std. výstup, výpočet nové generace a přepis staré kolonie vyžaduje provést jistou akci pro každou buňku v kolonii. Jak toto provést?
4. Jak implementovat pravidla pravidla hry (přežití, zánik nebo zrod buňky)?

**Odpovědi:**

1. Nejlépe dvourozměrným polem logických hodnot.
2. Použijeme funkci `rand()` generující náhodná čísla plus úprava rozsahu vygenerovaných hodnot na $0, \ldots, N-1$.
3. Dvěma vnořenými cykly: jeden přes všechny řádky, druhý přes všechny sloupce.
4. Logickým výrazem, který bude pravdivý v okamžiku, kdy má být buňka v další generaci živá.



## Reprezentace kolonie buněk

- Kolonii buněk reprezentujeme polem `bool a[N][N]`, kde hodnota `true` bude reprezentovat **živou** buňku a hodnota `false` **mrtvou** buňku.
- Okolí buňky `a[j][k]` definujeme následujícím schématem:

| | | |
|---|---|---|
| `a[j-1][k-1]` | `a[j-1][k]` | `a[j-1][k+1]` |
| `a[j][k-1]` | `a[j][k]` | `a[j][k+1]` |
| `a[j+1][k-1]` | `a[j+1][k]` | `a[j+1][k+1]` |

- Tento druh okolí se někdy také nazývá Moorovo okolí.
- Při implementaci musíme dávat pozor, abychom nepřistupovali k neexistujícím buňkám.



## Průchod všemi buňkami v kolonii

```
1 for(int i = 0; i < N; i++)
2 {
3   for(int j = 0; j < N; j++)
4   {
5     // some action for a[i][j]
6   }
7 }
```

- **Vnořené cykly** – cyklus je „vnořen“, „zanořen“, či „vložen“ do těla jiného cyklu.
- Cyklus s řídící proměnnou `i` je tzv. **vnější cyklus**, cyklus s řídící proměnnou `j` je tzv. **vnitřní cyklus**.
- Počet vnořených cyklů není omezen, s rostoucím vnořením ale klesá čitelnost kódu.
- Vnější cyklus probíhá přes všechny řádky kolonie, vnitřní přes všechny sloupce kolonie buněk.

## Generování náhodné kolonie buněk

Náhodnou kolonii vygenerujeme ve dvou krocích:

1. Projdeme všechny buňky v kolonii a nastavíme je na hodnotu `false`.
2. Pomocí cyklu `for` budeme $p$ krát opakovat:
   2.1 Vygeneruj náhodné číslo řádku `r = rand()% N`
   2.2 Vygeneruj náhodné číslo sloupce `c = rand()% N`
   2.3 Nastav buňku `a[r][c] = true`

**Poznámky**

- První krok je nezbytný, prvky v poli, buňky, nemají definovanou hodnotu!
- Funkce `rand()` vrací hodnoty v rozsahu `0` až `RAND_MAX` včetně. Protože $N <$`RAND_MAX` upravíme vygenerované náhodné číslo pomocí operace zbytku po dělení číslem $N$.
- V tomto postupu je však jedna malá logická chyba :-).



## Implementace pravidel hry

V nové generaci bude buňka $a_{ij}$ živá, pokud v současné generaci je:

- živá a zároveň má dva nebo tři živé sousedy (přežití) nebo
- mrtvá a zároveň má tři živé sousedy (zrození).

Označme $L$ počet živých sousedů buňky $a_{ij}$, pak přežití můžeme zapsat jako

$$a_{ij} \wedge (L = 2 \vee L = 3)$$

a zrození jako

$$\neg a_{ij} \wedge L = 3$$

Oba logické výrazy spojíme spojkou $\vee$ do jednoho výrazu

$$[a_{ij} \wedge (L = 2 \vee L = 3)] \vee [\neg a_{ij} \wedge L = 3]$$

V jazyce C++ bude tento logický výraz vypadat takto

```
1 a[i][j] && (L == 2 || L == 3) || !a[i][j] && L == 3
```



## Implementace pravidel hry (pokrač.)

Nnyní máme dvě možnosti:

1. použít tento výraz v implementaci tak jak je nebo
2. pokusit se jej zjednodušit.

**Zjednodušení**

Na výraz v první hranaté závorce můžeme aplikovat distributivní zákon a dostáváme

$$[a_{ij} \wedge L = 2 \vee a_{ij} \wedge L = 3] \vee [\neg a_{ij} \wedge L = 3]$$

Přeskupíme závorky

$$(a_{ij} \wedge L = 2) \vee (a_{ij} \wedge L = 3) \vee (\neg a_{ij} \wedge L = 3)$$

Nyní předpokládejme, že $L = 3$. Pokud se podíváme na druhou a třetí závorku, zjistíme že:

- pokud $a_{ij}$ má hodnotu 1, `true`, platí druhá „závorka“ nebo



## Implementace pravidel hry (pokrač.)

- pokud $a_{ij}$ má hodnotu 0, `false`, platí třetí „závorka“.

Je jasné, že na hodnotě $a_{ij}$ nezáleží, záleží jen na tom, zda je $L = 3$. Formálně bychom mohli postupovat tak, že nejprve aplikujeme na druhou a třetí závorku distributivní zákon

$$(a_{ij} \wedge L = 2) \vee [(a_{ij} \vee \neg a_{ij}) \wedge (L = 3)]$$

Podle zákona o vyloučení třetího má výraz $a_{ij} \vee \neg a_{ij}$ vždy hodnotu `true`, 1. Dosazením dostáváme

$$(a_{ij} \wedge L = 2) \vee [1 \wedge (L = 3)]$$

Dále podle zákona o neutralitě 1 v konjunkci dostáváme

$$(a_{ij} \wedge L = 2) \vee [L = 3]$$

## Implementace pravidel hry (pokrač.)

Závorky v tomto výrazu můžeme zcela vynechat a dostaneme konečnou podobu výrazu

$$a_{ij} \wedge L = 2 \vee L = 3$$

V jazyce C++ bude tento logický výraz vypadat takto

```
1 a[i][j] && L == 2 || L == 3
```



## Řešení verze 1 – na formátování přece nezáleží!

```
1  #include <iostream>
2
3    using namespace std;
4
5          void main()
6          {
7          bool a[10][10], b[10][10];
8  int i, j, k, l;
9             for (i = 0; i < 10; i++)
10         for (j = 0; j < 10; j++)
11 a[i][j] = false;
12
13            for (i = 0; i < 50; i++)
14       {
15               j = rand() % 10;
16       k = rand() % 10;
17    a[j][k] = true;
18         }
19    cout << "Puvodni kolonie" << endl;
20               for (i = 0; i < 10; i++)
21         {
22               for (j = 0; j < 10; j++)
23       {
24            if (a[i][j])
25 cout << "X";
26       else
27            cout << ".";
28
29    }
30    cout << endl;
```



## Řešení verze 1 – na formátování přece nezáleží! (pokrač.)

```
31              }
32              cout << endl;
33    for (i = 1; i <= 10; i++)
34
35
36
37       {
38    for (j = 0; j < 10; j++)
39 for (k = 0; k < 10; k++)
40              {
41             l = 0;
42    if (0 <= (j - 1) && (j - 1) < 10 && 0 <= (k - 1) && (k - 1) < 10 && a[j - 1][k - 1])
43          l += 1;
44             if (0 <= (j - 1) && (j - 1) < 10 && 0 <= k && k < 10 && a[j - 1][k])
45          l += 1;
46    if (0 <= (j - 1) && (j - 1) < 10 && 0 <= (k + 1) && (k + 1) < 10 && a[j - 1][k + 1])
47          l += 1;
48    if (0 <= j && j < 10 && 0 <= (k - 1) && (k - 1) < 10 && a[j][k - 1])
49             l += 1;
50       if (0 <= j && j < 10 && 0 <= (k + 1) && (k + 1) < 10 && a[j][k + 1])
51    l += 1;
52             if (0 <= (j + 1) && (j + 1) < 10 && 0 <= (k - 1) && (k - 1) < 10 && a[j + 1][k - 1])
53
54
55
56             l += 1;
57          if (0 <= (j + 1) && (j + 1) < 10 && 0 <= k && k < 10 && a[j + 1][k])
58 l += 1;
59             if (0 <= (j + 1) && (j + 1) < 10 && 0 <= (k + 1) && (k + 1) < 10 && a[j + 1][k + 1])
60       l += 1;
```



## Řešení verze 1 – na formátování přece nezáleží! (pokrač.)

```
61             if (a[j][k] && l == 2 || l == 3)
62
63
64    b[j][k] = true;
65
66          else
67                b[j][k] = false;
68       }
69 for (j = 0; j < 10; j++)
70                for (k = 0; k < 10; k++)
71 a[j][k] = b[j][k];
72       cout << "Generace #" << i << endl;
73             for (j = 0; j < 10; j++)
74 {
75                for (k = 0; k < 10; k++)
76          {
77       if (a[j][k])
78    cout << "X";
79       else
80
81 cout << ".";
82    }
83
84    cout << endl;
85 }
86
87    cout << endl;
88          }
89       }
```

# Řešení verze 2 – zdrojový kód je přece pro počítač!

```
#include <iostream>
using namespace std;
void main(){bool a[10][10],b[10][10];int i,j,k,l;for(i=0;
i<10;i++)for(j=0;j<10;j++)a[i][j]=false;for(i=0;i<50;
i++){j=rand()%10;k=rand()%10;a[j][k]=true;}cout<<
"Puvodni kolonie"<<endl;for(i=0;i<10;i++){for(j=0;j<10;
j++){if(a[i][j])cout<<"X";else cout<<".";}cout<<endl
;}cout<<endl;for(i=1;i<= 10;i++){for(j=0;j<10;j++
)for(k=0;k<10;k++){l=0;if(0<=(j-1)&&(j-1)<10&&0<=
(k-1)&&(k-1)<10&&a[j-1][k-1])l+=1;if(0<=(j-1)&&(j-1)
<10&&0<= k&&k<10&&a[j-1][k])l+=1;if(0<=(j-1)&&(j-1)
<10&&0<=(k+1)&&(k+1)<10&&a[j-1][k+1])l+=1;if(0<=j
&&j<10&&0<=(k-1)&&(k-1)<10&&a[j][k-1])l+=1;if(0<=
j&&j<10&&0<=(k+1)&&(k+1)<10&&a[j][k+1])l+=1;if(0<=(
j+1)&&(j+1)<10&&0<=(k-1)&&(k-1)<10&&a[j+1][k-1])l
+=1;if(0<=(j+1)&&(j+1)<10&&0<= k&&k<10&&a[j+1][k
])l+=1;if(0<=(j+1)&&(j+1)<10&&0<=(k+1)&&(k+1)<10
&&a[j+1][k+1])l+=1;if(a[j][k]&&l==2||l == 3)b[j]
[k]=true;else b[j][k]=false;}for(j=0;j<10;j++)for
(k=0;k<10;k++)a[j][k]=b[j][k];cout<<"Generace #"
<<i<<endl;for(j=0;j<10;j++){for(k=0;k<10;k++){if(a[j]
[k])cout<<"X";else cout<<".";}cout<<endl;}cout<<endl;}}
```



# Řešení verze 3 – tohle je přece jen čitelnější, ne?

```
#include <iostream>

using namespace std;

void main()
{
   bool a[10][10], b[10][10];
   int i, j, k, l;
   for (i = 0; i < 10; i++)
      for (j = 0; j < 10; j++)
         a[i][j] = false;
   for (i = 0; i < 50; i++)
   {
      j = rand() % 10;
      k = rand() % 10;
      a[j][k] = true;
   }
   cout << "Puvodni kolonie" << endl;
   for (i = 0; i < 10; i++)
   {
      for (j = 0; j < 10; j++)
      {
         if (a[i][j])
            cout << "X";
         else
            cout << ".";
      }
      cout << endl;
   }
   cout << endl;
```



# Řešení verze 3 – tohle je přece jen čitelnější, ne? (pokrač.)

```
   for (i = 1; i <= 10; i++)
   {
      for (j = 0; j < 10; j++)
         for (k = 0; k < 10; k++)
         {
            l = 0;
            if (0 <= (j - 1) && (j - 1) < 10 && 0 <= (k - 1) && (k - 1) < 10 && a[j - 1][k - 1])
               l += 1;
            if (0 <= (j - 1) && (j - 1) < 10 && 0 <= k && k < 10 && a[j - 1][k])
               l += 1;
            if (0 <= (j - 1) && (j - 1) < 10 && 0 <= (k + 1) && (k + 1) < 10 && a[j - 1][k + 1])
               l += 1;
            if (0 <= j && j < 10 && 0 <= (k - 1) && (k - 1) < 10 && a[j][k - 1])
               l += 1;
            if (0 <= j && j < 10 && 0 <= (k + 1) && (k + 1) < 10 && a[j][k + 1])
               l += 1;
            if (0 <= (j + 1) && (j + 1) < 10 && 0 <= (k - 1) && (k - 1) < 10 && a[j + 1][k - 1])
               l += 1;
            if (0 <= (j + 1) && (j + 1) < 10 && 0 <= k && k < 10 && a[j + 1][k])
               l += 1;
            if (0 <= (j + 1) && (j + 1) < 10 && 0 <= (k + 1) && (k + 1) < 10 && a[j + 1][k + 1])
               l += 1;
            if (a[j][k] && l == 2 || l == 3)
               b[j][k] = true;
            else
               b[j][k] = false;
         }
      for (j = 0; j < 10; j++)
         for (k = 0; k < 10; k++)
            a[j][k] = b[j][k];
```



# Řešení verze 3 – tohle je přece jen čitelnější, ne? (pokrač.)

```
      cout << "Generace #" << i << endl;
      for (j = 0; j < 10; j++)
      {
         for (k = 0; k < 10; k++)
         {
            if (a[j][k])
               cout << "X";
            else
               cout << ".";
         }
         cout << endl;
      }
      cout << endl;
   }
}
```

## Dosavadní řešení

Verze 1 – velice obtížně čitelné, ale přesto text připomíná vzdáleně zdrojový kód v C++.

Verze 2 – troufne si někdo odhadnout co tento zdrojový kód vlastně implementuje? Nebo v tomto zdrojovém kódu najít chybu?

Verze 3 – dosud nejlepší zdrojový kód

- Kladné rysy
  1. přijatelná čitelnost díky formátování zdrojového kódu
- Záporné rysy
  1. nulové strukturování zdrojového kódu – veškerý kód ve funkci `main`,
  2. přímo zapsané číselné hodnoty
  3. a mnoho dalších ...

Pokusíme se proto zdrojový kód zlepšit.



## Přímo zapsané číselné hodnoty

```
for (i = 1; i <= 10; i++)
{
   for (j = 0; j < 10; j++)
      for (k = 0; k < 10; k++)
      {
         l = 0;
         if (0 <= (j - 1) && (j - 1) < 10 && 0 <= (k - 1) && (k - 1) <
              10 && a[j - 1][k - 1] == 1) l += 1;
```

- Jaký je význam jednotlivých čísel 10?
- V zadání je 10 generací, velikost kolonie $10 \times 10$ buněk.
- Změníme zadání a budeme chtít vypsat 20 generací. Kterou desítku máme přepsat a kterou ne? Přepsali jsme všechny?

Řešením je použít místo přímo zapsaných číselných hodnot konstanty.



## Řešení verze 4 – konstanty

```
#include <iostream>

using namespace std;

const int NumberOfRows = 10;
const int NumberOfColumns = 10;
const int NumberOfGenerations = 10;
const int NumberOfLivingCells = 50;

void main()
{
   bool a[NumberOfRows][NumberOfColumns], b[NumberOfRows][NumberOfColumns];
   int i, j, k, l;
   for (i = 0; i < NumberOfRows; i++)
      for (j = 0; j < NumberOfColumns; j++)
         a[i][j] = false;
   for (i = 0; i < NumberOfLivingCells; i++)
   {
      j = rand() % NumberOfRows;
      k = rand() % NumberOfColumns;
      a[j][k] = true;
   }
   cout << "Puvodni kolonie" << endl;
   for (i = 0; i < NumberOfRows; i++)
   {
      for (j = 0; j < NumberOfColumns; j++)
      {
         if (a[i][j])
            cout << "X";
         else
```



## Řešení verze 4 – konstanty (pokrač.)

```
            cout << ".";
      }
      cout << endl;
   }
   cout << endl;
   for (i = 1; i <= NumberOfGenerations; i++)
   {
      for (j = 0; j < NumberOfRows; j++)
         for (k = 0; k < NumberOfColumns; k++)
         {
            l = 0;
            if (0 <= (j - 1) && (j - 1) < NumberOfRows && 0 <= (k - 1) && (k - 1) < NumberOfColumns && a[j
                 - 1][k - 1])
               l += 1;
            if (0 <= (j - 1) && (j - 1) < NumberOfRows && 0 <= k && k < NumberOfColumns && a[j - 1][k])
               l += 1;
            if (0 <= (j - 1) && (j - 1) < NumberOfRows && 0 <= (k + 1) && (k + 1) < NumberOfColumns && a[j
                 - 1][k + 1])
               l += 1;
            if (0 <= j && j < NumberOfRows && 0 <= (k - 1) && (k - 1) < NumberOfColumns && a[j][(k - 1)])
               l += 1;
            if (0 <= j && j < NumberOfRows && 0 <= (k + 1) && (k + 1) < NumberOfColumns && a[j][k + 1])
               l += 1;
            if (0 <= (j + 1) && (j + 1) < NumberOfRows && 0 <= (k - 1) && (k - 1) < NumberOfColumns && a[j
                 + 1][k - 1])
               l += 1;
            if (0 <= (j + 1) && (j + 1) < NumberOfRows && 0 <= k && k < NumberOfColumns && a[j + 1][k])
               l += 1;
            if (0 <= (j + 1) && (j + 1) < NumberOfRows && 0 <= (k + 1) && (k + 1) < NumberOfColumns && a[j
                 + 1][k + 1])
```

## Řešení verze 4 – konstanty (pokrač.)

```
57              l += 1;
58           if (a[j][k] && l == 2 || l == 3)
59              b[j][k] = true;
60           else
61              b[j][k] = false;
62        }
63     for (j = 0; j < NumberOfRows; j++)
64        for (k = 0; k < NumberOfColumns; k++)
65           a[j][k] = b[j][k];
66     cout << "Generace #" << i << endl;
67     for (j = 0; j < NumberOfRows; j++)
68     {
69        for (k = 0; k < NumberOfColumns; k++)
70        {
71           if (a[j][k])
72              cout << "X";
73           else
74              cout << ".";
75        }
76        cout << endl;
77     }
78     cout << endl;
79  }
80 }
```



## Opakující se části kódu

- Prozkoumejme řádky 42 až 57 – celkem 8(!) velice podobných řádků.
- Řádky testují všech 8 buněk v Moorově okolí.

| | | |
|---|---|---|
| a[j-1][k-1] | a[j-1][k] | a[j-1][k+1] |
| a[j][k-1] | a[j][k] | a[j][k+1] |
| a[j+1][k-1] | a[j+1][k] | a[j+1][k+1] |

- Opakující se řádky kódu se liší jen $\pm 1$ v indexech.
- Nebylo by vhodnější řešit opakování cyklem či cykly?
- Cykly $p = j - 1, \ldots, j + 1$ a $q = k - 1, \ldots k + 1$?
- Vnější cyklus proběhne $3\times$ a vnitřní také $3\times$, celkem $9\times$. Moorovo okolí má ale jen 8 buněk.
- Musíme totiž vyloučit případ kdy $p = j \vee q = k$!



## Opakující se části kódu (pokrač.)

```
1 for (p = j - 1; p <= j + 1; p++)
2    for (q = k - 1; q <= k + 1; q++)
3       if ((p != j || q != k) && 0 <= p && p < NumberOfRows && 0 <=
             q && q < NumberOfColumns && a[p][q])
4          l += 1;
```



## Řešení verze 5 – opakující se části kódu

```
1  #include <iostream>
2
3  using namespace std;
4
5  const int NumberOfRows = 10;
6  const int NumberOfColumns = 10;
7  const int NumberOfGenerations = 10;
8  const int NumberOfLivingCells = 50;
9
10 void main()
11 {
12    bool a[NumberOfRows][NumberOfColumns], b[NumberOfRows][NumberOfColumns];
13    int i, j, k, l, p, q;
14    for (i = 0; i < NumberOfRows; i++)
15       for (j = 0; j < NumberOfColumns; j++)
16          a[i][j] = false;
17    for (i = 0; i < NumberOfLivingCells; i++)
18    {
19       j = rand() % NumberOfRows;
20       k = rand() % NumberOfColumns;
21       a[j][k] = true;
22    }
23    cout << "Puvodni kolonie" << endl;
24    for (i = 0; i < NumberOfRows; i++)
25    {
26       for (j = 0; j < NumberOfColumns; j++)
27       {
28          if (a[i][j])
29             cout << "X";
30          else
```

## Řešení verze 5 – opakující se části kódu (pokrač.)

```
31            cout << ".";
32         }
33         cout << endl;
34      }
35      cout << endl;
36      for (i = 1; i <= NumberOfGenerations; i++)
37      {
38         for (j = 0; j < NumberOfRows; j++)
39            for (k = 0; k < NumberOfColumns; k++)
40            {
41               l = 0;
42               for (p = j - 1; p <= j + 1; p++)
43                  for (q = k - 1; q <= k + 1; q++)
44                     if ((p != j || q != k) && 0 <= p && p < NumberOfRows && 0 <= q && q < NumberOfColumns &&
                           a[p][q])
45                        l += 1;
46               if (a[j][k] && l == 2 || l == 3)
47                  b[j][k] = true;
48               else
49                  b[j][k] = false;
50            }
51         for (j = 0; j < NumberOfRows; j++)
52            for (k = 0; k < NumberOfColumns; k++)
53               a[j][k] = b[j][k];
54         cout << "Generace #" << i << endl;
55         for (j = 0; j < NumberOfRows; j++)
56         {
57            for (k = 0; k < NumberOfColumns; k++)
58            {
59               if (a[j][k])
```



## Řešení verze 5 – opakující se části kódu (pokrač.)

```
60                  cout << "X";
61               else
62                  cout << ".";
63            }
64            cout << endl;
65         }
66         cout << endl;
67      }
68   }
```



## Nepřehledná podmínka

Podívejme se ještě jednou na podmínku uvnitř cyklů

```
for (p = j - 1; p <= j + 1; p++)
   for (q = k - 1; q <= k + 1; q++)
      if ((p != j || q != k) && 0 <= p && p < NumberOfRows && 0 <= q &&
           q < NumberOfColumns && a[p][q])
         l += 1;
```

- Je složena z celkem šesti nerovností a jedné rovnosti – což není příliš přehledné.
- Zkoumá se tu najednou několik záležitostí:
  1. vynechání buňky ve středu okolí (`p != j || q != k`)
  2. test existence buňky
     `0 <= p && p < NumberOfRows && 0 <= q && q < NumberOfColumns`
  3. test zda je buňka živá `a[p][q]`
- Pokud toto vše platí zároveň, inkrementuje se počitadlo `l` živých buněk v Moorově okolí.



## Nepřehledná podmínka (pokrač.)

Řešením je přesun některých podmínek do samostatné funkce.

```
bool IsCellAlive(const bool Colony[NumberOfRows][NumberOfColumns],
     const int Row, const int Column)
{
   if (0 <= Row && Row < NumberOfRows && 0 <= Column && Column <
       NumberOfColumns)
   {
      return Colony[Row][Column];
   }
   return false;
}
```

## Řešení verze 6 – test živosti buňky jako funkce

```
1  #include <iostream>
2
3  using namespace std;
4
5  const int NumberOfRows = 10;
6  const int NumberOfColumns = 10;
7  const int NumberOfGenerations = 10;
8  const int NumberOfLivingCells = 50;
9
10 bool IsCellAlive(const bool Colony[NumberOfRows][NumberOfColumns], const int Row, const int Column)
11 {
12    if (0 <= Row && Row < NumberOfRows && 0 <= Column && Column < NumberOfColumns)
13    {
14       return Colony[Row][Column];
15    }
16    return false;
17 }
18
19 void main()
20 {
21    bool a[NumberOfRows][NumberOfColumns], b[NumberOfRows][NumberOfColumns];
22    int i, j, k, l, p, q;
23    for (i = 0; i < NumberOfRows; i++)
24       for (j = 0; j < NumberOfColumns; j++)
25          a[i][j] = false;
26    for (i = 0; i < NumberOfLivingCells; i++)
27    {
28       j = rand() % NumberOfRows;
29       k = rand() % NumberOfColumns;
30       a[j][k] = true;
```



## Řešení verze 6 – test živosti buňky jako funkce (pokrač.)

```
31    }
32    cout << "Puvodni kolonie" << endl;
33    for (i = 0; i < NumberOfRows; i++)
34    {
35       for (j = 0; j < NumberOfColumns; j++)
36       {
37          if (a[i][j])
38             cout << "X";
39          else
40             cout << ".";
41       }
42       cout << endl;
43    }
44    cout << endl;
45    for (i = 1; i <= NumberOfGenerations; i++)
46    {
47       for (j = 0; j < NumberOfRows; j++)
48          for (k = 0; k < NumberOfColumns; k++)
49          {
50             l = 0;
51             for (p = j - 1; p <= j + 1; p++)
52                for (q = k - 1; q <= k + 1; q++)
53                   if ((p != j || q != k) && IsCellAlive(a, p, q))
54                      l += 1;
55             if (a[j][k] && l == 2 || l == 3)
56                b[j][k] = true;
57             else
58                b[j][k] = false;
59          }
60       for (j = 0; j < NumberOfRows; j++)
```



## Řešení verze 6 – test živosti buňky jako funkce (pokrač.)

```
61          for (k = 0; k < NumberOfColumns; k++)
62             a[j][k] = b[j][k];
63       cout << "Generace #" << i << endl;
64       for (j = 0; j < NumberOfRows; j++)
65       {
66          for (k = 0; k < NumberOfColumns; k++)
67          {
68             if (a[j][k])
69                cout << "X";
70             else
71                cout << ".";
72          }
73          cout << endl;
74       }
75       cout << endl;
76    }
77 }
```



## Počet vnořených cyklů

Podívejme se ještě jednou cykly řešící naši úlohu:

```
1 for (i = 1; i <= NumberOfGenerations; i++)
2 {
3    for (j = 0; j < NumberOfRows; j++)
4       for (k = 0; k < NumberOfColumns; k++)
5       {
6          l = 0;
7          for (p = j - 1; p <= j + 1; p++)
8             for (q = k - 1; q <= k + 1; q++)
9                if ((p != j || q != k) && IsCellAlive(a, p, q))
```

Odstranili jsme sice osminásobné opakování téměř identické podmínky, ale přibyly nám další dva vnořené cykly. **Pět** (!!!) vnořených cyklů není asi zrovna přehledné řešení.

## Počet vnořených cyklů (pokrač.)

**Úrovně vnoření:**

1. iterace přes generace kolonie,
2. iterace přes řádky kolonie,
3. iterace přes sloupce kolonie,
4. iterace přes řádky Moorova okolí a
5. iterace přes sloupce Moorova okolí aktuální buňky.

**Řešení?**

- Iterace přes všechny buňky Moorova okolí přesunout do samostatné funkce.
- Tyto dva cykly řeší jen lokální, samostatnou úlohu.



## Počet vnořených cyklů (pokrač.)

```
1 int NumberOfLivingNeighbors(const bool Colony[NumberOfRows][
      NumberOfColumns], const int Row, const int Column)
2 {
3    int Counter = 0;
4    for (int i = Row - 1; i <= Row + 1; i++)
5    {
6       for (int j = Column - 1; j <= Column + 1; j++)
7       {
8          if ((i != Row || j != Column) && IsCellAlive(Colony, i, j))
9          {
10            Counter += 1;
11         }
12      }
13   }
14   return Counter;
15 }
```



## Řešení verze 8 – počet živých sousedů implementován jako funkce

```
1  #include <iostream>
2
3  using namespace std;
4
5  const int NumberOfRows = 10;
6  const int NumberOfColumns = 10;
7  const int NumberOfGenerations = 10;
8  const int NumberOfLivingCells = 50;
9
10 bool IsCellAlive(const bool Colony[NumberOfRows][NumberOfColumns], const int Row, const int Column)
11 {
12    if (0 <= Row && Row < NumberOfRows && 0 <= Column && Column < NumberOfColumns)
13    {
14       return Colony[Row][Column];
15    }
16    return false;
17 }
18
19 int NumberOfLivingNeighbors(const bool Colony[NumberOfRows][NumberOfColumns], const int Row, const int
       Column)
20 {
21    int Counter = 0;
22    for (int i = Row - 1; i <= Row + 1; i++)
23    {
24       for (int j = Column - 1; j <= Column + 1; j++)
25       {
26          if ((i != Row || j != Column) && IsCellAlive(Colony, i, j))
27          {
```



## Řešení verze 8 – počet živých sousedů implementován jako funkce (pokrač.)

```
28             Counter += 1;
29          }
30       }
31    }
32    return Counter;
33 }
34
35 void main()
36 {
37    bool a[NumberOfRows][NumberOfColumns], b[NumberOfRows][NumberOfColumns];
38    int i, j, k, l, p, q;
39    for (i = 0; i < NumberOfRows; i++)
40       for (j = 0; j < NumberOfColumns; j++)
41          a[i][j] = false;
42    for (i = 0; i < NumberOfLivingCells; i++)
43    {
44       j = rand() % NumberOfRows;
45       k = rand() % NumberOfColumns;
46       a[j][k] = true;
47    }
48    cout << "Puvodni kolonie" << endl;
49    for (i = 0; i < NumberOfRows; i++)
50    {
51       for (j = 0; j < NumberOfColumns; j++)
52       {
53          if (a[i][j])
54             cout << "X";
```

## Řešení verze 8 – počet živých sousedů implementován jako funkce (pokrač.)

```
55          else
56             cout << ".";
57       }
58       cout << endl;
59    }
60    cout << endl;
61    for (i = 1; i <= NumberOfGenerations; i++)
62    {
63       for (j = 0; j < NumberOfRows; j++)
64          for (k = 0; k < NumberOfColumns; k++)
65          {
66             l = NumberOfLivingNeighbors(a, j, k);
67             if (a[j][k] && l == 2 || l == 3)
68                b[j][k] = true;
69             else
70                b[j][k] = false;
71          }
72       for (j = 0; j < NumberOfRows; j++)
73          for (k = 0; k < NumberOfColumns; k++)
74             a[j][k] = b[j][k];
75       cout << "Generace #" << i << endl;
76       for (j = 0; j < NumberOfRows; j++)
77       {
78          for (k = 0; k < NumberOfColumns; k++)
79          {
80             if (a[j][k])
81                cout << "X";
```



## Řešení verze 8 – počet živých sousedů implementován jako funkce (pokrač.)

```
82             else
83                cout << ".";
84          }
85          cout << endl;
86       }
87       cout << endl;
88    }
89 }
```



## Zjednodušení výpočtu stavu buňky v nové generaci

Podívejme se na výpočet stavu nové buňky

```
1 if (a[j][k] && l == 2 || l == 3)
2    b[j][k] = true;
3 else
4    b[j][k] = false;
```

Co tato konstrukce znamená?

- Pokud je výraz v podmínce pravdivý, `true`, nastavíme `b[j][k]` na `true`.
- Pokud je nepravdivý, `false`, nastavíme `b[j][k]` na `false`.

Přiřazení tak vlastně jen kopíruje výsledek výrazu v podmínce. Nešlo by to `b[j][k]` přiřadit rovnou výsledek výrazu v podmínce? Ano, šlo.

```
1 b[j][k] = a[j][k] && l == 2 || l == 3;
```



## Řešení verze 9 – zjednodušení výpočtu stavu buňky v nové generaci

```
1  #include <iostream>
2
3  using namespace std;
4
5  const int NumberOfRows = 10;
6  const int NumberOfColumns = 10;
7  const int NumberOfGenerations = 10;
8  const int NumberOfLivingCells = 50;
9
10 bool IsCellAlive(const bool Colony[NumberOfRows][NumberOfColumns], const int Row, const int Column)
11 {
12    if (0 <= Row && Row < NumberOfRows && 0 <= Column && Column < NumberOfColumns)
13    {
14       return Colony[Row][Column];
15    }
16    return false;
17 }
18
19 int NumberOfLivingNeighbors(const bool Colony[NumberOfRows][NumberOfColumns], const int Row, const int
       Column)
20 {
21    int Counter = 0;
22    for (int i = Row - 1; i <= Row + 1; i++)
23    {
24       for (int j = Column - 1; j <= Column + 1; j++)
25       {
26          if ((i != Row || j != Column) && IsCellAlive(Colony, i, j))
```

## Řešení verze 9 – zjednodušení výpočtu stavu buňky v nové generaci (pokrač.)

```
27          {
28             Counter += 1;
29          }
30       }
31    }
32    return Counter;
33 }
34
35 void main()
36 {
37    bool a[NumberOfRows][NumberOfColumns], b[NumberOfRows][NumberOfColumns];
38    int i, j, k, l, p, q;
39    for (i = 0; i < NumberOfRows; i++)
40       for (j = 0; j < NumberOfColumns; j++)
41          a[i][j] = false;
42    for (i = 0; i < NumberOfLivingCells; i++)
43    {
44       j = rand() % NumberOfRows;
45       k = rand() % NumberOfColumns;
46       a[j][k] = true;
47    }
48    cout << "Puvodni kolonie" << endl;
49    for (i = 0; i < NumberOfRows; i++)
50    {
51       for (j = 0; j < NumberOfColumns; j++)
52       {
53          if (a[i][j])
```



## Řešení verze 9 – zjednodušení výpočtu stavu buňky v nové generaci (pokrač.)

```
54             cout << "X";
55          else
56             cout << ".";
57       }
58       cout << endl;
59    }
60    cout << endl;
61    for (i = 1; i <= NumberOfGenerations; i++)
62    {
63       for (j = 0; j < NumberOfRows; j++)
64          for (k = 0; k < NumberOfColumns; k++)
65          {
66             int l = NumberOfLivingNeighbors(a, j, k);
67             b[j][k] = a[j][k] && l == 2 || l == 3;
68          }
69       for (j = 0; j < NumberOfRows; j++)
70          for (k = 0; k < NumberOfColumns; k++)
71             a[j][k] = b[j][k];
72       cout << "Generace #" << i << endl;
73       for (j = 0; j < NumberOfRows; j++)
74       {
75          for (k = 0; k < NumberOfColumns; k++)
76          {
77             if (a[j][k])
78                cout << "X";
79             else
80                cout << ".";
```



## Řešení verze 9 – zjednodušení výpočtu stavu buňky v nové generaci (pokrač.)

```
81          }
82          cout << endl;
83       }
84       cout << endl;
85    }
86 }
```



## Identifikace významu částí kódu

Pokud se podíváme na současný stav zdrojového kódu, tak zjistíme:

- podařilo se nám odstranit množství duplicitních testů (testování Moorova okolí), ale
- dosud je v cyklu iterujícím přes generace příliš mnoho vnořených cyklů.

Naší snahou nyní bude

1. identifikovat části kódu, které lze významově ohraničit, části které lze pojmenovat,
2. tyto části kódu přesunout do samostatných funkcí a
3. nahradit identifikované části kódu voláním nových funkcí.

V kódu lze identifikovat tyto části:

- řádky 63 až 68 – výpočet nové kolonie buněk,
- řádky 69 až 71 – přepsání současné kolonie buněk novou kolonií,
- řádky 73 až 84 – tisk současné kolonie buněk,

## Identifikace významu částí kódu (pokrač.)

- řádky 49 až 60 – tisk současné kolonie buněk, identická kopie řádků 73 až 84 (!!!) a
- řádky 39 až 47 – inicializace počáteční kolonie buněk.

Přesunutím těchto částí do funkcí získáme:

1. vyšší modularitu programu – funkce jako menší samostatné části lze samostatně implementovat, ladit, testovat a znovupoužívat
2. vyšší čitelnost celého kódu i funkce `main` samotné.

V dalších fází zlepšování kódu budeme pracovat s těmito identifikovanými částmi.



## Výpočet nové generace

Tato část řeší výpočet nové generace – vyjmeme ji a vytvoříme z ní funkci.

```
1 for (j = 0; j < NumberOfRows; j++)
2    for (k = 0; k < NumberOfColumns; k++)
3    {
4       int l = NumberOfLivingNeighbors(a, j, k);
5       b[j][k] = a[j][k] && l == 2 || l == 3;
6    }
```

```
1 void CreateNewColony(const bool CurrentColony[NumberOfRows][
      NumberOfColumns], bool NewColony[NumberOfRows][NumberOfColumns])
2 {
3    for (int i = 0; i < NumberOfRows; i++)
4    {
5       for (int j = 0; j < NumberOfColumns; j++)
6       {
7          int l = NumberOfLivingNeighbors(CurrentColony, i, j);
8          NewColony[i][j] = CurrentColony[i][j] && l == 2 || l == 3;
9       }
```



## Výpočet nové generace (pokrač.)

```
10    }
11 }
```



## Řešení verze 10 – výpočet nové generace buněk jako funkce

```
1  #include <iostream>
2
3  using namespace std;
4
5  const int NumberOfRows = 10;
6  const int NumberOfColumns = 10;
7  const int NumberOfGenerations = 10;
8  const int NumberOfLivingCells = 50;
9
10 bool IsCellAlive(const bool Colony[NumberOfRows][NumberOfColumns], const int Row, const int Column)
11 {
12    if (0 <= Row && Row < NumberOfRows && 0 <= Column && Column < NumberOfColumns)
13    {
14       return Colony[Row][Column];
15    }
16    return false;
17 }
18
19 int NumberOfLivingNeighbors(const bool Colony[NumberOfRows][NumberOfColumns], const int Row, const int
       Column)
20 {
21    int Counter = 0;
22    for (int i = Row - 1; i <= Row + 1; i++)
23    {
24       for (int j = Column - 1; j <= Column + 1; j++)
25       {
26          if ((i != Row || j != Column) && IsCellAlive(Colony, i, j))
27          {
```

## Řešení verze 10 – výpočet nové generace buněk jako funkce (pokrač.)

```
28            Counter += 1;
29         }
30      }
31   }
32   return Counter;
33 }
34
35 void CreateNewColony(const bool CurrentColony[NumberOfRows][NumberOfColumns], bool NewColony[NumberOfRows
       ][NumberOfColumns])
36 {
37   for (int i = 0; i < NumberOfRows; i++)
38   {
39      for (int j = 0; j < NumberOfColumns; j++)
40      {
41         int l = NumberOfLivingNeighbors(CurrentColony, i, j);
42         NewColony[i][j] = CurrentColony[i][j] && l == 2 || l == 3;
43      }
44   }
45 }
46
47 void main()
48 {
49   bool a[NumberOfRows][NumberOfColumns], b[NumberOfRows][NumberOfColumns];
50   int i, j, k, l, p, q;
51   for (i = 0; i < NumberOfRows; i++)
52      for (j = 0; j < NumberOfColumns; j++)
53         a[i][j] = false;
```



## Řešení verze 10 – výpočet nové generace buněk jako funkce (pokrač.)

```
54   for (i = 0; i < NumberOfLivingCells; i++)
55   {
56      j = rand() % NumberOfRows;
57      k = rand() % NumberOfColumns;
58      a[j][k] = true;
59   }
60   cout << "Puvodni kolonie" << endl;
61   for (i = 0; i < NumberOfRows; i++)
62   {
63      for (j = 0; j < NumberOfColumns; j++)
64      {
65         if (a[i][j])
66            cout << "X";
67         else
68            cout << ".";
69      }
70      cout << endl;
71   }
72   cout << endl;
73   for (i = 1; i <= NumberOfGenerations; i++)
74   {
75      CreateNewColony(a, b);
76      for (j = 0; j < NumberOfRows; j++)
77         for (k = 0; k < NumberOfColumns; k++)
78            a[j][k] = b[j][k];
79      cout << "Generace #" << i << endl;
80      for (j = 0; j < NumberOfRows; j++)
```



## Řešení verze 10 – výpočet nové generace buněk jako funkce (pokrač.)

```
81      {
82         for (k = 0; k < NumberOfColumns; k++)
83         {
84            if (a[j][k])
85               cout << "X";
86            else
87               cout << ".";
88         }
89         cout << endl;
90      }
91      cout << endl;
92   }
93 }
```



## Výstup kolonie na standardní výstup

Fragment kódu

```
for (i = 0; i < NumberOfRows; i++)
{
   for (j = 0; j < NumberOfColumns; j++)
   {
      if (a[i][j])
         cout << "X";
      else
         cout << ".";
   }
   cout << endl;
}
cout << endl;
```

nahradíme funkcí

## Výstup kolonie na standardní výstup (pokrač.)

```
1  void PrintColony(const bool Colony[NumberOfRows][NumberOfColumns])
2  {
3     for (int i = 0; i < NumberOfRows; i++)
4     {
5        for (int j = 0; j < NumberOfColumns; j++)
6        {
7           if (Colony[i][j])
8              cout << "X";
9           else
10             cout << ".";
11       }
12       cout << endl;
13    }
14    cout << endl;
15 }
```



## Řešení verze 11 – výstup kolonie na standardní výstup jako funkce

```
1  #include <iostream>
2
3  using namespace std;
4
5  const int NumberOfRows = 10;
6  const int NumberOfColumns = 10;
7  const int NumberOfGenerations = 10;
8  const int NumberOfLivingCells = 50;
9
10 bool IsCellAlive(const bool Colony[NumberOfRows][NumberOfColumns], const int Row, const int Column)
11 {
12    if (0 <= Row && Row < NumberOfRows && 0 <= Column && Column < NumberOfColumns)
13    {
14       return Colony[Row][Column];
15    }
16    return false;
17 }
18
19 int NumberOfLivingNeighbors(const bool Colony[NumberOfRows][NumberOfColumns], const int Row, const int
       Column)
20 {
21    int Counter = 0;
22    for (int i = Row - 1; i <= Row + 1; i++)
23    {
24       for (int j = Column - 1; j <= Column + 1; j++)
25       {
26          if ((i != Row || j != Column) && IsCellAlive(Colony, i, j))
27          {
```



## Řešení verze 11 – výstup kolonie na standardní výstup jako funkce (pokrač.)

```
28             Counter += 1;
29          }
30       }
31    }
32    return Counter;
33 }
34
35 void CreateNewColony(const bool CurrentColony[NumberOfRows][NumberOfColumns], bool NewColony[NumberOfRows
       ][NumberOfColumns])
36 {
37    for (int i = 0; i < NumberOfRows; i++)
38    {
39       for (int j = 0; j < NumberOfColumns; j++)
40       {
41          int l = NumberOfLivingNeighbors(CurrentColony, i, j);
42          NewColony[i][j] = CurrentColony[i][j] && l == 2 || l == 3;
43       }
44    }
45 }
46
47 void PrintColony(const bool Colony[NumberOfRows][NumberOfColumns])
48 {
49    for (int i = 0; i < NumberOfRows; i++)
50    {
51       for (int j = 0; j < NumberOfColumns; j++)
52       {
53          if (Colony[i][j])
```



## Řešení verze 11 – výstup kolonie na standardní výstup jako funkce (pokrač.)

```
54             cout << "X";
55          else
56             cout << ".";
57       }
58       cout << endl;
59    }
60    cout << endl;
61 }
62
63 void main()
64 {
65    bool a[NumberOfRows][NumberOfColumns], b[NumberOfRows][NumberOfColumns];
66    int i, j, k, l, p, q;
67    for (i = 0; i < NumberOfRows; i++)
68       for (j = 0; j < NumberOfColumns; j++)
69          a[i][j] = false;
70    for (i = 0; i < NumberOfLivingCells; i++)
71    {
72       j = rand() % NumberOfRows;
73       k = rand() % NumberOfColumns;
74       a[j][k] = true;
75    }
76    cout << "Puvodni kolonie" << endl;
77    PrintColony(a);
78    cout << endl;
79    for (i = 1; i <= NumberOfGenerations; i++)
80    {
```

## Řešení verze 11 – výstup kolonie na standardní výstup jako funkce (pokrač.)

```
81       CreateNewColony(a, b);
82       for (j = 0; j < NumberOfRows; j++)
83          for (k = 0; k < NumberOfColumns; k++)
84             a[j][k] = b[j][k];
85       cout << "Generace #" << i << endl;
86       PrintColony(a);
87    }
88 }
```



## Přepsání současné kolonie buněk novou kolonií

Fragment kódu

```
1 for (j = 0; j < NumberOfRows; j++)
2    for (k = 0; k < NumberOfColumns; k++)
3       a[j][k] = b[j][k];
```

nahradíme funkcí

```
1 void CopyColony(const bool SourceColony[NumberOfRows][NumberOfColumns],
        bool DestinationColony[NumberOfRows][NumberOfColumns])
2 {
3    for (int i = 0; i < NumberOfRows; i++)
4    {
5       for (int j = 0; j < NumberOfColumns; j++)
6       {
7          DestinationColony[i][j] = SourceColony[i][j];
8       }
9    }
10 }
```



## Řešení verze 12 – přepsání současné kolonie buněk novou kolonií jako funkce

```
1  #include <iostream>
2
3  using namespace std;
4
5  const int NumberOfRows = 10;
6  const int NumberOfColumns = 10;
7  const int NumberOfGenerations = 10;
8  const int NumberOfLivingCells = 50;
9
10 bool IsCellAlive(const bool Colony[NumberOfRows][NumberOfColumns], const int Row, const int Column)
11 {
12    if (0 <= Row && Row < NumberOfRows && 0 <= Column && Column < NumberOfColumns)
13    {
14       return Colony[Row][Column];
15    }
16    return false;
17 }
18
19 int NumberOfLivingNeighbors(const bool Colony[NumberOfRows][NumberOfColumns], const int Row, const int
        Column)
20 {
21    int Counter = 0;
22    for (int i = Row - 1; i <= Row + 1; i++)
23    {
24       for (int j = Column - 1; j <= Column + 1; j++)
25       {
26          if ((i != Row || j != Column) && IsCellAlive(Colony, i, j))
```



## Řešení verze 12 – přepsání současné kolonie buněk novou kolonií jako funkce (pokrač.)

```
27          {
28             Counter += 1;
29          }
30       }
31    }
32    return Counter;
33 }
34
35 void CreateNewColony(const bool CurrentColony[NumberOfRows][NumberOfColumns], bool NewColony[NumberOfRows
        ][NumberOfColumns])
36 {
37    for (int i = 0; i < NumberOfRows; i++)
38    {
39       for (int j = 0; j < NumberOfColumns; j++)
40       {
41          int l = NumberOfLivingNeighbors(CurrentColony, i, j);
42          NewColony[i][j] = CurrentColony[i][j] && l == 2 || l == 3;
43       }
44    }
45 }
46
47 void PrintColony(const bool Colony[NumberOfRows][NumberOfColumns])
48 {
49    for (int i = 0; i < NumberOfRows; i++)
50    {
51       for (int j = 0; j < NumberOfColumns; j++)
52       {
```

## Řešení verze 12 – přepsání současné kolonie buněk novou kolonií jako funkce (pokrač.)

```
53          if (Colony[i][j])
54             cout << "X";
55          else
56             cout << ".";
57       }
58       cout << endl;
59    }
60    cout << endl;
61 }
62
63 void CopyColony(const bool SourceColony[NumberOfRows][NumberOfColumns], bool DestinationColony[
         NumberOfRows][NumberOfColumns])
64 {
65    for (int i = 0; i < NumberOfRows; i++)
66    {
67       for (int j = 0; j < NumberOfColumns; j++)
68       {
69          DestinationColony[i][j] = SourceColony[i][j];
70       }
71    }
72 }
73
74 void main()
75 {
76    bool a[NumberOfRows][NumberOfColumns], b[NumberOfRows][NumberOfColumns];
77    int i, j, k, l, p, q;
78    for (i = 0; i < NumberOfRows; i++)
```



## Řešení verze 12 – přepsání současné kolonie buněk novou kolonií jako funkce (pokrač.)

```
79       for (j = 0; j < NumberOfColumns; j++)
80          a[i][j] = false;
81    for (i = 0; i < NumberOfLivingCells; i++)
82    {
83       j = rand() % NumberOfRows;
84       k = rand() % NumberOfColumns;
85       a[j][k] = true;
86    }
87    cout << "Puvodni kolonie" << endl;
88    PrintColony(a);
89    cout << endl;
90    for (i = 1; i <= NumberOfGenerations; i++)
91    {
92       CreateNewColony(a, b);
93       CopyColony(b, a);
94       cout << "Generace #" << i << endl;
95       PrintColony(a);
96    }
97 }
```



## Inicializace počáteční kolonie buněk

Fragment kódu

```
for (i = 0; i < NumberOfRows; i++)
   for (j = 0; j < NumberOfColumns; j++)
      a[i][j] = false;
for (i = 0; i < NumberOfLivingCells; i++)
{
   j = rand() % NumberOfRows;
   k = rand() % NumberOfColumns;
   a[j][k] = true;
}
```

nahradíme funkcí



## Inicializace počáteční kolonie buněk (pokrač.)

```
void InitColony(bool Colony[NumberOfRows][NumberOfColumns], const int
     NumberOfLivingCells)
{
   for (int i = 0; i < NumberOfRows; i++)
   {
      for (int j = 0; j < NumberOfColumns; j++)
      {
         Colony[i][j] = false;
      }
   }
   for (int i = 0; i < NumberOfLivingCells; i++)
   {
      int r = rand() % NumberOfRows;
      int c = rand() % NumberOfColumns;
      Colony[r][c] = true;
   }
}
```

# Řešení verze 13 – inicializace počáteční kolonie buněk jako funkce

```
 1 #include <iostream>
 2
 3 using namespace std;
 4
 5 const int NumberOfRows = 10;
 6 const int NumberOfColumns = 10;
 7 const int NumberOfGenerations = 10;
 8 const int NumberOfLivingCells = 50;
 9
10 bool IsCellAlive(const bool Colony[NumberOfRows][NumberOfColumns], const int Row, const int Column)
11 {
12    if (0 <= Row && Row < NumberOfRows && 0 <= Column && Column < NumberOfColumns)
13    {
14       return Colony[Row][Column];
15    }
16    return false;
17 }
18
19 int NumberOfLivingNeighbors(const bool Colony[NumberOfRows][NumberOfColumns], const int Row, const int
       Column)
20 {
21    int Counter = 0;
22    for (int i = Row - 1; i <= Row + 1; i++)
23    {
24       for (int j = Column - 1; j <= Column + 1; j++)
25       {
26          if ((i != Row || j != Column) && IsCellAlive(Colony, i, j))
27          {
```



# Řešení verze 13 – inicializace počáteční kolonie buněk jako funkce (pokrač.)

```
28             Counter += 1;
29          }
30       }
31    }
32    return Counter;
33 }
34
35 void CreateNewColony(const bool CurrentColony[NumberOfRows][NumberOfColumns], bool NewColony[NumberOfRows
       ][NumberOfColumns])
36 {
37    for (int i = 0; i < NumberOfRows; i++)
38    {
39       for (int j = 0; j < NumberOfColumns; j++)
40       {
41          int l = NumberOfLivingNeighbors(CurrentColony, i, j);
42          NewColony[i][j] = CurrentColony[i][j] && l == 2 || l == 3;
43       }
44    }
45 }
46
47 void PrintColony(const bool Colony[NumberOfRows][NumberOfColumns])
48 {
49    for (int i = 0; i < NumberOfRows; i++)
50    {
51       for (int j = 0; j < NumberOfColumns; j++)
52       {
53          if (Colony[i][j])
```



# Řešení verze 13 – inicializace počáteční kolonie buněk jako funkce (pokrač.)

```
54             cout << "X";
55          else
56             cout << ".";
57       }
58       cout << endl;
59    }
60    cout << endl;
61 }
62
63 void CopyColony(const bool SourceColony[NumberOfRows][NumberOfColumns], bool DestinationColony[
       NumberOfRows][NumberOfColumns])
64 {
65    for (int i = 0; i < NumberOfRows; i++)
66    {
67       for (int j = 0; j < NumberOfColumns; j++)
68       {
69          DestinationColony[i][j] = SourceColony[i][j];
70       }
71    }
72 }
73
74 void InitColony(bool Colony[NumberOfRows][NumberOfColumns], const int NumberOfLivingCells)
75 {
76    for (int i = 0; i < NumberOfRows; i++)
77    {
78       for (int j = 0; j < NumberOfColumns; j++)
79       {
```



# Řešení verze 13 – inicializace počáteční kolonie buněk jako funkce (pokrač.)

```
80          Colony[i][j] = false;
81       }
82    }
83    for (int i = 0; i < NumberOfLivingCells; i++)
84    {
85       int r = rand() % NumberOfRows;
86       int c = rand() % NumberOfColumns;
87       Colony[r][c] = true;
88    }
89 }
90
91 void main()
92 {
93    bool a[NumberOfRows][NumberOfColumns], b[NumberOfRows][NumberOfColumns];
94    int i, j, k, l, p, q;
95    InitColony(a, NumberOfLivingCells);
96    cout << "Puvodni kolonie" << endl;
97    PrintColony(a);
98    cout << endl;
99    for (i = 1; i <= NumberOfGenerations; i++)
100   {
101      CreateNewColony(a, b);
102      CopyColony(b, a);
103      cout << "Generace #" << i << endl;
104      PrintColony(a);
105   }
106 }
```

## Řešení verze 13 – inicializace počáteční kolonie buněk jako funkce (pokrač.)



## Je inicializace počáteční kolonie buněk korektní?

Podívejme se na náhodné generování živých buněk v počáteční kolonii

```
1 for (int i = 0; i < NumberOfLivingCells; i++)
2 {
3    int r = rand() % NumberOfRows;
4    int c = rand() % NumberOfColumns;
5    Colony[r][c] = true;
6 }
```

**Problém**

Při inicializaci kolonie může dojít k situaci, kdy se vygeneruje pozice již živé buňky! Buňka může tedy obživnout několikrát, což je chyba a výsledný počet živých buněk tak může být menší než je požadováno.



## Je inicializace počáteční kolonie buněk korektní? (pokrač.)

**Řešení**

- Počítadlo živých buněk `i` můžeme tedy inkrementovat jen v okamžiku, kdy jsme vygenerovali **novou** živou buňku.
- Cyklus `for` ale proměnnou `i` inkrementuje po každém průchodu cyklem.
- Při neúspěšném generování nové buňky musíme proměnnou `i` zmenšit o 1 abychom korigovali automatickou inkrementaci po průchodu cyklem.

```
1 for (int i = 0; i < NumberOfLivingCells; i++)
2 {
3    int r = rand() % NumberOfRows;
4    int c = rand() % NumberOfColumns;
5    if (!Colony[r][c])
6    {
7       Colony[r][c] = true;
8    }
```



## Je inicializace počáteční kolonie buněk korektní? (pokrač.)

```
9    else
10   {
11      i -= 1;
12   }
13 }
```

# Řešení verze 14 – korekce počitadla živých buněk

```
1  #include <iostream>
2
3  using namespace std;
4
5  const int NumberOfRows = 10;
6  const int NumberOfColumns = 10;
7  const int NumberOfGenerations = 10;
8  const int NumberOfLivingCells = 50;
9
10 bool IsCellAlive(const bool Colony[NumberOfRows][NumberOfColumns], const int Row, const int Column)
11 {
12    if (0 <= Row && Row < NumberOfRows && 0 <= Column && Column < NumberOfColumns)
13    {
14       return Colony[Row][Column];
15    }
16    return false;
17 }
18
19 int NumberOfLivingNeighbors(const bool Colony[NumberOfRows][NumberOfColumns], const int Row, const int
       Column)
20 {
21    int Counter = 0;
22    for (int i = Row - 1; i <= Row + 1; i++)
23    {
24       for (int j = Column - 1; j <= Column + 1; j++)
25       {
26          if ((i != Row || j != Column) && IsCellAlive(Colony, i, j))
27          {
28             Counter += 1;
29          }
```



# Řešení verze 14 – korekce počitadla živých buněk (pokrač.)

```
30       }
31    }
32    return Counter;
33 }
34
35 void CreateNewColony(const bool CurrentColony[NumberOfRows][NumberOfColumns], bool NewColony[NumberOfRows
       ][NumberOfColumns])
36 {
37    for (int i = 0; i < NumberOfRows; i++)
38    {
39       for (int j = 0; j < NumberOfColumns; j++)
40       {
41          int l = NumberOfLivingNeighbors(CurrentColony, i, j);
42          NewColony[i][j] = CurrentColony[i][j] && l == 2 || l == 3;
43       }
44    }
45 }
46
47 void PrintColony(const bool Colony[NumberOfRows][NumberOfColumns])
48 {
49    for (int i = 0; i < NumberOfRows; i++)
50    {
51       for (int j = 0; j < NumberOfColumns; j++)
52       {
53          if (Colony[i][j])
54             cout << "X";
55          else
56             cout << ".";
57       }
58       cout << endl;
```



# Řešení verze 14 – korekce počitadla živých buněk (pokrač.)

```
59    }
60    cout << endl;
61 }
62
63 void CopyColony(const bool SourceColony[NumberOfRows][NumberOfColumns], bool DestinationColony[
       NumberOfRows][NumberOfColumns])
64 {
65    for (int i = 0; i < NumberOfRows; i++)
66    {
67       for (int j = 0; j < NumberOfColumns; j++)
68       {
69          DestinationColony[i][j] = SourceColony[i][j];
70       }
71    }
72 }
73
74 void InitColony(bool Colony[NumberOfRows][NumberOfColumns], const int NumberOfLivingCells)
75 {
76    for (int i = 0; i < NumberOfRows; i++)
77    {
78       for (int j = 0; j < NumberOfColumns; j++)
79       {
80          Colony[i][j] = false;
81       }
82    }
83    for (int i = 0; i < NumberOfLivingCells; i++)
84    {
85       int r = rand() % NumberOfRows;
86       int c = rand() % NumberOfColumns;
87       if (!Colony[r][c])
```



# Řešení verze 14 – korekce počitadla živých buněk (pokrač.)

```
88       {
89          Colony[r][c] = true;
90       }
91       else
92       {
93          i -= 1;
94       }
95    }
96 }
97
98 void main()
99 {
100   bool a[NumberOfRows][NumberOfColumns], b[NumberOfRows][NumberOfColumns];
101   int i, j, k, l, p, q;
102   InitColony(a, NumberOfLivingCells);
103   cout << "Puvodni kolonie" << endl;
104   PrintColony(a);
105   cout << endl;
106   for (i = 1; i <= NumberOfGenerations; i++)
107   {
108      CreateNewColony(a, b);
109      CopyColony(b, a);
110      cout << "Generace #" << i << endl;
111      PrintColony(a);
112   }
113 }
```

## Inicializace počáteční kolonie buněk ještě jednou

- Můžeme být s předchozím řešením spokojeni?
- Určitě ne!
- Současné řešení je ukázkou, jak nikdy s cyklem `for` nepracovat!
- Cyklus `for` existuje proto, aby veškerá manipulace s iterační proměnnou byla soustředěna přehledně na jednom řádku kódu.
- Pokud toto neplatí musíme
  - použít cyklus `while`, kde se předpokládá manipulace s iterační proměnnou na více řádcích nebo
  - cyklus zcela přepsat.

Zvolíme druhou možnost:

- zachováme cyklus `for` a
- tělo cyklu přepíšeme tak, aby se bez vygenerování nové živé buňky vůbec nedalo v cyklu `for` pokračovat.



## Inicializace počáteční kolonie buněk ještě jednou (pokrač.)

```
1 for (int i = 0; i < NumberOfLivingCells; i++)
2 {
3    int r, c;
4    do
5    {
6       r = rand() % NumberOfRows;
7       c = rand() % NumberOfColumns;
8    } while (Colony[r][c]);
9    Colony[r][c] = true;
10 }
```



## Řešení verze 15 – korekce počitadla živých buněk II

```
1  #include <iostream>
2
3  using namespace std;
4
5  const int NumberOfRows = 10;
6  const int NumberOfColumns = 10;
7  const int NumberOfGenerations = 10;
8  const int NumberOfLivingCells = 50;
9
10 bool IsCellAlive(const bool Colony[NumberOfRows][NumberOfColumns], const int Row, const int Column)
11 {
12    if (0 <= Row && Row < NumberOfRows && 0 <= Column && Column < NumberOfColumns)
13    {
14       return Colony[Row][Column];
15    }
16    return false;
17 }
18
19 int NumberOfLivingNeighbors(const bool Colony[NumberOfRows][NumberOfColumns], const int Row, const int
       Column)
20 {
21    int Counter = 0;
22    for (int i = Row - 1; i <= Row + 1; i++)
23    {
24       for (int j = Column - 1; j <= Column + 1; j++)
25       {
26          if ((i != Row || j != Column) && IsCellAlive(Colony, i, j))
27          {
28             Counter += 1;
29          }
```



## Řešení verze 15 – korekce počitadla živých buněk II (pokrač.)

```
30       }
31    }
32    return Counter;
33 }
34
35 void CreateNewColony(const bool CurrentColony[NumberOfRows][NumberOfColumns], bool NewColony[NumberOfRows
       ][NumberOfColumns])
36 {
37    for (int i = 0; i < NumberOfRows; i++)
38    {
39       for (int j = 0; j < NumberOfColumns; j++)
40       {
41          int l = NumberOfLivingNeighbors(CurrentColony, i, j);
42          NewColony[i][j] = CurrentColony[i][j] && l == 2 || l == 3;
43       }
44    }
45 }
46
47 void PrintColony(const bool Colony[NumberOfRows][NumberOfColumns])
48 {
49    for (int i = 0; i < NumberOfRows; i++)
50    {
51       for (int j = 0; j < NumberOfColumns; j++)
52       {
53          if (Colony[i][j])
54             cout << "X";
55          else
```

## Řešení verze 15 – korekce počitadla živých buněk II (pokrač.)

```
56            cout << ".";
57         }
58         cout << endl;
59      }
60      cout << endl;
61   }
62
63   void CopyColony(const bool SourceColony[NumberOfRows][NumberOfColumns], bool DestinationColony[
         NumberOfRows][NumberOfColumns])
64   {
65      for (int i = 0; i < NumberOfRows; i++)
66      {
67         for (int j = 0; j < NumberOfColumns; j++)
68         {
69            DestinationColony[i][j] = SourceColony[i][j];
70         }
71      }
72   }
73
74   void InitColony(bool Colony[NumberOfRows][NumberOfColumns], const int NumberOfLivingCells)
75   {
76      for (int i = 0; i < NumberOfRows; i++)
77      {
78         for (int j = 0; j < NumberOfColumns; j++)
79         {
80            Colony[i][j] = false;
81         }
```



## Řešení verze 15 – korekce počitadla živých buněk II (pokrač.)

```
82      }
83      for (int i = 0; i < NumberOfLivingCells; i++)
84      {
85         int r, c;
86         do
87         {
88            r = rand() % NumberOfRows;
89            c = rand() % NumberOfColumns;
90         } while (Colony[r][c]);
91         Colony[r][c] = true;
92      }
93   }
94
95   void main()
96   {
97      bool a[NumberOfRows][NumberOfColumns], b[NumberOfRows][NumberOfColumns];
98      int i, j, k, l, p, q;
99      InitColony(a, NumberOfLivingCells);
100     cout << "Puvodni kolonie" << endl;
101     PrintColony(a);
102     cout << endl;
103     for (i = 1; i <= NumberOfGenerations; i++)
104     {
105        CreateNewColony(a, b);
106        CopyColony(b, a);
107        cout << "Generace #" << i << endl;
108        PrintColony(a);
```



## Řešení verze 15 – korekce počitadla živých buněk II (pokrač.)

```
109     }
110  }
```



## Závěrečný úklid

Program jsme kompletně přepsali. Co závěrem?

- odstranění nepoužívaných proměnných,
- výstižnější jména současné a nové kolonie buněk.

# Řešení verze 15 – závěrečný úklid

```
 1 #include <iostream>
 2
 3 using namespace std;
 4
 5 const int NumberOfRows = 10;
 6 const int NumberOfColumns = 10;
 7 const int NumberOfGenerations = 10;
 8 const int NumberOfLivingCells = 50;
 9
10 bool IsCellAlive(const bool Colony[NumberOfRows][NumberOfColumns], const int Row, const int Column)
11 {
12    if (0 <= Row && Row < NumberOfRows && 0 <= Column && Column < NumberOfColumns)
13    {
14       return Colony[Row][Column];
15    }
16    return false;
17 }
18
19 int NumberOfLivingNeighbors(const bool Colony[NumberOfRows][NumberOfColumns], const int Row, const int
       Column)
20 {
21    int Counter = 0;
22    for (int i = Row - 1; i <= Row + 1; i++)
23    {
24       for (int j = Column - 1; j <= Column + 1; j++)
25       {
26          if ((i != Row || j != Column) && IsCellAlive(Colony, i, j))
27          {
28             Counter += 1;
29          }
```



# Řešení verze 15 – závěrečný úklid (pokrač.)

```
30       }
31    }
32    return Counter;
33 }
34
35 void CreateNewColony(const bool CurrentColony[NumberOfRows][NumberOfColumns], bool NewColony[NumberOfRows
       ][NumberOfColumns])
36 {
37    for (int i = 0; i < NumberOfRows; i++)
38    {
39       for (int j = 0; j < NumberOfColumns; j++)
40       {
41          int l = NumberOfLivingNeighbors(CurrentColony, i, j);
42          NewColony[i][j] = CurrentColony[i][j] && l == 2 || l == 3;
43       }
44    }
45 }
46
47 void PrintColony(const bool Colony[NumberOfRows][NumberOfColumns])
48 {
49    for (int i = 0; i < NumberOfRows; i++)
50    {
51       for (int j = 0; j < NumberOfColumns; j++)
52       {
53          if (Colony[i][j])
54             cout << "X";
55          else
56             cout << ".";
57       }
58       cout << endl;
```



# Řešení verze 15 – závěrečný úklid (pokrač.)

```
59    }
60    cout << endl;
61 }
62
63 void CopyColony(const bool SourceColony[NumberOfRows][NumberOfColumns], bool DestinationColony[
       NumberOfRows][NumberOfColumns])
64 {
65    for (int i = 0; i < NumberOfRows; i++)
66    {
67       for (int j = 0; j < NumberOfColumns; j++)
68       {
69          DestinationColony[i][j] = SourceColony[i][j];
70       }
71    }
72 }
73
74 void InitColony(bool Colony[NumberOfRows][NumberOfColumns], const int NumberOfLivingCells)
75 {
76    for (int i = 0; i < NumberOfRows; i++)
77    {
78       for (int j = 0; j < NumberOfColumns; j++)
79       {
80          Colony[i][j] = false;
81       }
82    }
83    for (int i = 0; i < NumberOfLivingCells; i++)
84    {
85       int r, c;
86       do
87       {
```



# Řešení verze 15 – závěrečný úklid (pokrač.)

```
88          r = rand() % NumberOfRows;
89          c = rand() % NumberOfColumns;
90       } while (Colony[r][c]);
91       Colony[r][c] = true;
92    }
93 }
94
95 void main()
96 {
97    bool CurrentColony[NumberOfRows][NumberOfColumns];
98    bool NewColony[NumberOfRows][NumberOfColumns];
99    InitColony(CurrentColony, NumberOfLivingCells);
100   cout << "Puvodni kolonie" << endl;
101   PrintColony(CurrentColony);
102   cout << endl;
103   for (int i = 1; i <= NumberOfGenerations; i++)
104   {
105      CreateNewColony(CurrentColony, NewColony);
106      CopyColony(NewColony, CurrentColony);
107      cout << "Generace #" << i << endl;
108      PrintColony(CurrentColony);
109   }
110 }
```

## Refaktorování

- Refaktorování je disciplinovaný proces provádění změn v softwarovém systému takovým způsobem, že nemají vliv na vnější chování kódu, ale vylepšují jeho vnitřní strukturu s minimálním rizikem vnášení chyb.
- Při refaktoringu provádíme malé až primitivní změny, ale celkový efekt je velký a to v podobě čistšího, průhlednějšího a čitelnějšího kódu, kód se také lépe udržuje a rozšiřuje.
- Zlepšuje se také celková kvalita kódu a architektura, snižuje se počet chyb a tím i zvyšuje rychlost vývoje programu.
- Refaktoring nám pomáhá pochopit a více si ujasnit kód, což je vhodné zejména, pokud upravujeme zdrojový kód po někom jiném.

**Kdy refaktorovat?**
Martin Fowler uvádí 22 takzvaných „pachů v kódu“, které jsou zvěstí toho, že bychom kód měli začít refaktorovat. Nejzákladnější a většinou i nejčastější pachy, které potkáte:



## Refaktorování (pokrač.)

Duplicitní kód – pokud najdeme v kódu na více místech stejnou strukturu

Dlouhá funkce – většinou se v takové funkci řeší více problémů najednou a měli bychom ji rozdělit menší funkce, které řeší právě jeden problém.

Komentáře – pokud máme tendenci komentovat uvnitř metody náš kód, je většinou lepší tento kód vyčlenit do samostatné funkce, a pomocí vhodného názvu této funkce nám odpadne nutnost komentáře

Složité struktury podmínek – vyčleněním do samostatné funkce se zlepší čitelnost kódu,

**Doporučená literatura**

- FOWLER, Martin. Refaktoring : Zlepšení existujícího kódu (původním názvem: Refactoring: Improving the Design of Existing Code). Překlad Vladimír Lahoda. Praha : Grada, 2003. ISBN 80-247-0299-1



## Refaktorování (pokrač.)

- Refaktorování. In: Wikipedia: the free encyclopedia [online]. San Francisco (CA): Wikimedia Foundation, 2001- [cit. 2017-10-15]. Dostupné z: `https://cs.wikipedia.org/wiki/Refaktorov%C3%A1n%C3%AD`



## Poznámka na závěr aneb proč „pach kódu“?

- Označení pach kódu vymyslel Martin Fowler v době, kdy se stal otcem a měl přebalovat mimino.
- Ideální stav mimina – čisté mimino s novou plenou. Pach, v tomto případě vůně, mimina je ideální.
- Mimino se počurá – pach je už nějaký jiný.
- Mimino se pokaká – opět jiný pach.
- Mimino má průjem – děsivý pach.
- Pach tedy není fatální, život ohrožující problém, ale signalizuje nám cosi podezřelého...

**Závěr** – ucítíme-li jiný než správný pach je nutné refaktorovat – vyměnit plenu.

# Děkuji za pozornost



# Rekurze

doc. Mgr. Jiří Dvorský, Ph.D.

Katedra informatiky
Fakulta elektrotechniky a informatiky
VŠB – TU Ostrava





## Rekurze

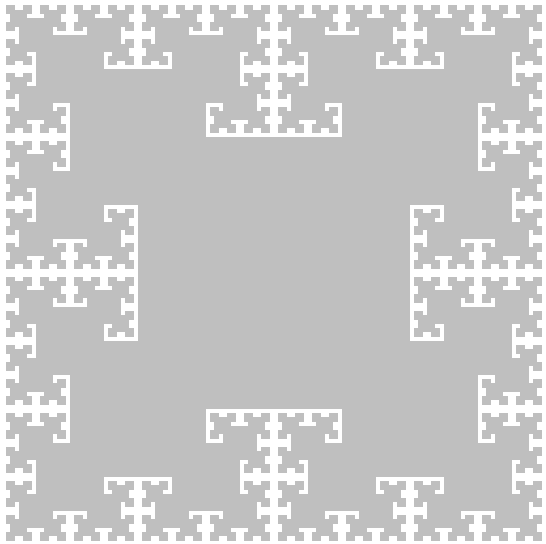
Co si představit pod pojmem rekurze?

1. „seber všechny hračky!“,
2. soběpodobnost, fraktály,
3. z hlediska algoritmů, je to algoritmus, který v určitém okamžiku výpočtu volá sám sebe,
4. rozklad na menší podproblémy, princip **rozděl a panuj**, (divide et impera),
5. dělení končí u atomického problému – řešení je známo například z definice,
6. rekurze je jistým způsobem ekvivalentní iteraci (cyklu).



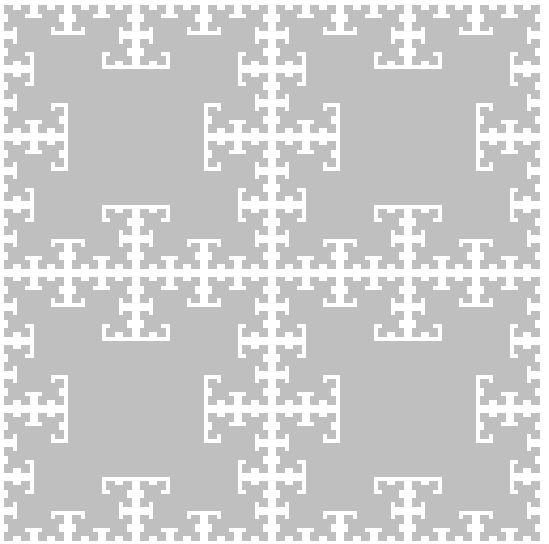
## Ukázka rekurze graficky

## Ukázka rekurze graficky



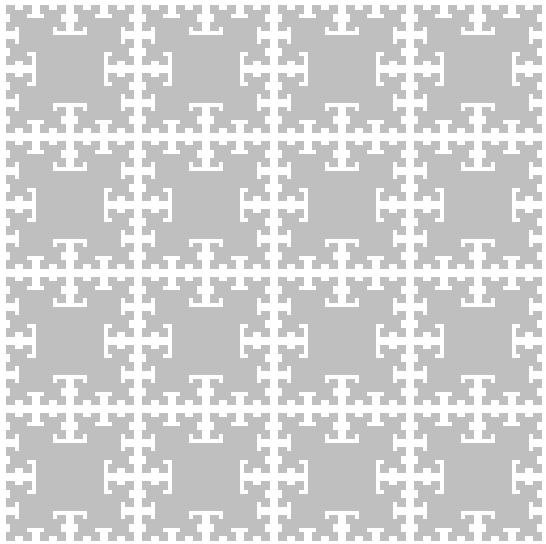


## Ukázka rekurze graficky



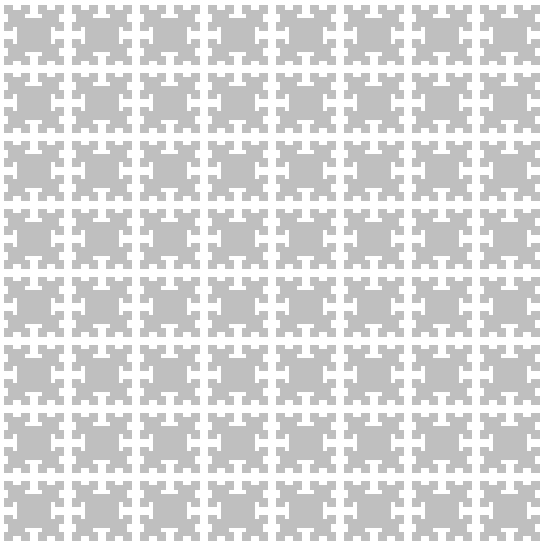


## Ukázka rekurze graficky

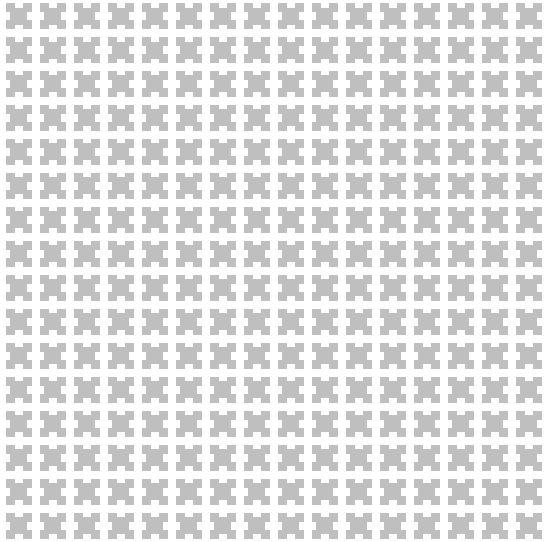




## Ukázka rekurze graficky

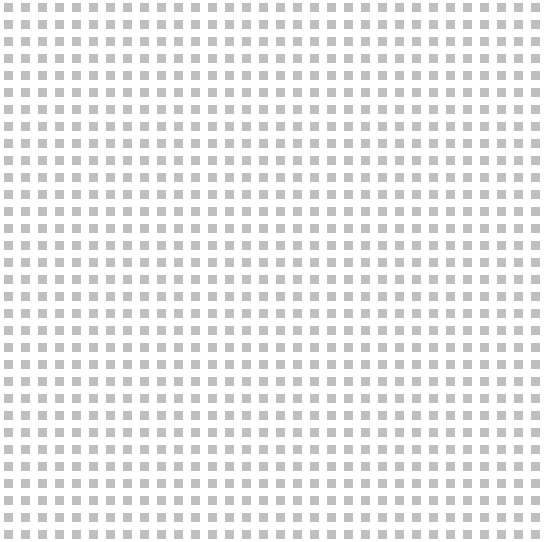


## Ukázka rekurze graficky



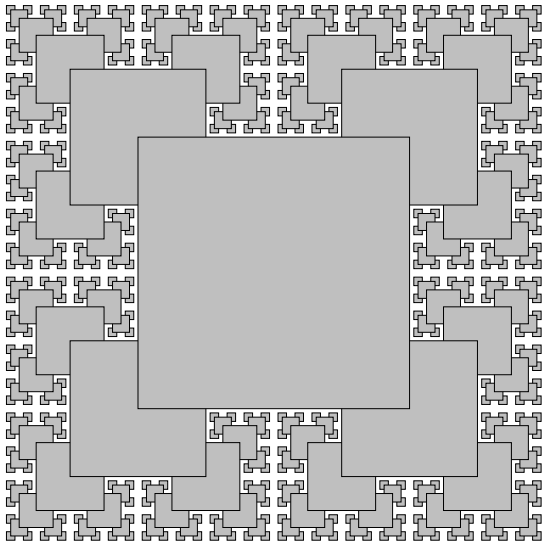


## Ukázka rekurze graficky





## Ukázka rekurze graficky





## Funkce vykreslující ukázkový fraktál

```
1 void Draw(const int Level, const double CenterX, const double CenterY,
      const double A)
2 {
3   if (Level > MaxLevel)
4     return;
5   double A2 = A / 2;
6   Draw(Level + 1, CenterX - A2, CenterY - A2, A2);
7   Draw(Level + 1, CenterX + A2, CenterY - A2, A2);
8   Draw(Level + 1, CenterX + A2, CenterY + A2, A2);
9   Draw(Level + 1, CenterX - A2, CenterY + A2, A2);
10  DrawSquare(CenterX - A2, CenterY - A2, A);
11 }
```

Ukázka volání

```
Draw(0, 0, 0, 50);
```

## Některé pojmy

- přímá rekurze – funkce volá sama sebe.
- nepřímá rekurze – funkce **A** volá funkci **B** a ta opět volá funkce **A**.
- ukončovací podmínka – jistá část řešení problému je atomická, řešení je známo např. z definice problému.



## Faktoriál

Princip rekurze si ukážeme na výpočtu faktoriálu čísla $n$. Funkce faktoriál je definována jako

$$n! = \begin{cases} 1 & pro\ n = 0 \\ n * (n-1)! & pro\ n \geq 1 \end{cases}$$

Pokud funkci faktoriál místo $n!$ označíme jako $f(n)$ můžeme faktoriál zapsat jako

$$f(n) = \begin{cases} 1 & pro\ n = 0 \\ n * f(n-1) & pro\ n \geq 1 \end{cases}$$

Což vede ke známému vyjádření

$$n! = n*(n-1)! = n*(n-1)(n-2)! = \cdots = n*(n-1)*(n-2)*\cdots*1*0!$$



## Výpočet hodnot funkce faktoriál

| Vstupní hodnota | Výpočet hodnoty | Hodnota faktoriálu |
|---:|---:|---:|
| 0 | podle definice | 1 |
| 1 | $1 * 1$ | 1 |
| 2 | $2 * 1$ | 2 |
| 3 | $3 * 2$ | 6 |
| 4 | $4 * 6$ | 24 |
| 5 | $5 * 24$ | 120 |
| 6 | $6 * 120$ | 720 |
| 7 | $7 * 720$ | 5 040 |
| 8 | $8 * 5\,040$ | 40 320 |
| 9 | $9 * 40\,320$ | 362 880 |
| 10 | $10 * 362\,880$ | 3 628 800 |



## Faktoriál – ukázka kódu

```
1 int factorial(const int n)
2 {
3   if(n == 0)
4     return 1;
5   return n * factorial(n - 1);
6 }
```

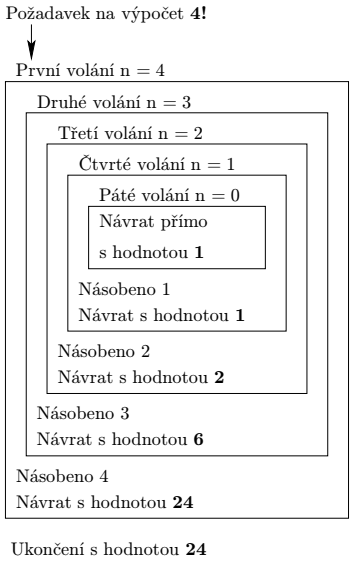
## Faktoriál – výpočet

Co se děje při rekurzívním výpočtu? Jak je zajištěno, aby se uchovaly všechny hodnoty proměnných v okamžiku rekurentního volání dílčí úlohy a při jejím ukončení, tj.návratu z poslední rekurentní funkce či procedury byly předány správné hodnoty?

- jednotlivé úlohy seřadíme do řady, aktuální úloha je na konci
- v okamžiku dokončení výpočtu, je výsledek předán volající metodě, ruší se lokální proměnné
- je to **zásobník**, LIFO
- každým rekurzivním voláním metody vzniká nová množina všech parametrů a lokálních proměnných
- mají sice stejné identifikátory jako při prvním volání, ale jejich hodnoty jsou jiné



## Faktoriál – schéma rekurze






## Fibonacciho posloupnost

aneb *Kolik potomků – párů králíků bude mít po roce jeden původní králičí pár?*

1. Na pole vypustíme pár králíků.
2. Králíci dospívají po jednom měsíci, březost trvá jeden měsíc. To znamená, že po dvou měsících samice vyprodukuje nový pár králíků.
3. Králíci neumírají a nejsou nemocní.
4. Každý pár králíků vždy vyprodukuje jeden nový pár samec/samice.

Úloha byla poprvé publikována roku 1202 Leonardem Pisano (Leonardo Fibonacci) v knize *Liber Abacci*



## Fibonacciho posloupnost

**Řešení**:

1. Po prvním měsíci bude na poli stále jeden pár králíků. Na poli je 1 pár.
2. Na konci druhého měsíce vznikne jeden nový pár králíků. Na poli jsou 2 páry.
3. Na konci třetího měsíce původní pár vyprodukuje nový pár, druhý pár dospívá. Na poli jsou 3 páry.
4. Na konci čtvrtého měsíce, původní pár vyprodukoval další pár, pár narozený před dvěma měsíci má svoje první potomstvo. Na poli je 5 párů.

**Na konci n-tého měsíce je počet párů králíků roven počtu nových párů králíků (což je počet párů v měsíci n-2) plus počet párů žijících v minulém měsíci.**

## Fibonacciho posloupnost

**Matematické vyjádření:**

$$F(n) = \begin{cases} 0 & \text{pro } n = 0 \\ 1 & \text{pro } n = 1 \\ F(n-1) + F(n-2) & \text{pro } n > 1 \end{cases}$$

$0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, \ldots$

$F_0 = 0,\ F_1 = 1,\ F_2 = 1,\ F_3 = 2,\ F_4 = 3,\ F_5 = 5,\ F_6 = 8,\ F_7 = 13,\ \ldots$

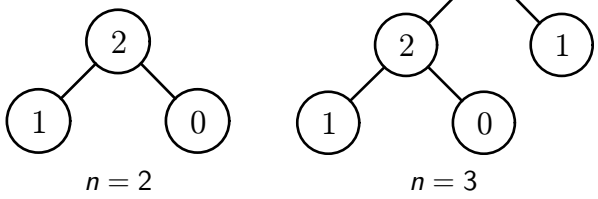


## Fibonacciho posloupnost – ukázka kódu

```
1 int F(const int n)
2 {
3   if (n == 0)
4     return 0;
5   if (n == 1)
6     return 1;
7   return F(n - 1) + F(n - 2);
8 }
```



## Fibonacciho posloupnost – ukázka výpočtu



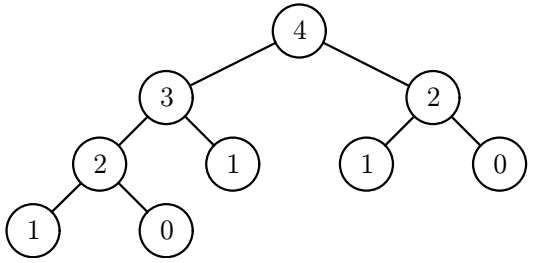

$n = 2$




$n = 3$

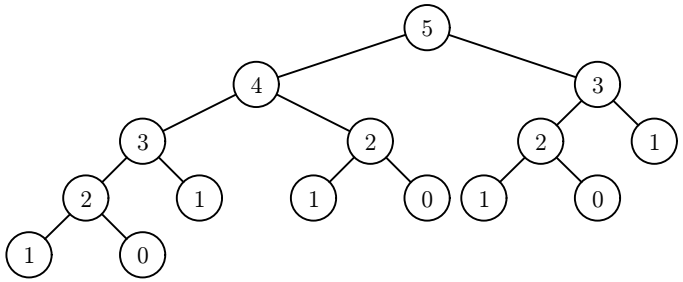


## Fibonacciho posloupnost – ukázka výpočtu




$n = 4$

## Fibonacciho posloupnost – ukázka výpočtu




$n = 5$



## Vyhledání maxima hodnot v poli iterativně

```
1 int maximumIter(int[] a, const int N)
2 {
3   int max = a[0];
4   for(int i = 0; i < N; i++)
5   {
6     if (a[i] > max)
7       max = a[i];
8   }
9   return max;
10 }
```



## Vyhledání maxima hodnot v poli rekurzivně

```
1 int maxRecursive(int[] a, const int left, const int right)
2 {
3   if (left == right)
4     return a[left];
5   int m = (left + right) / 2;
6   int lm = maxRecursive(a, left, m);
7   int rm = maxRecursive(a, m+1, right);
8   return lm > rm ? lm : rm;
9 }
```



## Rekurze – shrnutí

- užitečná technika jak psát programy,
- založena na rozdělení problému na menší podproblémy,
- z vyřešených dílčích částí skládá celek,
- spíše deklarujeme co chceme řešit, než jak to máme řešit,
- používána pro vyhledávání řešení.

## Děkuji za pozornost



# Využití rekurze pro řešení problémů

doc. Mgr. Jiří Dvorský, Ph.D.

Katedra informatiky
Fakulta elektrotechniky a informatiky
VŠB – TU Ostrava





## Backtracking

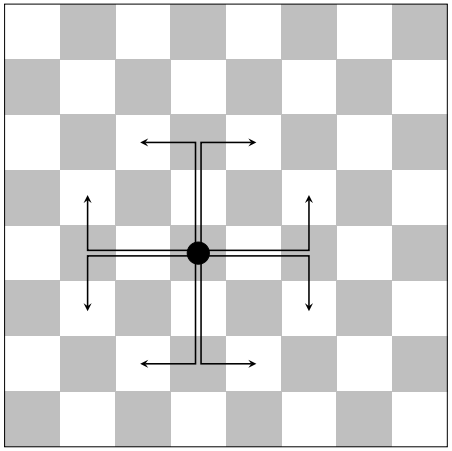
- zpětné vyhledávání, metoda pokusů a oprav, metoda zpětného sledování, metoda prohledávání do hloubky,
- způsob řešení algoritmických problémů
- založený na prohledávání stavového stromu problému
- možné použít pro řešení velkého množství problémů
- obecně exponenciální časová složitost
- jen pro data malé velikosti, popřípadě pro něj existuje dobrá heuristika.



## Jezdcova procházka

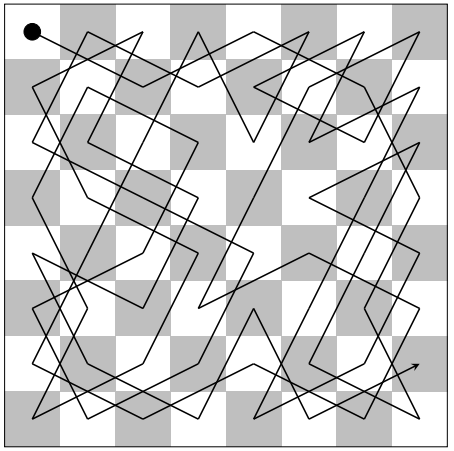
- šachový a matematický problém popsaný pomocí šachové figury jezdce a šachovnice,
- jezdec se pohybuje v souladu s šachovými pravidly po prázdné šachovnici,
- úkolem je, aby každé pole navštívil právě jednou,
- úloha již z 9. století,
- pro šachovnici $8 \times 8$ existuje 26 534 728 821 064 řešení, kdy kůň ohrožuje své výchozí postavení,
- řešení uzavřená vs otevřená.

## Možné tahy šachovým koněm





## Výsledné řešení





## Animace výsledného řešení

Animace jsou v poznámkách k prezentaci vynechány.



## Zdrojové kódy řešení, soubor KnightsTour.h

```
1 #pragma once
2
3 #include <iomanip>
4 #include <iostream>
5
6 using namespace std;
7
8 /**
9  * Demonstracni ukazka backtrackingu.
10 * Cilem ukazky je implementace sachove ulohy zname pod nazvem Jezdcova
       prochazka, anglicky Kniht's tour.
11 * Informace o uloze lze najit napriklad na http://cs.wikipedia.org/
      wiki/Jezdcova_proch%C3%A1zka
12 *
13 * @author     Jiri Dvorsky <jiri.dvorsky@vsb.cz>
14 *
15 * Prostor jmen pro projekt KnightsTour.
16 */
17 namespace KnightsTour
```

## Zdrojové kódy řešení, soubor KnightsTour.h (pokrač.)

```
18 {
19   /**
20    * Velikost sachovnice
21    */
22   const int BoardSize = 8;
23
24   /**
25    * Hodnota volneho, dosud nenavstiveneho, pole sachovnice
26    */
27   const int FreeCell = 0;
28
29   /**
30    * Pocet vsech potencialne moznych tahu konem z policka sachovnice
31    */
32   const int NumberOfKnightMoves = 8;
33
34   /**
35    * Zmeny hodnoty radku pro jednotlive mozne tahy konem
36    */
```



## Zdrojové kódy řešení, soubor KnightsTour.h (pokrač.)

```
37   const int RowMoveDelta[NumberOfKnightMoves] = {-2, -1, +1, +2, +2,
         +1, -1, -2};
38
39   /**
40    * Zmeny hodnoty sloupce pro jednotlive mozne tahy konem
41    */
42   const int ColumnMoveDelta[NumberOfKnightMoves] = {+1, +2, +2, +1,
         -1, -2, -2, -1};
43
44   /**
45    * Struktura pro ulozeni pozice kone na sachovnici
46    */
47   struct Position
48   {
49     /**
50      * radek dane pozice
51      */
52     int Row;
53     /**
```



## Zdrojové kódy řešení, soubor KnightsTour.h (pokrač.)

```
54       * Sloupec dane pozice
55       */
56     int Column;
57   };
58
59
60   /**
61    * Inicializace sachovnice.
62    *
63    * Funkce vyplni vsechna policka sachovnice nulami, coz znamena, ze
         na dane pole kun dosud neskocil.
64    *
65    * @param Board Inicializovana sachovnice
66    */
67   void InitBoard(int Board[BoardSize][BoardSize]);
68
69
70   /**
71    * Vystup sachovnice na standardni vystup (obrazovku).
```



## Zdrojové kódy řešení, soubor KnightsTour.h (pokrač.)

```
72    *
73    * @param Board Vypisovana sachovnice
74    */
75   void PrintBoard(const int Board[BoardSize][BoardSize]);
76
77
78   /**
79    * Posun konem po sachovnici.
80    *
81    * Funkce provede jeden z NumberOfKnightMoves moznych tahu.
82    * Funkce nezkouma, jestli provedeny tah je platny (napriklad, zda
         se kun neocitnul mimo sachovnici).
83    *
84    * @param CurrentPos Soucasna poloha kone na sachovnici
85    * @param MoveIndex cislo tahu, ktery z NumberOfKnightMoves moznych
         tahu chceme provest.
86    * @return Funkce vraci novou polohu kone na sachovnici.
87    */
88   Position Move(const Position& CurrentPos, const int MoveIndex);
```

## Zdrojové kódy řešení, soubor KnightsTour.h (pokrač.)

```
89
90
91      /**
92       * Test, zda je tah platny.
93       *
94       * Tah je platny pokud se kun neocitnul mimo sachovnici a pokud
             neskocil na policko, kde jiz jednou byl.
95       *
96       * @param Pos Soucasna poloha kone na sachovnici
97       * @param Board sachovnice
98       * @return Funkce vraci true pokud je tah platny, jinak vraci false.
99       */
100     bool IsValidMove(const Position& Pos, const int Board[BoardSize][
             BoardSize]);
101
102
103     /**
104      * Rekurzivni hledani reseni ulohy, implementace pomoci
             backtrackingu.
```



## Zdrojové kódy řešení, soubor KnightsTour.h (pokrač.)

```
105      *
106      * Funkce postupne provadi jednotlive mozne tahy konem ze soucasne
             polohy kone na sachovnici (celkem az NumberOfKnightMoves tahu).
107      * Pokud ja takto provedeny tah platny, tah se zaznamena do
             sachovnice a rekurzivne se pokracuje dal.
108      * Pokud funkce vycerpa vsechny mozne tahy ze soucasne pozice, vraci
              funkce false jako signal, ze reseni
109      * z teto polohy kone na sachovnici neni mozne v dalsich tazich
             najit. Prohledavani se tedy vraci o jeden tah zpet (vraceni
             zpet - backtracking).
110      *
111      * Funkce hleda prvni mozne reseni, nikoliv vsechna mozna reseni.
112      *
113      * @param Board sachovnice, v sachovnici jsou zaznamenany vsechny
             tahy pocinaje prvnim (cislo 1) a konce tahem s cislem
             MoveNumber-1.
114      * @param CurrentPos Soucasna poloha kone na sachovnici, poloha v
             tahu s cislem MoveNumber-1.
115      * @param MoveNumber cislo tahu, ktery prave resime
```



## Zdrojové kódy řešení, soubor KnightsTour.h (pokrač.)

```
116      * @return Funkce vraci true pokud bylo v naslednych tazich po tahu
             s cislem MoveNumber nalezeno reseni, jinak vraci false.
117      */
118     bool SolveKnightsTour(int Board[BoardSize][BoardSize], const
             Position& CurrentPos, const int MoveNumber);
119 }
```



## Zdrojové kódy řešení, soubor KnightsTour.cpp

```
1  #include "KnightsTour.h"
2
3  namespace KnightsTour
4  {
5     void InitBoard(int Board[BoardSize][BoardSize])
6     {
7        for(int i = 0; i < BoardSize; i++)
8        {
9           for(int j = 0; j < BoardSize; j++)
10          {
11             Board[i][j] = FreeCell;
12          }
13       }
14    }
15
16    void PrintBoard(const int Board[BoardSize][BoardSize])
17    {
18       for(int i = 0; i < BoardSize; i++)
19       {
```

## Zdrojové kódy řešení, soubor KnightsTour.cpp (pokrač.)

```
20          for(int j = 0; j < BoardSize; j++)
21          {
22             cout << setw(4) << Board[i][j];
23          }
24          cout << endl;
25       }
26       cout << endl;
27    }
28
29    Position Move(const Position& CurrentPos, const int MoveIndex)
30    {
31       Position NewPos;
32       NewPos.Row = CurrentPos.Row + RowMoveDelta[MoveIndex];
33       NewPos.Column = CurrentPos.Column + ColumnMoveDelta[MoveIndex];
34       return NewPos;
35    }
36
37    bool IsValidMove(const Position& Pos, const int Board[BoardSize][
          BoardSize])
```



## Zdrojové kódy řešení, soubor KnightsTour.cpp (pokrač.)

```
38    {
39       if (0 <= Pos.Row && Pos.Row < BoardSize && 0 <= Pos.Column && Pos
             .Column < BoardSize)
40       {
41          return Board[Pos.Row][Pos.Column] == FreeCell;
42       }
43       return false;
44    }
45
46    bool SolveKnightsTour(int Board[BoardSize][BoardSize], const
          Position& CurrentPos, const int MoveNumber)
47    {
48       if (MoveNumber == BoardSize*BoardSize + 1)
49       {
50          return true;
51       }
52       for(int i = 0; i < NumberOfKnightMoves; i++)
53       {
54          Position p = Move(CurrentPos, i);
```



## Zdrojové kódy řešení, soubor KnightsTour.cpp (pokrač.)

```
55          if (IsValidMove(p, Board))
56          {
57             Board[p.Row][p.Column] = MoveNumber;
58             if (SolveKnightsTour(Board, p, MoveNumber+1))
59             {
60                return true;
61             }
62             Board[p.Row][p.Column] = FreeCell;
63          }
64       }
65       return false;
66    }
67 }
```



## Zdrojové kódy řešení, soubor main.cpp

```
1  #include "KnightsTour.h"
2
3  using namespace KnightsTour;
4
5  void main()
6  {
7     int Board[BoardSize][BoardSize];
8     InitBoard(Board);
9     Position InitPos;
10    // vychozi pozice je zvolena nahodne
11    InitPos.Row = 0;
12    InitPos.Column = 0;
13    Board[InitPos.Row][InitPos.Column] = 1;
14    bool HasSolution = SolveKnightsTour(Board, InitPos, 2);
15    if (HasSolution)
16    {
17       PrintBoard(Board);
18    }
19    else
```

## Zdrojové kódy řešení, soubor main.cpp (pokrač.)

```
20    {
21        cout << "Reseni nenalezeno!" << endl;
22    }
23 }
```



## Děkuji za pozornost



# Vyhledávání

doc. Mgr. Jiří Dvorský, Ph.D.

Katedra informatiky
Fakulta elektrotechniky a informatiky
VŠB – TU Ostrava





## Vyhledávání – definice problému

### Základní pojmy

- vyhledávací prostor $S$,
- klíč,
- hodnota.

### Příklad

- seznam loginů studentů a jejich hodnocení v předmětu Algoritmy I,
- autorský katalog v knihovně,
- množina všech českých samohlásek.

## Vyhledávání – jednorozměrné vyhledávání

- adresní vyhledávací algoritmy – využívají jednoznačného vztahu mezi hodnotou klíče prvku a umístěním prvku ve struktuře reprezentující vyhledávací prostor $S$ (přímý přístup k prvkům pole pomocí indexů, hašovací tabulky)
- asociativní vyhledávací algoritmy – porovnávání prvků (relativní hodnota vzhledem k ostatním prvkům) při hledání prvku $x \in S$ se využívají relace (porovnávání) mezi prvky struktury reprezentující $S$ (vyhledávání v poli).



## Vyhledávání – vícerozměrné vyhledávání

- nevyhledáváme jen podle jednoho klíče, ale podle více klíčů,
- nutnost přizpůsobit datové struktury a algoritmy,
- samotné vyhledávací algoritmy se dají rozdělit do tří kategorií:
  - dotazy na úplnou shodu – dotazy požadující shodu na všech klíčích.
  - dotazy na částečnou shodu – dotazy požadující shodu jen na některých složkách klíče.
  - dotazy na intervalovou shodu – dotazy požadující, aby klíč záznamu byl v určeném intervalu.

Do této kategorie vlastně spadají všechny databázové systémy.



## Vyhledávání – typické úlohy

příslušnost prvku – je obsažen nebo ne? Matematicky $x \in S$ nebo $x \notin S$

hledání extrému – minimum nebo maximum

dotaz na shodu – úplnou nebo částečnou

Při konkrétní implementaci zadané úlohy existuje vzájemná souvislost mezi všemi operacemi a všechny operace závisejí na zvolené reprezentaci vyhledávacího prostoru.



## Vyhledávání v poli

- nejjednodušší paměťová struktura,
- přímý přístup k prvkům,
- typicky jednorozměrné vyhledávání (klíč, hodnota),

**Řešené úlohy**

příslušnost prvku
- výsledek jako logická hodnota
- výsledek index hledaného prvku v poli

dotaz na úplnou shodu

Budeme předpokládat jeden výskyt každého prvku v poli

## Složitost vyhledávání

### Prostorová složitost

Konstantní – jen prohledávané pole a konstantní rozsah pomocných proměnných

### Časová složitost

Jaké operace zkoumat?
Porovnání hledaného prvku s prvkem v poli.



## Sekvenční vyhledávání

- bez požadavků na prohledávané pole,
- prohledávané pole tedy nemusí být setříděné,
- algoritmus prohledává celé pole – postupně testuje jednotlivé prvky pole,
- složitost $O(n)$, kde $n$ je počet prvků v prohledávaném poli.



## Iterativní implementace

```
1 bool LinearSearch1(const int a[], const int n, const int x)
2 {
3    for (int i = 0; i < n; i++)
4    {
5       if (a[i] == x)
6       {
7          return true;
8       }
9    }
10   return false;
11 }
```

Funkce vrací `true`, pokud se prvek `x` nachází v poli `a`, jinak vrací `false`.



## Iterativní implementace

```
1 int LinearSearch2(const int a[], const int n, const int x)
2 {
3    for (int i = 0; i < n; i++)
4    {
5       if (a[i] == x)
6       {
7          return i;
8       }
9    }
10   return -1;
11 }
```

Funkce vrací index prvního výskytu prvku `x` v poli `a`, jinak vrací -1.

## Rekurzivní implementace

```
1 bool LinearSearchRecursive1(const int a[], const int n, const int x,
      const int i)
2 {
3   if (i == n)
4   {
5     return false;
6   }
7   if (a[i] == x)
8   {
9     return true;
10  }
11  return LinearSearchRecursive1(a, n, x, i + 1);
12 }
```

Funkce vrací `true`, pokud se prvek `x` nachází v poli `a`, jinak vrací `false`.



## Rekurzivní implementace

```
1 int LinearSearchRecursive2(const int a[], const int n, const int x,
      const int i)
2 {
3   if (i == n)
4   {
5     return -1;
6   }
7   if (a[i] == x)
8   {
9     return i;
10  }
11  return LinearSearchRecursive2(a, n, x, i + 1);
12 }
```

Funkce vrací index prvního výskytu prvku `x` v poli `a`, jinak vrací -1.



## Rekurzivní implementace, alternativní přístup

```
1 bool LinearSearchRecursive3(const int a[], const int l, const int r,
      const int x)
2 {
3   if (l > r)
4   {
5     return false;
6   }
7   int m = (l + r) / 2;
8   if (x == a[m])
9   {
10    return true;
11  }
12  return LinearSearchRecursive3(a, l, m - 1, x) ||
        LinearSearchRecursive3(a, m + 1, r, x);
13 }
```

Funkce vrací `true`, pokud se prvek `x` nachází v poli `a`, jinak vrací `false`.



## Rekurzivní implementace, alternativní přístup

```
1 bool LinearSearchRecursive4(const int a[], const int l, const int r,
      const int x)
2 {
3   if (l > r)
4   {
5     return false;
6   }
7   int m = (l + r) / 2;
8   if (x == a[m])
9   {
10    return true;
11  }
12  if (LinearSearchRecursive4(a, l, m - 1, x))
13  {
14    return true;
15  }
16  return LinearSearchRecursive4(a, m + 1, r, x);
17 }
```

## Rekurzivní implementace, alternativní přístup (pokrač.)

Funkce vrací `true`, pokud se prvek `x` nachází v poli `a`, jinak vrací `false`.



## Vyhledávání půlením intervalu

- Lze zlepšit časovou složitost sekvenčního vyhledávání?
- Pokud o prohledávaném poli nevíme vůbec nic asi těžko?
- Co kdybychom pole setřídili? Budeme předpokládat, že pole je setříděno vzestupně.
- Potom bychom nemuseli prohledávat pole celé. Po nalezení většího prvku než je hledaný už nemá cenu dále hledat.
- Pokud použijeme techniku dělení pole na poloviny, musíme stále prohledávat obě poloviny?
- Mohlo by nám uspořádání prvků v poli napovědět v které polovině máme hledat a kde to nemá cenu?



## Půlení intervalu, úspěšné hledání prvku 19

Určíme střed celého pole a srovnáme 19 s 34

| 7 | 11 | 16 | 19 | 23 | 31 | 34 | 40 | 43 | 52 | 56 | 61 | 65 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|



## Půlení intervalu, úspěšné hledání prvku 19

Určíme střed celého pole a srovnáme 19 s 34

| 7 | 11 | 16 | 19 | 23 | 31 | 34 | 40 | 43 | 52 | 56 | 61 | 65 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|

Určíme střed úseku pole a srovnáme 19 s 16

| 7 | 11 | 16 | 19 | 23 | 31 | 34 | 40 | 43 | 52 | 56 | 61 | 65 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|

## Půlení intervalu, úspěšné hledání prvku 19

Určíme střed celého pole a srovnáme 19 s 34

| 7 | 11 | 16 | 19 | 23 | 31 | 34 | 40 | 43 | 52 | 56 | 61 | 65 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|

Určíme střed úseku pole a srovnáme 19 s 16

| 7 | 11 | 16 | 19 | 23 | 31 | 34 | 40 | 43 | 52 | 56 | 61 | 65 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|

Určíme střed úseku pole a srovnáme 19 s 23

| 7 | 11 | 16 | 19 | 23 | 31 | 34 | 40 | 43 | 52 | 56 | 61 | 65 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|



## Půlení intervalu, úspěšné hledání prvku 19

Určíme střed celého pole a srovnáme 19 s 34

| 7 | 11 | 16 | 19 | 23 | 31 | 34 | 40 | 43 | 52 | 56 | 61 | 65 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|

Určíme střed úseku pole a srovnáme 19 s 16

| 7 | 11 | 16 | 19 | 23 | 31 | 34 | 40 | 43 | 52 | 56 | 61 | 65 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|

Určíme střed úseku pole a srovnáme 19 s 23

| 7 | 11 | 16 | 19 | 23 | 31 | 34 | 40 | 43 | 52 | 56 | 61 | 65 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|

Určíme střed úseku pole a srovnáme 19 s 19

| 7 | 11 | 16 | 19 | 23 | 31 | 34 | 40 | 43 | 52 | 56 | 61 | 65 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|

**Prvek 19 byl úspěšně nalezen!**



## Půlení intervalu, úspěšné hledání prvku 19

Zkráceně můžeme postup hledání prvku 19 zachytit takto:

| 7 | 11 | 16 | 19 | 23 | 31 | 34 | 40 | 43 | 52 | 56 | 61 | 65 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|

| 7 | 11 | 16 | 19 | 23 | 31 | 34 | 40 | 43 | 52 | 56 | 61 | 65 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|

| 7 | 11 | 16 | 19 | 23 | 31 | 34 | 40 | 43 | 52 | 56 | 61 | 65 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|

| 7 | 11 | 16 | 19 | 23 | 31 | 34 | 40 | 43 | 52 | 56 | 61 | 65 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|

**Poznámka**

Celé pole můžeme považovat také za „úsek pole“, který je shodou okolností roven celému poli. Tím se nám zjednoduší další výklad a v důsledku i výsledný algoritmus.



## Půlení intervalu, úspěšné hledání prvku 7

| 7 | 11 | 16 | 19 | 23 | 31 | 34 | 40 | 43 | 52 | 56 | 61 | 65 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|

| 7 | 11 | 16 | 19 | 23 | 31 | 34 | 40 | 43 | 52 | 56 | 61 | 65 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|

| 7 | 11 | 16 | 19 | 23 | 31 | 34 | 40 | 43 | 52 | 56 | 61 | 65 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|

## Půlení intervalu, úspěšné hledání prvku 65

| 7 | 11 | 16 | 19 | 23 | 31 | 34 | 40 | 43 | 52 | 56 | 61 | 65 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|

| 7 | 11 | 16 | 19 | 23 | 31 | 34 | 40 | 43 | 52 | 56 | 61 | 65 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|

| 7 | 11 | 16 | 19 | 23 | 31 | 34 | 40 | 43 | 52 | 56 | 61 | 65 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|

| 7 | 11 | 16 | 19 | 23 | 31 | 34 | 40 | 43 | 52 | 56 | 61 | 65 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|



## Půlení intervalu – úsek pole

### Jak reprezentovat úsek pole?

- L – index levého okraje prohledávaného úseku pole, **L**eft
- R – index pravého okraje prohledávaného úseku pole, **R**ight
- M – index středu prohledávaného úseku pole, **M**id

### Indexy L a R

Indexy **L** a **R** jsou inkluzivní – první a poslední prvek úseku leží na těchto indexech.
Srovnej s délkou pole a indexem posledního prvku pole v jazyku C++.

### Poznámka

Index levého resp. pravého okraje úseku pole budeme také nazývat levou resp. pravou **mezí** úseku pole.



## Půlení intervalu – výpočet středu úseku

### Výpočet středu úseku

$$M = L + \frac{R - L}{2} = \frac{2L + R - L}{2} = \frac{L + R}{2}$$

### Konvence při výpočtu středu úseku

- Je jasná, že součet $L + R$ nemusí být sudé číslo. Co s tím?!
- Dělení vždy probíhá **celočíselně**! Například $L = 0$ a $R = 5$. Střed je $M = 2$, nikoliv $M = 2.5$.
- Levá část děleného úseku tak bude o jeden prvek kratší.
- Pokud bychom dělili s plovoucí řádovou čárkou a zaokrouhlovali nahoru, jak je obvyklé, dosáhli bychom jedině toho, že levá část by byla o jeden prvek delší než pravá.



## Půlení intervalu, úspěšné hledání prvku 19

### Úseky pole

| 7 | 11 | 16 | 19 | 23 | 31 | 34 | 40 | 43 | 52 | 56 | 61 | 65 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|

| 7 | 11 | 16 | 19 | 23 | 31 | 34 | 40 | 43 | 52 | 56 | 61 | 65 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|

| 7 | 11 | 16 | 19 | 23 | 31 | 34 | 40 | 43 | 52 | 56 | 61 | 65 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|

| 7 | 11 | 16 | 19 | 23 | 31 | 34 | 40 | 43 | 52 | 56 | 61 | 65 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|

### Indexy

| L | R | M |
|---|---|---|
| 0 | 12 | 6 |
| 0 | 5 | 2 |
| 3 | 5 | 4 |
| 3 | 3 | 3 |

### Posun doleva

| | Nová hodnota |
|---|---|
| L | beze změny |
| R | $M - 1$ |

### Posun doprava

| | Nová hodnota |
|---|---|
| L | $M + 1$ |
| R | beze změny |

## Půlení intervalu, neúspěšné hledání prvku 55

Určíme střed celého pole a srovnáme 55 s 34

| 7 | 11 | 16 | 19 | 23 | 31 | 34 | 40 | 43 | 52 | 56 | 61 | 65 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|

Určíme střed úseku pole a srovnáme 55 s 52

| 7 | 11 | 16 | 19 | 23 | 31 | 34 | 40 | 43 | 52 | 56 | 61 | 65 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|

Určíme střed úseku pole a srovnáme 55 s 61

| 7 | 11 | 16 | 19 | 23 | 31 | 34 | 40 | 43 | 52 | 56 | 61 | 65 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|

Určíme střed úseku pole a srovnáme 55 s 56

| 7 | 11 | 16 | 19 | 23 | 31 | 34 | 40 | 43 | 52 | 56 | 61 | 65 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|

**Prvek 55 nebyl nalezen!**



## Půlení intervalu, neúspěšné hledání prvku 55

### Jak rozpoznat konec u neúspěšného hledání?

- Konec rozpoznáme snadno – úsek obsahuje jen jeden prvek a není to ten, který hledáme.
- Ale! Při úspěšném hledání prvku 19 jsme taky skončili u jednoprvkového úseku.
- Jsou tedy jednoprvkové úseky nějakou zvláštností? Proč by měly?
- Takže bychom měli vyhledávání ukončit až úsek zkrátíme na jeden prvek?
- A když se u více prvkového úseku „trefíme“ tak, že hledaný prvek bude přímo ve středu. To bychom mohli skončit rovnou, ne?
- Takže redukce na jeden prvek není asi nutná.
- A opačná otázka? Jak poznám, že mám ve vyhledávání naopak stále pokračovat?



## Půlení intervalu, invariant algoritmu

- Při výpočtu středu úseku

$$M = L + \frac{R - L}{2}$$

  předpokládáme, že $R \geq L$!
- Otázka zní, platí tento předpoklad po celou dobu vykonávání algoritmu?
- A pokud toto neplatí, kdy je předpoklad porušen?

### Invariant algoritmu

**Invariant** je podmínka v algoritmu, která musí být splněna po celou dobu vykonávání algoritmu.

### Invariant algoritmu vyhledávání půlením intervalu

V našem případě musí vždy platit, že $R \geq L$.



## Půlení intervalu, neúspěšné hledání prvku 55

### Úseky pole

| 7 | 11 | 16 | 19 | 23 | 31 | 34 | 40 | 43 | 52 | 56 | 61 | 65 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|

| 7 | 11 | 16 | 19 | 23 | 31 | 34 | 40 | 43 | 52 | 56 | 61 | 65 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|

| 7 | 11 | 16 | 19 | 23 | 31 | 34 | 40 | 43 | 52 | 56 | 61 | 65 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|

| 7 | 11 | 16 | 19 | 23 | 31 | 34 | 40 | 43 | 52 | 56 | 61 | 65 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|

Prvek 55 je menší než 56, „pokračujeme“ levou částí

### Indexy

| L | R | M |
|---|---|---|
| 0 | 12 | 6 |
| 7 | 12 | 9 |
| 10 | 12 | 11 |
| 10 | 10 | 10 |
| **10** | **9** | |

### Porušení invariantu algoritmu

Levá mez je větší než pravá. Došlo k překřížení mezí.

## Odhad časové složitosti

- Výchozí poznatky:
  1. prohledávaný úsek pole je opakovaně dělen na poloviny,
  2. dělení končí u úseku o jednom prvku,
  3. každé dělení je provedeno po porovnání jednoho prvku.
- Počet porovnání tedy odpovídá počtu dělení pole.
- Kolik takových dělení je ale nutné?
- Předpokládejme, že počet prvků v poli $n$ je mocninou čísla 2, tj. $n = 2^k$. Pokud ne, vezmeme nejbližší větší mocninu tj. $2^{k-1} < n \leq 2^k$.



## Odhad časové složitosti (pokrač.)

| Pořadí dělení | Pole rozděleno na úseků | Počet prvků v úseku | Porovnáno prvků v úseku |
|---|---|---|---|
| 0 | $1 = 2^0$ | $\frac{n}{1}$ | 1 |
| 1 | $2 = 2^1$ | $\frac{n}{2}$ | 1 |
| 2 | $4 = 2^2$ | $\frac{n}{4}$ | 1 |
| 3 | $8 = 2^3$ | $\frac{n}{8}$ | 1 |
| 4 | $16 = 2^4$ | $\frac{n}{16}$ | 1 |
| ⋮ | ⋮ | ⋮ | ⋮ |
| k | $2^k$ | $\frac{n}{2^k} = 1$ | 1 |



## Odhad časové složitosti (pokrač.)

- Po provedení $k+1$ dělení pole jsme se dostali k úseku s jediným prvkem a provedli jsme při tom celkem $k+1$ porovnání.
- Horní odhad složitosti vyhledávání půlením intervalu tedy je $O(k+1)$.
- Jak vypočítáme $k$? Řešením rovnice

$$\begin{aligned} n &= 2^k \\ \log_2 n &= k \log_2 2 \\ \log_2 n &= k \end{aligned}$$

- Odhad složitosti tedy můžeme psát jako $O(\log_2 n + 1)$.
- Vzhledem k tomu, že rozhodující člen této funkce je výraz $\log_2 n$, můžeme psát zjednodušeně

$$O(\log_2 n)$$



## Rekurzivní implementace

```
1  bool BinarySearchRecursive1(const int a[], const int l, const int r,
       const int x)
2  {
3    if (l > r)
4    {
5      return false;
6    }
7    int m = (l + r) / 2;
8    if (x == a[m])
9    {
10     return true;
11   }
12   if (x < a[m])
13   {
14     return BinarySearchRecursive1(a, l, m - 1, x);
15   }
16   return BinarySearchRecursive1(a, m + 1, r, x);
17 }
```

## Rekurzivní implementace (pokrač.)

Funkce vrací `true`, pokud se prvek `x` nachází v poli `a`, jinak vrací `false`.



## Rekurzivní implementace

```
1 bool BinarySearchRecursive1a(const int a[], const int l, const int r,
      const int x)
2 {
3   if (l > r)
4   {
5     return false;
6   }
7   int m = (l + r) / 2;
8   if (x == a[m])
9   {
10    return true;
11  }
12  return x < a[m] ? BinarySearchRecursive1a(a, l, m - 1, x) :
        BinarySearchRecursive1a(a, m + 1, r, x);
13 }
```

Funkce vrací `true`, pokud se prvek `x` nachází v poli `a`, jinak vrací `false`. Místo příkazu `if` použit ternární operátor.



## Rekurzivní implementace

```
1 int BinarySearchRecursive2(const int a[], const int l, const int r,
      const int x)
2 {
3   if (l > r)
4   {
5     return -1;
6   }
7   int m = (l + r) / 2;
8   if (x == a[m])
9   {
10    return m;
11  }
12  if (x < a[m])
13  {
14    return BinarySearchRecursive2(a, l, m - 1, x);
15  }
16  return BinarySearchRecursive2(a, m + 1, r, x);
17 }
```



## Rekurzivní implementace (pokrač.)

Výsledkem je index nalezeného prvku, jinak -1.

## Rekurzivní implementace

```
1 int BinarySearchRecursive3(const int a[], const int l, const int r,
      const int x)
2 {
3    if (l > r)
4    {
5       return ~l;
6    }
7    int m = (l + r) / 2;
8    if (x == a[m])
9    {
10      return m;
11   }
12   if (x < a[m])
13   {
14      return BinarySearchRecursive3(a, l, m - 1, x);
15   }
16   return BinarySearchRecursive3(a, m + 1, r, x);
17 }
```



## Rekurzivní implementace (pokrač.)

Výsledkem je index nalezeného prvku, jinak bitový doplněk správné pozice.



## Iterativní implementace

```
1 bool BinarySearch1(const int a[], const int n, const int x)
2 {
3    int l = 0;
4    int r = n - 1;
5    while (l <= r)
6    {
7       int m = (l + r) / 2;
8       if (x == a[m])
9       {
10         return true;
11      }
12      if (x < a[m])
13      {
14         r = m - 1;
15      }
16      else
17      {
18         l = m + 1;
19      }
```



## Iterativní implementace (pokrač.)

```
20    }
21    return false;
22 }
```

Funkce vrací `true`, pokud se prvek `x` nachází v poli `a`, jinak vrací `false`.

## Iterativní implementace

```
1 int BinarySearch2(const int a[], const int n, const int x)
2 {
3    int l = 0;
4    int r = n - 1;
5    while (l <= r)
6    {
7       int m = (l + r) / 2;
8       if (x == a[m])
9       {
10         return m;
11      }
12      if (x < a[m])
13      {
14         r = m - 1;
15      }
16      else
17      {
18         l = m + 1;
19      }
```



## Iterativní implementace (pokrač.)

```
20    }
21    return -1;
22 }
```

Výsledkem je index nalezeného prvku, jinak -1.



## Iterativní implementace

```
1 int BinarySearch3(const int a[], const int n, const int x)
2 {
3    int l = 0;
4    int r = n - 1;
5    while (l <= r)
6    {
7       int m = (l + r) / 2;
8       if (x == a[m])
9       {
10         return m;
11      }
12      if (x < a[m])
13      {
14         r = m - 1;
15      }
16      else
17      {
18         l = m + 1;
19      }
```



## Iterativní implementace (pokrač.)

```
20    }
21    return ~l;
22 }
```

Výsledkem je index nalezeného prvku, jinak bitový doplněk správné pozice.

# Děkuji za pozornost



# Třídění

doc. Mgr. Jiří Dvorský, Ph.D.

Katedra informatiky
Fakulta elektrotechniky a informatiky
VŠB – TU Ostrava





## RadixSort

```
1 int GetBit(const int Value, const int BitPosition)
2 {
3    return (Value >> BitPosition) & 1;
4 }
```



## RadixSort

```
1  void RadixSortInternal(int a[], const int l, const int r, const int b)
2  {
3     if (r > l && b >= 0)
4     {
5        int i = l;
6        int j = r;
7        do
8        {
9           while (GetBit(a[i], b) == 0 && i < j)
10             i++;
11          while (GetBit(a[j], b) == 1 && i < j)
12             j--;
13          Exchange(a[i], a[j]);
14       } while (j != i);
15       if (GetBit(a[r], b) == 0)
16          j++;
17       RadixSortInternal(a, l, j - 1, b - 1);
18       RadixSortInternal(a, j, r, b - 1);
19    }
20 }
```

## RadixSort

```
1 void RadixSort(int a[], const int n)
2 {
3    const int BitsPerInt = sizeof(int) * CHAR_BIT;
4    RadixSortInternal(a, 0, n - 1, BitsPerInt - 1);
5 }
```



## RadixSort – náhodná permutace

Animace jsou v poznámkách k prezentaci vynechány.



## RadixSort – identická permutace

Animace jsou v poznámkách k prezentaci vynechány.



## RadixSort – opačná permutace

Animace jsou v poznámkách k prezentaci vynechány.

## RadixSort – permutace pouze několika hodnot

Animace jsou v poznámkách k prezentaci vynechány.



## Děkuji za pozornost



# Základní třídící algoritmy

doc. Mgr. Jiří Dvorský, Ph.D.

Katedra informatiky
Fakulta elektrotechniky a informatiky
VŠB – TU Ostrava





## Asociativní třídicí algoritmy

- jsou nejvšeobecnější skupinou třídících algoritmů,
- nepředpokládají o prvcích tříděné množiny nic víc než že jsou vybrané z uspořádaného univerza,
- jinak řečeno mezi prvky je definována relace „menší nebo rovno“,
- jsou založeny na operacích **porovnání** a **přesun** (výměna) dvou prvků,
- tyto dvě operace významně ovlivňují složitost asociativních třídicích algoritmů.

## Složitost asociativních třídicích algoritmů

- Složitost asociativních třídících algoritmů budeme měřit počtem porovnání a přesunů prvků.
- Porovnání a přesuny prvků nemusí být triviální operace.
- Porovnání se týká jen klíče, zatímco přesun se týká celého záznamu.
- Markantní rozdíl mezi cenou porovnání a cenou přesunu se projeví zvláště tehdy, když je délka záznamu podstatně větší než délka klíče.



## Složitost asociativních třídicích algoritmů

- Efektivnost některých asociativních třídicích algoritmů závisí od toho, v jakém pořadí jsou uspořádané prvky ve vstupní posloupnosti.
- Je-li vstupní posloupnost již setříděná, neměl by třídící algoritmus, po odhalení tohoto faktu, vykonávat žádnou činnost.
- Složitost třídícího algoritmu by měla vzrůstat s mírou nesetříděnosti posloupnosti.
- Třídící algoritmus by měl být **přirozený**.



## Míra nesetříděnosti posloupnosti

- Budeme ji definovat jako počet inverzí v permutaci.
- Uvažujme permutaci $\pi : \{1, \ldots, n\} \to \{1, \ldots, n\}$
- Řekneme, že dvojice různých prvků $(i, j)$ představuje **inverzi permutace** $\pi$, jestliže
$$i \leq j \wedge \pi_i > \pi_j.$$
- Inverzi permutace můžeme volně interpretovat tak, že „na menším indexu je větší prvek a naopak na větším indexu je menší prvek“.
- Ještě jinak řečeno, tyto dva prvky jsou přehozeny, za předpokladu, že chceme posloupnost třídit vzestupně.



## Míra nesetříděnosti posloupnosti

**Příklad**

$$\pi = \begin{pmatrix} 1 & 2 & 3 & 4 \\ 4 & 1 & 3 & 2 \end{pmatrix}$$

Inverze:

- $1 < 2 \wedge 4 > 1$
- $1 < 3 \wedge 4 > 3$
- $1 < 4 \wedge 4 > 2$
- $3 < 4 \wedge 3 > 2$

## Míra nesetříděnosti posloupnosti

- Počet inverzí setříděné posloupnosti je 0.
- Maximální počet inverzí, rovný $\frac{1}{2}n(n-1)$, obsahuje posloupnost setříděná v opačném pořadí.
- Průměrný počet inverzí v posloupnosti je přibližně $\frac{1}{4}n(n-1)$, za předpokladu, že každá posloupnosti je stejně pravděpodobná.
- Uvažujme všechny permutace $n$ prvků $\{1, 2, \ldots, n\}$.
- Permutace $n$ prvků vytvoříme přidáním čísla $n$ k $(n-1)!$ permutacím $n-1$ prvků $\{1, 2, \ldots, n-1\}$.
- Číslo $n$ můžeme přidat:
  - na poslední místo – pak se počet inverzí nezvýší,
  - na předposlední místo – pak přibude jedna inverze, a tak dále až
  - na první místo – pak přibude $n-1$ inverzí.



## Míra nesetříděnosti posloupnosti

Pro průměrný počet inverzí $I_n$ tedy bude platit:

$$\begin{aligned} n!I_n &= (n-1)!I_{n-1} + 0 \cdot (n-1)! + \\ &\quad (n-1)!I_{n-1} + 1 \cdot (n-1)! + \\ &\quad (n-1)!I_{n-1} + 2 \cdot (n-1)! + \\ &\quad (n-1)!I_{n-1} + (n-1) \cdot (n-1)! \end{aligned}$$

Tedy

$$\begin{aligned} n!I_n &= n(n-1)!I_{n-1} + (0 + 1 + 2 + \cdots + (n-1)) \cdot (n-1)! \\ &= n!I_{n-1} + \frac{1}{2}n(n-1)(n-1)! \\ &= n!I_{n-1} + \frac{1}{2}(n-1)n! \end{aligned}$$



## Míra nesetříděnosti posloupnosti

Odkud

$$\begin{aligned} I_n &= I_{n-1} + \frac{1}{2}(n-1) \\ &\vdots \\ &= \frac{1}{2}(n-1) + \frac{1}{2}(n-2) + \frac{1}{2}(n-3) + \frac{1}{2} \cdot 1 \\ &= \frac{1}{4}n(n-1) \end{aligned}$$



## Asociativní třídicí algoritmy

**Princip**

Na algoritmy asociativního třídění se tedy můžeme dívat jako na algoritmy odstraňující postupně inverze ze vstupní posloupnosti pomocí vzájemných výměn vhodných prvků, do té doby, dokud posloupnost neobsahuje žádné inverze.

## Vizualizace permutací

Permutaci

$$\pi = \begin{pmatrix} 1 & 2 & \dots & n \\ \pi_1 & \pi_2 & \dots & \pi_n \end{pmatrix}$$

můžeme znázornit jako posloupnost $n$ svislých pruhů, kde

- pořadí pruhu odpovídá indexu $i$ a
- výška pruhu odpovídá hodnotě $\pi_i$.

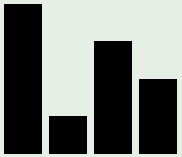
Šířka všech pruhů je stejná a ve vizualizaci nehraje roli.

**Příklad**

Permutaci

$$\pi = \begin{pmatrix} 1 & 2 & 3 & 4 \\ 4 & 1 & 3 & 2 \end{pmatrix}$$

odpovídá zobrazení





## Vizualizace permutací – ukázky

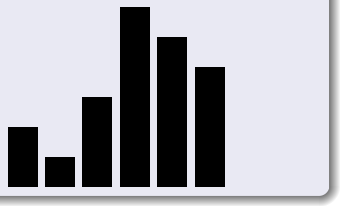
Identická



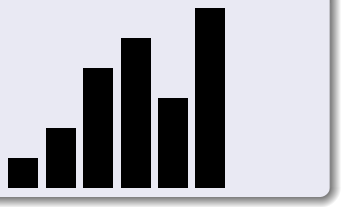
Opačně setříděná



Náhodná



Téměř setříděná





## Obecná funkce pro výměnu dvou prvků pole

```
1 void Exchange(int& x, int& y)
2 {
3    int aux = x;
4    x = y;
5    y = aux;
6 }
```



## Třídění výběrem – SelectSort

```
1 void SelectSort(int a[], const int n)
2 {
3    for (int i = 0; i < n; i++)
4    {
5       int min = i;
6       for (int j = i + 1; j < n; j++)
7       {
8          if (a[j] < a[min])
9          {
10            min = j;
11         }
12      }
13      Exchange(a[min], a[i]);
14   }
15 }
```

Nahrazení výměny prvků funkcí `Exchange`.

## Třídění výběrem – náhodná permutace

Animace jsou v poznámkách k prezentaci vynechány.



## Třídění výběrem – identická permutace

Animace jsou v poznámkách k prezentaci vynechány.



## Třídění výběrem – opačná permutace

Animace jsou v poznámkách k prezentaci vynechány.



## Třídění výběrem – permutace pouze několika hodnot

Animace jsou v poznámkách k prezentaci vynechány.

## Třídění vkládáním – InsertSort

```
1  void InsertSort(int a[], const int n)
2  {
3     for (int i = 1; i < n; i++)
4     {
5        int v = a[i];
6        int j = i;
7        while (j > 0)
8        {
9           if (a[j - 1] > v)
10          {
11             a[j] = a[j - 1];
12             j -= 1;
13          }
14          else
15          {
16             break;
17          }
18       }
19       a[j] = v;
```



## Třídění vkládáním – InsertSort (pokrač.)

```
20    }
21 }
```



## Třídění vkládáním – náhodná permutace

Animace jsou v poznámkách k prezentaci vynechány.



## Třídění vkládáním – identická permutace

Animace jsou v poznámkách k prezentaci vynechány.

## Třídění vkládáním – opačná permutace

Animace jsou v poznámkách k prezentaci vynechány.



## Třídění vkládáním – permutace pouze několika hodnot

Animace jsou v poznámkách k prezentaci vynechány.



## Bublinové třídění – BubbleSort

- Vezmu dvojici sousedních prvků $a_i$ a $a_{i+1}$.
- Pokud tvoří inverzi, vyměním je.
- Pokračuji další dvojicí prvků $a_i$ a $a_{i+1}$, ale pro $i + 1$ atd.
- Výměnou dvojice prvků odstraním jednu inverzi.
- Proč bublinové třídění – největší prvek „se rozjede“, „probublá“ směrem ke konci pole.



## Bublinové třídění – BubbleSort

```
1 void BubbleSort0(int a[], const int n)
2 {
3    for (int i = 0; i < n; i++)
4    {
5       for (int j = 0; j < n - 1; j++)
6       {
7          if (a[j] > a[j + 1])
8          {
9             Exchange(a[j], a[j + 1]);
10         }
11      }
12   }
13 }
```

**Tak takhle ne!**

Tato implementace bublinového třídění je nejhorší možná! Takto by se bublinové třídění nemělo nikdy implementovat.

## Bublinové třídění – BubbleSort

```
1 void BubbleSort1(int a[], const int n)
2 {
3    for (int i = n - 1; i > 0; i--)
4    {
5       for (int j = 0; j < i; j++)
6       {
7          if (a[j] > a[j + 1])
8          {
9             Exchange(a[j], a[j + 1]);
10         }
11      }
12   }
13 }
```

- Po každém průchodu se dostane jeden prvek určitě na své místo – probublá nahoru.
- Není nutné jej proto dále zpracovávat.
- Tříděnou posloupnost je možné o tento prvek zkrátit.



## Bublinové třídění – BubbleSort

- Jiná možnost ukončení algoritmu – nedošlo k výměně prvků.
- Zavedeme si logickou proměnnou `AnyExchange`, kam si budeme ukládat informaci o případné výměně.
- Pokud algoritmus obdrží již setříděnou posloupnost – po jednom průchodu pozná, že je posloupnost již setříděná.



## Bublinové třídění – BubbleSort

```
1 void BubbleSort2(int a[], const int n)
2 {
3    bool AnyExchange;
4    do
5    {
6       AnyExchange = false;
7       for (int i = 0; i < n - 1; i++)
8       {
9          if (a[i] > a[i + 1])
10         {
11            Exchange(a[i], a[i + 1]);
12            AnyExchange = true;
13         }
14      }
15   } while (AnyExchange);
16 }
```



## Bublinové třídění – BubbleSort

- Kombinovaná možnost ukončení algoritmu:
  - nedošlo k výměně prvků a
  - zkracování posloupnosti o jeden prvek zprava.
- Zavedeme si logickou proměnnou `AnyExchange`, kam si budeme ukládat informaci o případné výměně.
- Proměnná `Right` bude označovat pravý okraj tříděné posloupnosti.

## Bublinové třídění

```
1 void BubbleSort3(int a[], const int n)
2 {
3    bool AnyExchange;
4    int Right = n - 1;
5    do
6    {
7       AnyExchange = false;
8       for (int i = 0; i < Right; i++)
9       {
10         if (a[i] > a[i + 1])
11         {
12            Exchange(a[i], a[i + 1]);
13            AnyExchange = true;
14         }
15      }
16      Right -= 1;
17   } while (AnyExchange);
18 }
```



## Bublinové třídění – BubbleSort

- Kombinovaná možnost ukončení algoritmu:
  - nedošlo k výměně prvků a
  - zkracování posloupnosti na index poslední výměny.
- Zavedeme si logickou proměnnou `AnyExchange`, kam si budeme ukládat informaci o případné výměně.
- Proměnná `LastExchangeIndex` bude označovat pravý okraj tříděné posloupnosti, index poslední výměny prvků.



## Bublinové třídění

```
1 void BubbleSort4(int a[], const int n)
2 {
3    int Right = n - 1;
4    int LastExchangeIndex;
5    do
6    {
7       LastExchangeIndex = 0;
8       for (int i = 0; i < Right; i++)
9       {
10         if (a[i] > a[i + 1])
11         {
12            Exchange(a[i], a[i + 1]);
13            LastExchangeIndex = i + 1;
14         }
15      }
16      Right = LastExchangeIndex;
17   } while (LastExchangeIndex > 0);
18 }
```



## Bublinové třídění – náhodná permutace

Animace jsou v poznámkách k prezentaci vynechány.

## Bublinové třídění – identická permutace

Animace jsou v poznámkách k prezentaci vynechány.



## Bublinové třídění – opačná permutace

Animace jsou v poznámkách k prezentaci vynechány.



## Bublinové třídění – permutace pouze několika hodnot

Animace jsou v poznámkách k prezentaci vynechány.



## Třídění přetřásáním

```
1  void ShakerSort(int a[], const int n)
2  {
3     int ExchangeIndex = 0;
4     int Left = 0;
5     int Right = n - 1;
6     do
7     {
8        for (int i = Left; i < Right; i++)
9        {
10          if (a[i] > a[i + 1])
11          {
12             Exchange(a[i], a[i + 1]);
13             ExchangeIndex = i;
14          }
15       }
16       Right = ExchangeIndex;
17       for (int i = Right; i > Left; i--)
18       {
19          if (a[i - 1] > a[i])
```

## Třídění přetřásáním (pokrač.)

```
20          {
21             Exchange(a[i - 1], a[i]);
22             ExchangeIndex = i;
23          }
24       }
25       Left = ExchangeIndex;
26    } while (Left < Right);
27 }
```



## Třídění přetřásáním – náhodná permutace

Animace jsou v poznámkách k prezentaci vynechány.



## Třídění přetřásáním – identická permutace

Animace jsou v poznámkách k prezentaci vynechány.



## Třídění přetřásáním – opačná permutace

Animace jsou v poznámkách k prezentaci vynechány.

## Třídění přetřásáním – permutace pouze několika hodnot

Animace jsou v poznámkách k prezentaci vynechány.



## Děkuji za pozornost



# Pokročilé třídící algoritmy

doc. Mgr. Jiří Dvorský, Ph.D.

Katedra informatiky
Fakulta elektrotechniky a informatiky
VŠB – TU Ostrava





## QuickSort – Třídění rozdělováním

- C. A. R. Hoare, 1962.
- využití metody „rozděl a panuj“ (divide et impera),
- rekurzivní algoritmus,
- výměny na velkou vzdálenost,
- v každém kroku rozdělení tříděného úseku pole na dvě disjunktní části,
- všechny prvky v levé části jsou menší než všechny prvky v pravé částí,
- rozdělení pomocí prvku pole zvaného **pivot**,
- levá a pravá část pole je tříděna pomocí rekurze stejným způsobem.

## QuickSort, výběr pivota

- pivotem může být libovolný prvek ze tříděného úseku pole,
- tříděný úsek pole se prochází:
  1. zleva a hledá se prvek větší než pivot,
  2. zprava a hledá se prvek menší než pivot,
  3. takto nesprávně zařazené prvky se vzájemně prohodí.
- až se indexy kterými pole zleva a zprava „potkají“ je tříděný úsek pole rozdělen na dvě části
- algoritmus funguje vždy, **ale**
- výběr pivota silně ovlivňuje složitost algoritmu!



## QuickSort, výběr pivota

- ideální případ – pivot rozdělí tříděný úsek pole na dvě přibližně stejné části – **medián**,
- nejhorší případ – pivot rozdělí tříděný úsek pole na část s jedním prvkem a zbytkem – **minimum** či **maximum**,
- další možnosti:
  - prvek ve středu tříděného úseku,
  - medián ze tří prvků tříděného úseku,
  - první prvek,
  - libovolný prvek.

**Složitost algoritmu QuickSort**

- průměrná časová složitost $O(n \log_2 n)$
- nejhorší časová složitost $O(n^2)$!



## QucikSort – Třídění rozdělováním

```
1 void QuickSortInternal(int a[], const int l, const int r)
2 {
3    int i = l;
4    int j = r;
5    int pivot = a[(l + r) / 2];
6    do
7    {
8       while (a[i] < pivot)
9          i += 1;
10      while (pivot < a[j])
11         j -= 1;
12      if (i <= j)
13      {
14         Exchange(a[i], a[j]);
15         i += 1;
16         j -= 1;
17      }
18   } while (i <= j);
19   if (l < j)
```



## QucikSort – Třídění rozdělováním (pokrač.)

```
20       QuickSortInternal(a, l, j);
21    if (i < r)
22       QuickSortInternal(a, i, r);
23 }
```

## QucikSort – Třídění rozdělováním

```
1 void QuickSort(int a[], const int n)
2 {
3   QuickSortInternal(a, 0, n - 1);
4 }
```



## QuickSort – náhodná permutace, pivot prostřední prvek úseku

Animace jsou v poznámkách k prezentaci vynechány.



## QuickSort – náhodná permutace, pivot medián úseku

Animace jsou v poznámkách k prezentaci vynechány.



## QuickSort – náhodná permutace, pivot minimum úseku

Animace jsou v poznámkách k prezentaci vynechány.

## QuickSort – náhodná permutace, pivot levý prvek úseku

Animace jsou v poznámkách k prezentaci vynechány.



## QuickSort – náhodná permutace, pivot náhodný z prvek úseku

Animace jsou v poznámkách k prezentaci vynechány.



## QuickSort – identická permutace, pivot prostřední prvek úseku

Animace jsou v poznámkách k prezentaci vynechány.



## QuickSort – opačná permutace, pivot prostřední prvek úseku

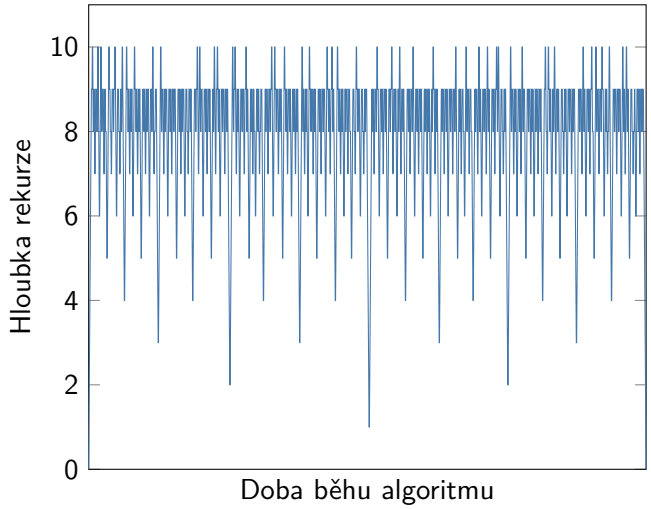
Animace jsou v poznámkách k prezentaci vynechány.

## QuickSort – permutace pouze několika hodnot

Animace jsou v poznámkách k prezentaci vynechány.

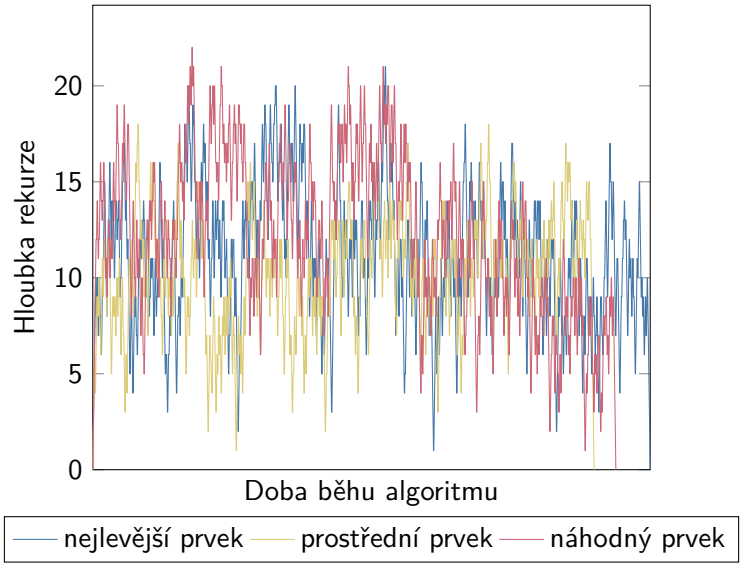


## Hloubka rekurze v závislosti na volbě pivota – nejlepší případ, medián



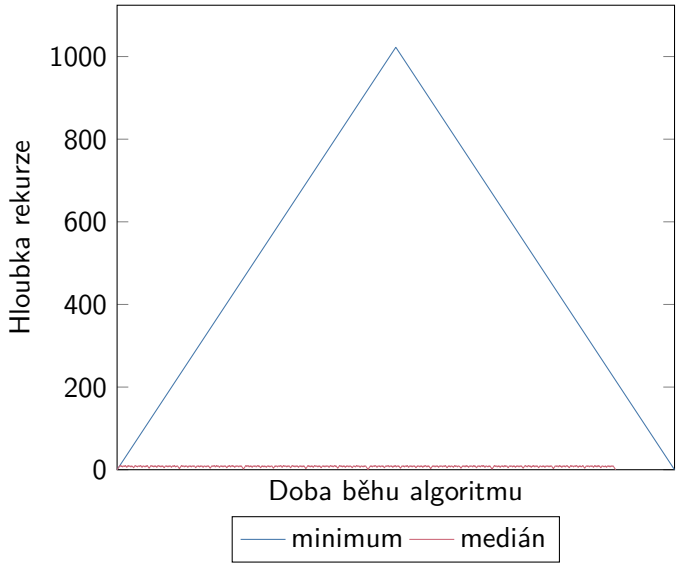



## Hloubka rekurze v závislosti na volbě pivota

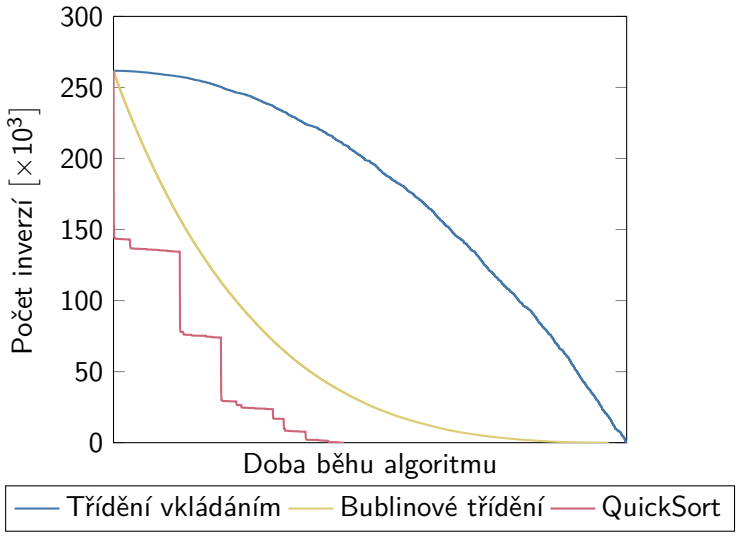





## Hloubka rekurze v závislosti na volbě pivota – extrémy

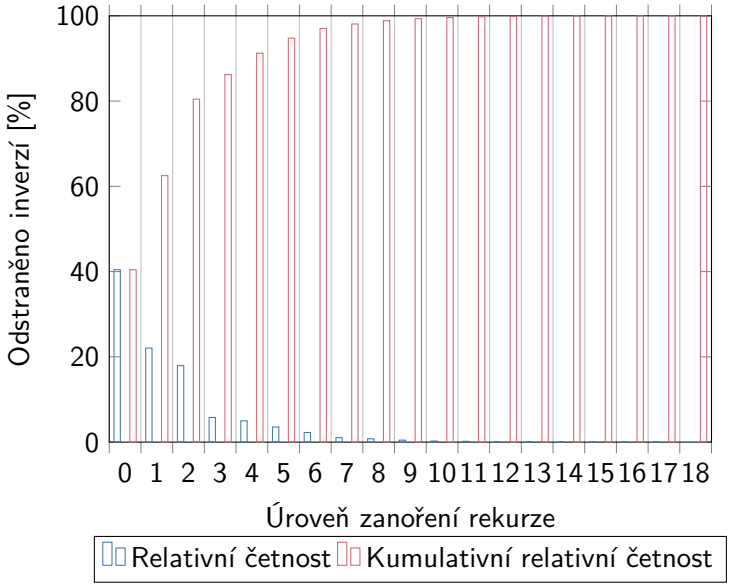


## Průběh odstraňování inverzí v čase






## QuickSort – průběh odstraňování inverzí






## Třídění haldou – HeapSort

- lze říci, že zlepšuje třídění výběrem,
- efektivně řeší problém výběru $i$-tého největšího prvku,
- pro $i = 1$ je jasné, že potřebujeme $n - 1$ porovnání,
- ale pro další $i$ by bylo vhodné si nějakým způsobem „zapamatovat“ výsledky předchozích porovnání,
- tomuto účelu nejlépe vyhovuje datová struktura **halda** (angl. heap).

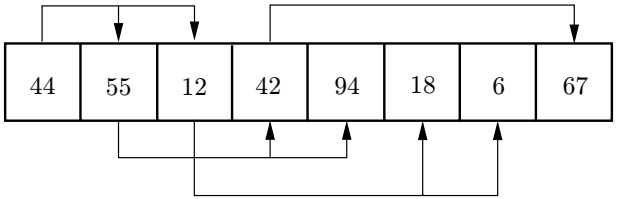


## Halda (Heap)

Halda reprezentující posloupnost $S$ délky $n$, je úplný binární strom výšky $h \geq 1$ s $n$ vrcholy, a následujícími vlastnostmi:

- všechny listy se nachází ve vzdálenosti $h$ nebo $h - 1$ od kořene,
- všechny listy na úrovni $h$ jsou vlevo od listů na úrovni $h - 1$,
- každému vrcholu tohoto stromu je přiřazen jeden prvek posloupnosti $S$ tak, že všem jeho potomkům jsou přiřazeny menší prvky.

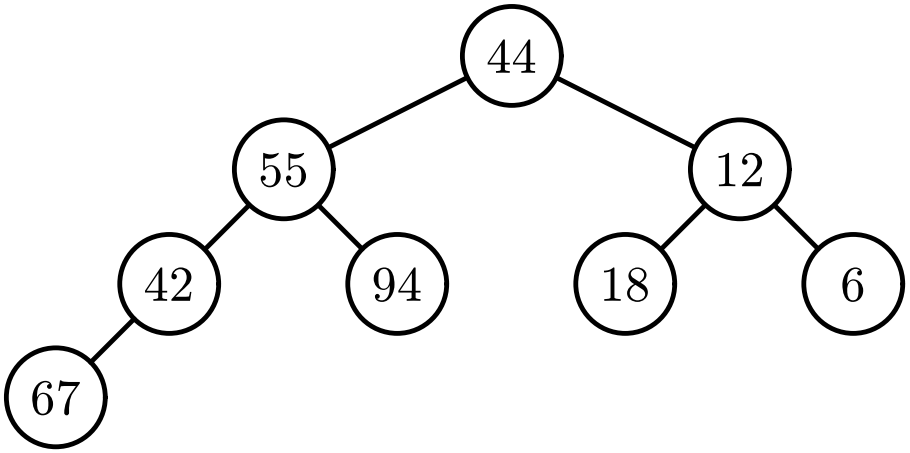
## Konstrukce haldy – počáteční stav




Haldu lze efektivně reprezentovat v poli tak, že leží-li rodič na indexu $i$, pak jeho levý potomek leží na indexu $2i$ a pravý potomek na indexu $2i + 1$.

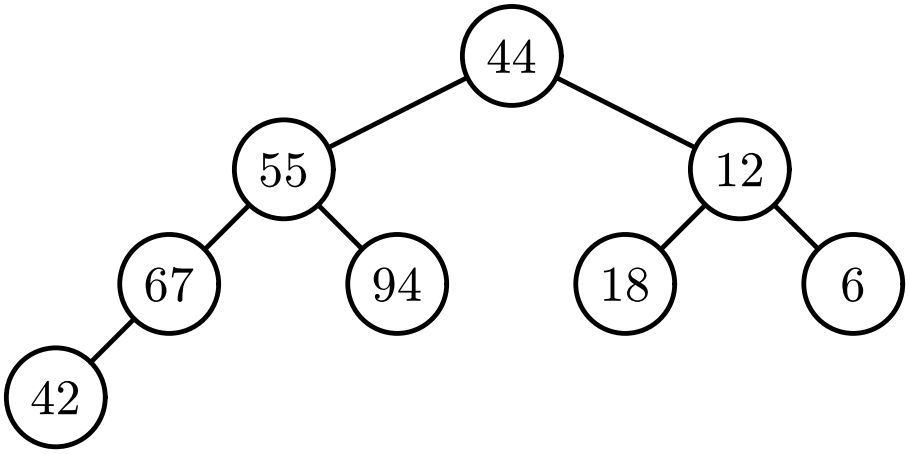


## Původní halda



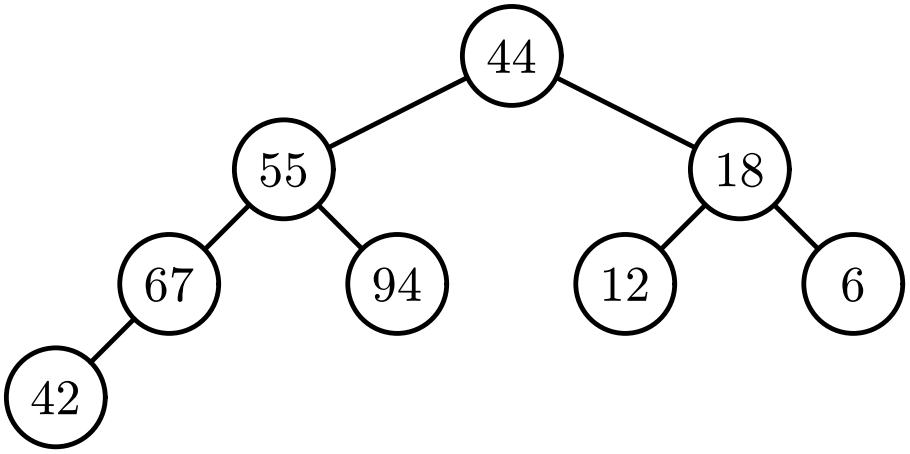



## Přehozeno 42 a 67

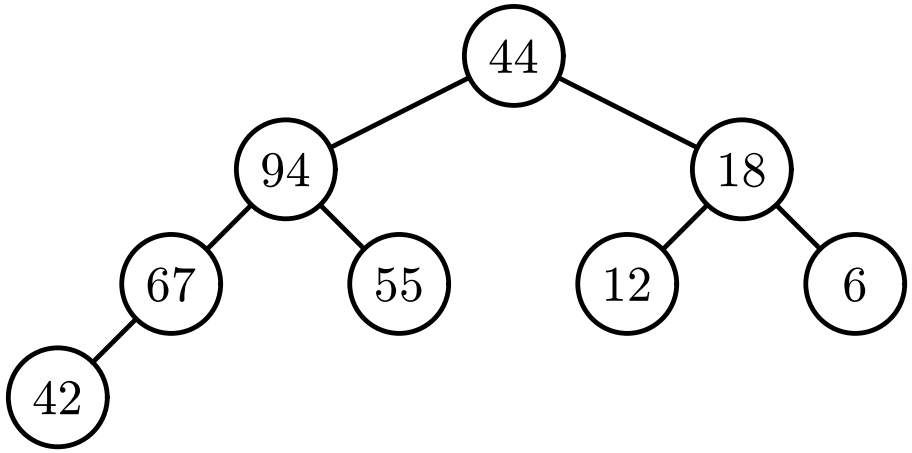





## Přehozeno 18 s 12

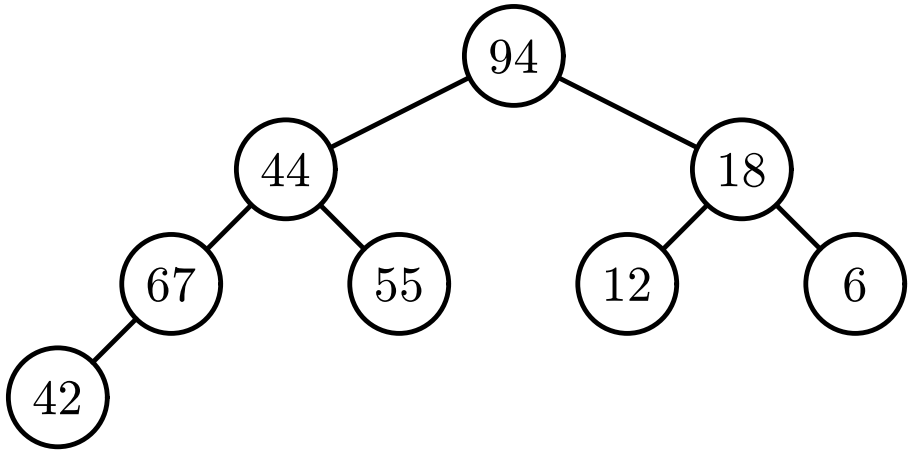


## Přehozeno 55 a 94



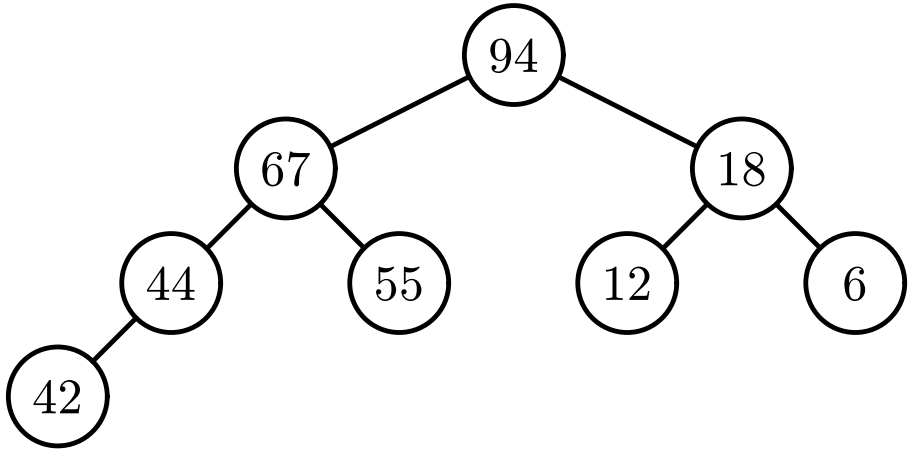



## Přehozeno 94 a 44





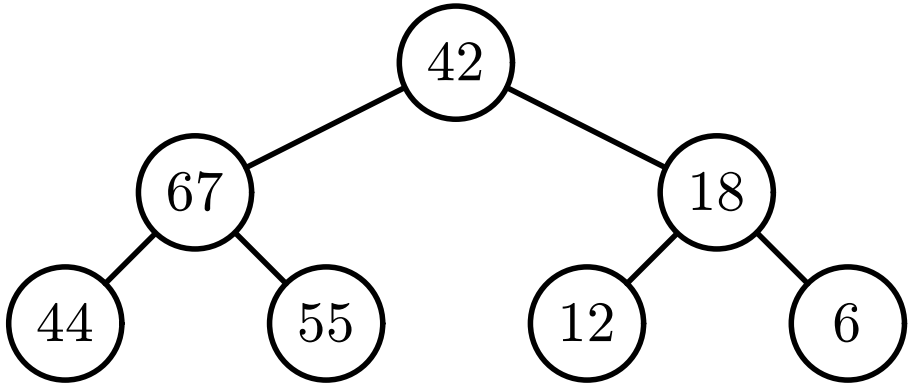

## Přehozeno 44 a 67



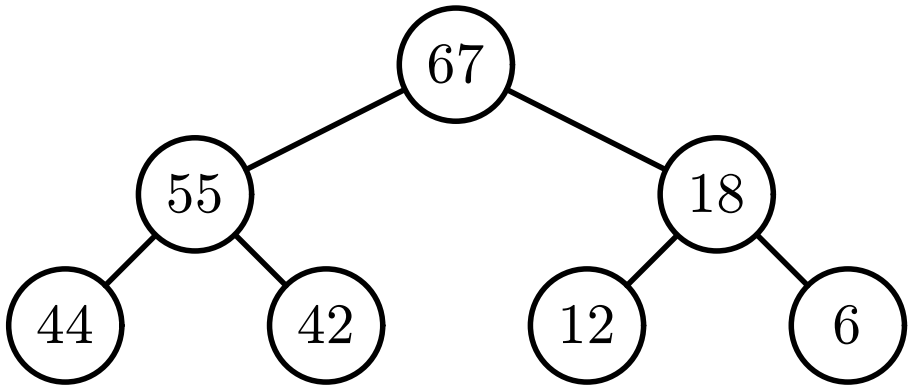

Tím je halda hotová!



## Odebrání 94

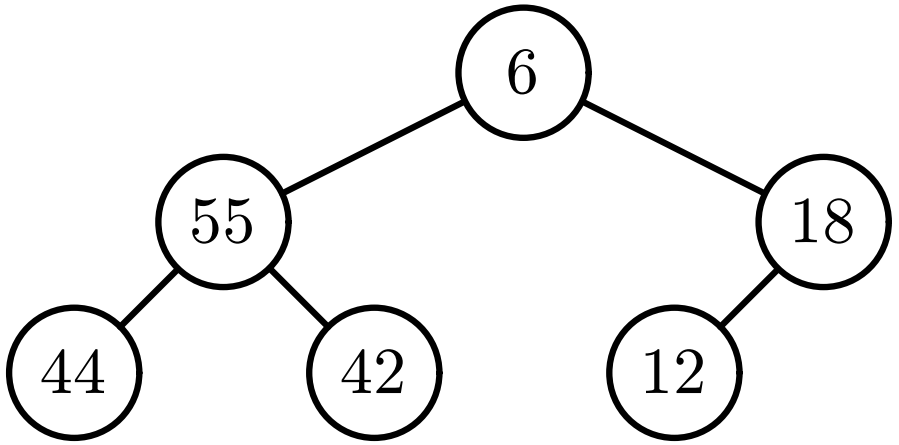


## Oprava haldy



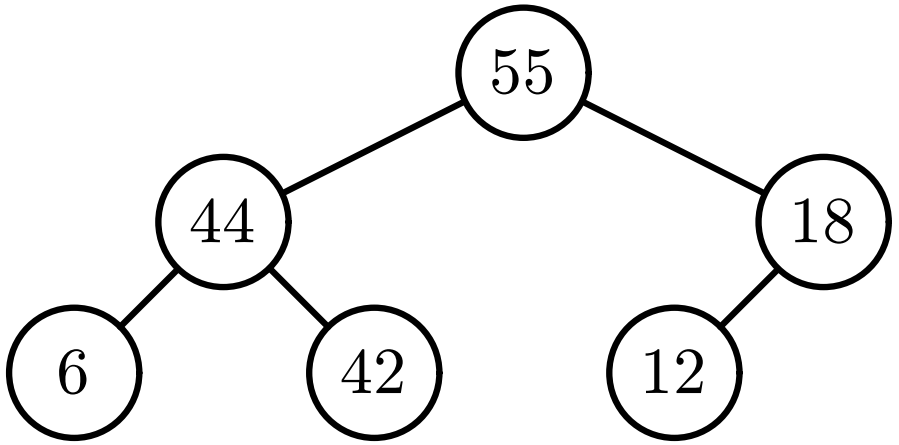



## Odebrání 67



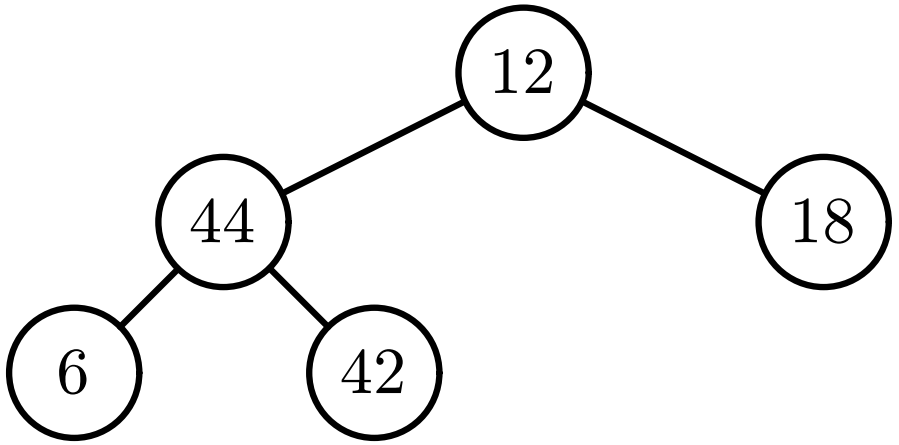



## Oprava haldy

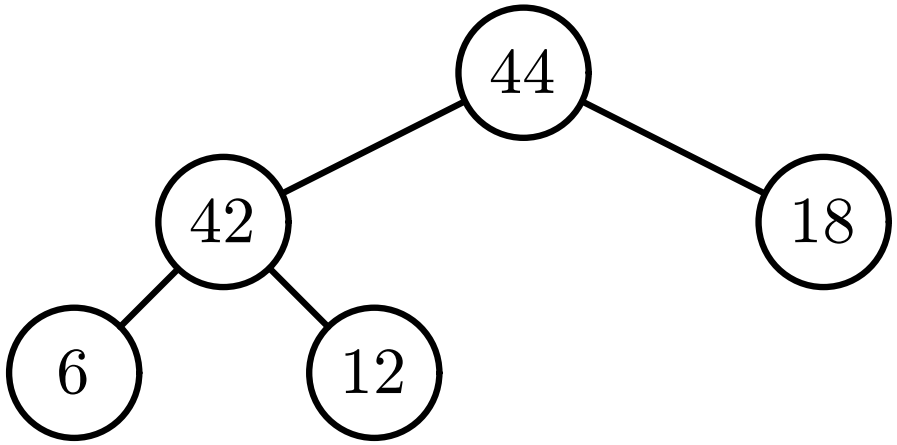





## Odebrání 55

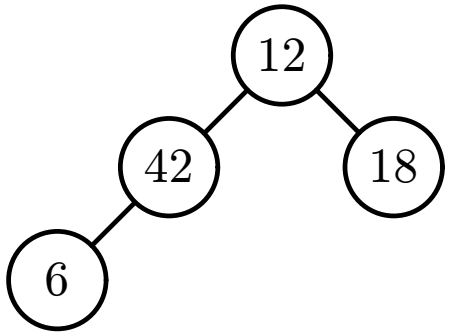


## Oprava haldy



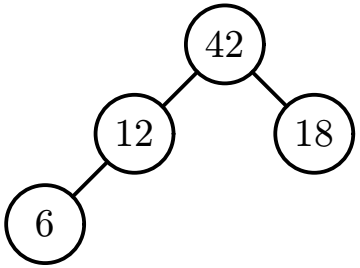



## Odebrání 44



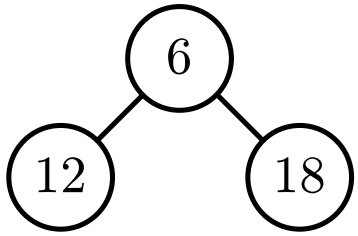



## Oprava haldy

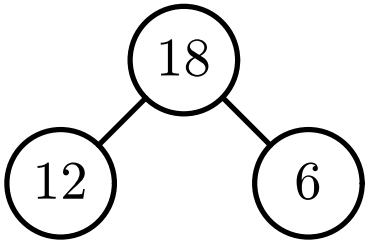





## Odebrání 42

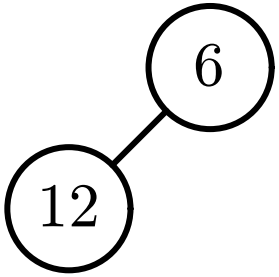


## Oprava haldy



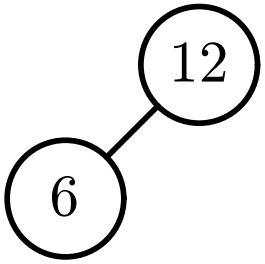



## Odebrání 18






## Oprava haldy

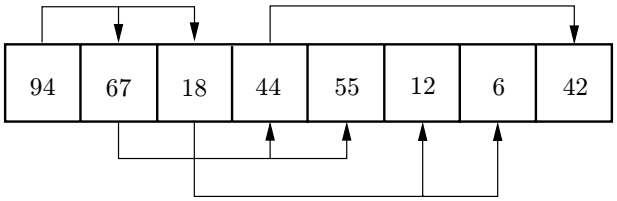





## Koncový stav haldy



## Halda reprezentovaná v poli






## Třídění haldou – HeapSort

```
1 void BuildHeap(int a[], int n, int i)
2 {
3    int largest = i;
4    int left = 2 * i + 1;
5    int right = 2 * i + 2;
6    if (left < n && a[left] > a[largest])
7       largest = left;
8    if (right < n && a[right] > a[largest])
9       largest = right;
10   if (largest != i)
11   {
12      Exchange(a[i], a[largest]);
13      BuildHeap(a, n, largest);
14   }
15 }
```



## Třídění haldou – HeapSort

```
1 void HeapSort(int a[], const int n)
2 {
3    for (int i = n / 2 - 1; i >= 0; i--)
4       BuildHeap(a, n, i);
5    for (int i = n - 1; i >= 0; i--)
6    {
7       Exchange(a[0], a[i]);
8       BuildHeap(a, i, 0);
9    }
10 }
```



## Třídění haldou – náhodná permutace

Animace jsou v poznámkách k prezentaci vynechány.

## Třídění haldou – identická permutace

Animace jsou v poznámkách k prezentaci vynechány.



## Třídění haldou – opačná permutace

Animace jsou v poznámkách k prezentaci vynechány.

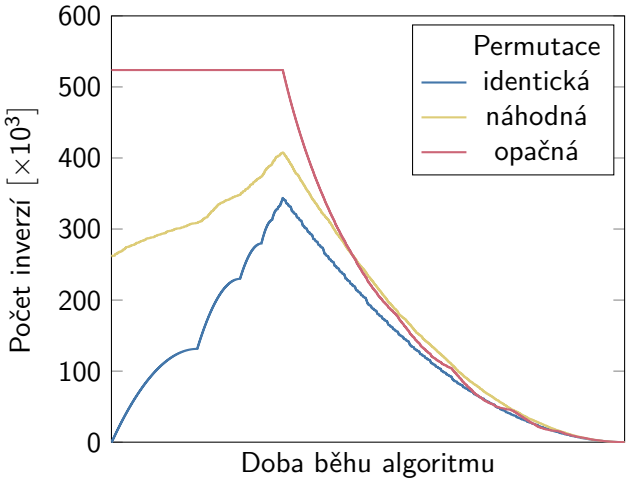


## Třídění haldou – permutace pouze několika hodnot

Animace jsou v poznámkách k prezentaci vynechány.



## Průběh odstraňování inverzí v čase – třídění haldou



## Děkuji za pozornost



# Hledání nejkratší trasy aneb Sex, chlast a C++

doc. Mgr. Jiří Dvorský, Ph.D.

Katedra informatiky
Fakulta elektrotechniky a informatiky
VŠB – TU Ostrava





## Motivační přednáška

**Upozornění**

1. Tato přednáška je pouze motivační. Má za cíl demonstrovat, co lze se znalostmi programování a algoritmů po prvním semestru zvládnout.
2. Náplň této přednášky **nebude** součástí žádného testu či písemné práce.



## Život studentský, život veselý...

- Rodiče daleko...
- Plnoletost...
- Kapesné či vlastní příjem...
- Minimální povinnosti...
- Mladý organismus schopný rychlé regenerace...
- Tak proč nejít do hospody?!
- Důvod se vždycky najde...
- A když nenajde, tak se musí vymyslet...

## Tour de pub

- klasická kolejní zábava,
- závod spočívá v návštěvě předem daných hospod,
- závodníci se přesunují výhradně pěšky,
- v každé hospodě je nutno vypít pivo nebo/a, panáka,
- závodí buď jednotlivci nebo týmy (většinou smíšené),
- trasa nemusí být přesně stanovena – je možné volit pořadí navštívených hospod,
- vítězí ten, kdo dorazí jako první do cíle s potvzenou konzumací.



## Tour de pub Olomouc 2017

- předpokládejme, že se zúčastníme Tour de pub v Olomouci,
- máme daný seznam soutěžních hospod,
- v každé hospodě si dá závodník jedno pivo,
- start a cíl je v hospodě Doga,
- závodníci se přesunují výhradně pěšky, tramvaje či taxíky jsou zakázány,
- pořadí hospod je na každém závodníkovi.

**Zadání úlohy**

Navrhněte optimální trasu pro tento závod.



## Soutěžní hospody

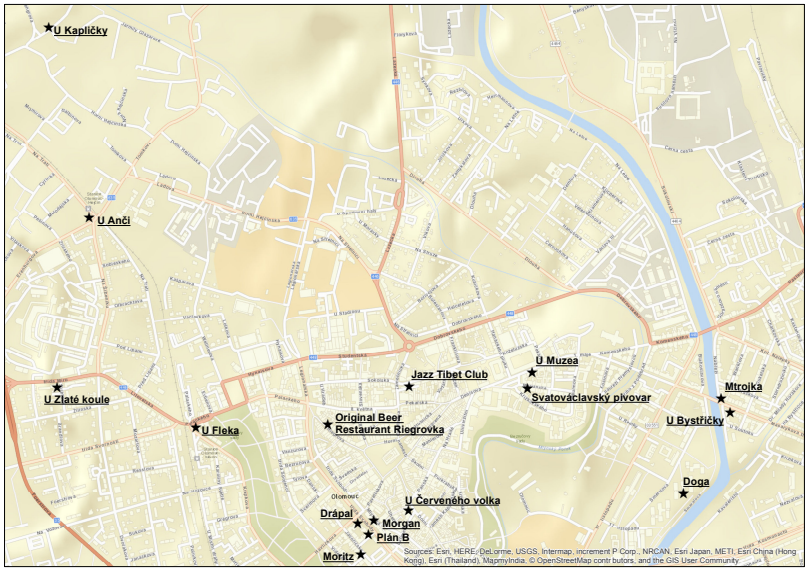
| Id | Restaurace | Adresa, web |
|---|---|---|
| 0 | **Drápal** | Havlíčkova 1, 779 00 Olomouc<br>`www.restauracedrapal.cz/` |
| 1 | **Plán B** | Palachovo nám. 620/1, 779 00 Olomouc<br>`www.bar-planb.cz/` |
| 2 | **Pivovar Moritz** | Nešverova 683/2, 779 00 Olomouc<br>`www.hostinec-moritz.cz/` |
| 3 | **U červeného volka** | Dolní nám. 26/39, 779 00 Olomouc<br>`www.ucervenehovolka.cz/` |
| 4 | **U zlaté koule** | tř. Míru 168/15, 779 00 Olomouc<br>`www.uzlatekoule.cz/` |
| 5 | **Mtrojka** | Masarykova tř. 889/3, 779 00 Olomouc<br>`www.m3olomouc.cz/` |
| 6 | **U Musea** | 1.máje 827/8, 779 00 Olomouc<br>`www.ponorka.com/` |
| 7 | **U Bystřičky** | Masarykova tř. 802/ 10, 779 00 Olomouc |



## Soutěžní hospody (pokrač.)

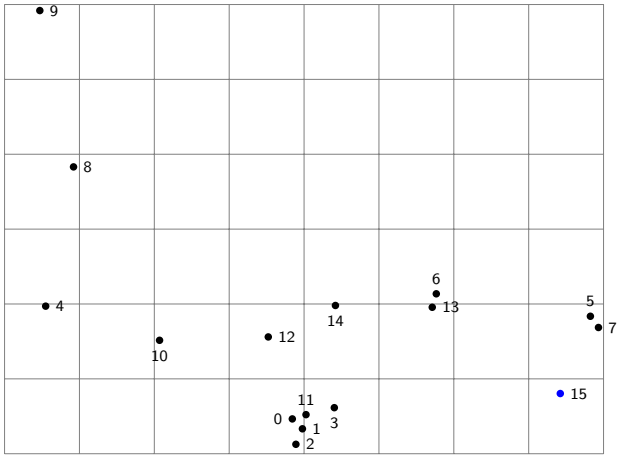
| Id | Restaurace | Adresa, web |
|---|---|---|
| 8 | **U Anči** | Na Trati 213/59, 779 00 Olomouc |
| 9 | **U kapličky** | Břetislavova 29/3, 779 00 Olomouc |
| 10 | **U Fleka** | Palackého 132/25, 779 00 Olomouc<br>`www.ufleka.cz/` |
| 11 | **Morgan** | Mlýnská 2, 779 00 Olomouc<br>`morgansrestaurant.cz/` |
| 12 | **Riegrovka** | Riegrova 22, 779 00 Olomouc<br>`riegrovka.eu/` |
| 13 | **Svatováclavský pivovar** | Mariánská 845/4, 779 00 Olomouc<br>`www.svatovaclavsky-pivovar.cz/` |
| 14 | **Jazz Tibet club** | Sokolská 48, Olomouc<br>`www.jazzclub.olomouc.com/` |
| 15 | **Doga** | Šmeralova 1116/10, 779 00 Olomouc<br>`www.pivnicedoga.cz/` |

## Mapa soutěžních hospod






## Schematická mapa soutěžních hospod






## Tour de pub Olomouc 2017 – analýza zadání

- Máme navrhnout optimální trasu pro závod. Co je to **optimální trasa**?
- Předpokládejme, že
  1. průměrná doba vypití jednoho piva je zhruba konstantní a
  2. průměrná rychlost chůze závodníka je také zhruba konstantní.
- Soutěžních piv je daný počet, tuto dobu asi příliš neovlivníme.
- Rychlost chůze asi nejsme schopni nijak výrazně zvýšit, ona se bude spíše snižovat. . .
- Můžeme ale volit pořadí navštívených hospod.
- Asi nebude vhodné pobíhat z jednoho konce města na druhý, ale brát hospody „nějak postupně“.
- Budeme-li brát hospody „nějak postupně“ ujdeme pravděpodobně menší vzdálenost, než když budeme chaoticky pobíhat po městě.



## Tour de pub Olomouc 2017 – analýza zadání (pokrač.)

- Jinak řečeno, čím kratší trasu zvolíme, tím rychleji se nám podaří závod dokončit.
- Formálněji řečeno, hledáme nejkratší trasu ze startovní hospody přes všechny soutěžní hospody zpět do cílové hospody.

### Nástroje pro hledání nejkratší soutěžní trasy

Musíme vymyslet následující:

- způsob, jak reprezentovat trasu,
- způsob, jak určit délku libovolné trasy a
- algoritmus řešení úlohy.

## Formální specifikace zadání I.

- Počet soutěžních hospod $N$, v našem případě $N = 16$.
- Hospody budou očíslovány $0, \ldots, N-1$.
- Startovní a cílová hospoda má číslo $N-1$, v našem případě 15.
- Volit pořadí můžeme jen u hospod $0, \ldots, N-2$, v našem případě $0, \ldots, 14$.



## Reprezentace trasy

- Co je pro reprezentaci nějaké trasy rozhodující? Čím se jednotlivé trasy od sebe liší?
- Startovní a cílová hospoda je neměnná.
- Délka? Ne! Délka trasy je jen důsledkem toho jak jsou hospody na trase uspořádány – jestli budeme pobíhat po městě nebo ne.
- Takže záleží na pořadí hospod na trase.
- Jak matematicky reprezentovat různé možnosti „přeházení“ či „přeuspořádání“ nějaké množiny? Konečné množiny čísel $0, \ldots, 14$?
- No přece pomocí **permutace**!
- Permutace jednoznačně přiřazuje jednomu přirozenému číslu jiné přirozené číslo. Formálně jde o zobrazení $\pi : \mathbb{N} \to \mathbb{N}$.
- V našem případě, permutace bude číslu označujícímu pořadí hospody na trase přiřazovat identifikační číslo hospody. Například takto:

$$\pi = \begin{pmatrix} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 \\ 6 & 14 & 13 & 9 & 12 & 4 & 3 & 1 & 0 & 11 & 2 & 10 & 8 & 5 & 7 \end{pmatrix}$$



## Tour de pub Olomouc 2017 – analýza zadání

**Nástroje pro hledání nejkratší soutěžní trasy**

Musíme vymyslet následující:

- způsob, jak reprezentovat trasu, ✓
- způsob, jak určit délku libovolné trasy a
- algoritmus řešení úlohy.



## Výpočet délky trasy

- Algoritmus výpočtu délky trasy bude přiřazovat každé trase nějaké kladné, patrně reálné, číslo.
- Z matematického pohledu se tedy bude jednat o funkci, nazvěme ji $d$, které každé permutaci $\pi$, reprezentující libovolnou trasu, přiřadí číslo z množiny $\mathbb{R}$.
- Je zřejmé, že délku celé trasy vypočteme jako součet jednotlivých úseků „od hospody k hospodě“.
- Budeme tedy potřebovat tabulku vzdáleností od hospody $i$ k hospodě $j$.
- Jak reprezentovat matematicky tabulku? Budeme potřebovat něco, co má dva indexy – řádek a sloupec.
- Řešení? **Matice**! Matice $D$ jednoznačně přiřazuje uspořádané dvojici přirozených čísel $(i, j)$ jiné, většinou reálné, číslo $D_{i,j}$. Formálně jde opět o zobrazení, $D : \mathbb{N} \times \mathbb{N} \to \mathbb{R}$.

## Výpočet délky trasy (pokrač.)

- Matici $D$ můžeme znázornit obvyklým způsobem tj. pomocí mřížky čísel ohraničených závorkami.

$$D = \begin{pmatrix}
0 & 100 & 160 & 320 & 1400 & 1600 & 890 & 1700 & 1800 & 2600 & 820 & 110 & 480 & 900 & 650 & 1600 \\
100 & 0 & 80 & 280 & 1500 & 1700 & 970 & 1700 & 1900 & 2700 & 890 & 190 & 560 & 980 & 730 & 1500 \\
160 & 80 & 0 & 310 & 1500 & 1800 & 1000 & 1800 & 1900 & 2800 & 900 & 270 & 640 & 1100 & 810 & 1500 \\
320 & 280 & 310 & 0 & 1700 & 1500 & 760 & 1500 & 2000 & 2700 & 1100 & 210 & 550 & 780 & 600 & 1300 \\
1400 & 1500 & 1500 & 1700 & 0 & 2600 & 1900 & 2600 & 780 & 1700 & 580 & 1500 & 1100 & 1900 & 1500 & 3100 \\
1600 & 1700 & 1800 & 1500 & 2600 & 0 & 750 & 60 & 2800 & 3200 & 2100 & 1600 & 1600 & 840 & 1300 & 1000 \\
890 & 970 & 1000 & 760 & 1900 & 750 & 0 & 770 & 2000 & 2600 & 1400 & 870 & 860 & 150 & 510 & 1900 \\
1700 & 1700 & 1800 & 1500 & 2600 & 60 & 770 & 0 & 2800 & 3200 & 2100 & 1600 & 1600 & 860 & 1300 & 1000 \\
1800 & 1900 & 1900 & 2000 & 780 & 2800 & 2000 & 2800 & 0 & 950 & 1000 & 1900 & 1400 & 2100 & 1700 & 3500 \\
2600 & 2700 & 2800 & 2700 & 1700 & 3200 & 2600 & 3200 & 950 & 0 & 2000 & 2600 & 2200 & 2600 & 2300 & 3900 \\
820 & 890 & 900 & 1100 & 580 & 2100 & 1400 & 2100 & 1000 & 2000 & 0 & 880 & 620 & 1400 & 980 & 2300 \\
110 & 190 & 270 & 210 & 1500 & 1600 & 870 & 1600 & 1900 & 2600 & 880 & 0 & 460 & 880 & 630 & 1400 \\
480 & 560 & 640 & 550 & 1100 & 1600 & 860 & 1600 & 1400 & 2200 & 620 & 460 & 0 & 880 & 570 & 1700 \\
900 & 980 & 1100 & 780 & 1900 & 840 & 150 & 860 & 2100 & 2600 & 1400 & 880 & 880 & 0 & 540 & 1500 \\
650 & 730 & 810 & 600 & 1500 & 1300 & 510 & 1300 & 1700 & 2300 & 980 & 630 & 570 & 540 & 0 & 1700 \\
1600 & 1500 & 1500 & 1300 & 3100 & 1000 & 1900 & 1000 & 3500 & 3900 & 2300 & 1400 & 1700 & 1500 & 1700 & 0
\end{pmatrix}$$

- Čtvercová matice $D$ řádu $N$ bude tedy představovat matici vzdáleností jednotlivých hospod.



## Výpočet délky trasy (pokrač.)

- Prvek matice $D_{i,j}$ představuje vzdálenost mezi $i$-tou a $j$-tou soutěžní hospodou.
- Funkci $d$ můžeme s pomocí matice $D$ zapsat takto:

$$d(\pi) = D_{N-1,\pi(0)} + \sum_{i=0}^{N-3} D_{\pi(i),\pi(i+1)} + D_{\pi(N-2),N-1},$$

kde

- $D_{N-1,\pi(0)}$ je délka úseku ze startovací hospody do první hospody na trase,
- $\sum_{i=0}^{N-3} D_{\pi(i),\pi(i+1)}$ je součet všech úseků mezi hospodami na trase a
- $D_{\pi(N-2),N-1}$ je délka úseku z poslední hospody na trase do cílové hospody.



## Tour de pub Olomouc 2017 – analýza zadání

**Nástroje pro hledání nejkratší soutěžní trasy**

Musíme vymyslet následující:

- způsob, jak reprezentovat trasu, ✓
- způsob, jak určit délku libovolné trasy a ✓
- algoritmus řešení úlohy.



## Metoda řešení – sex

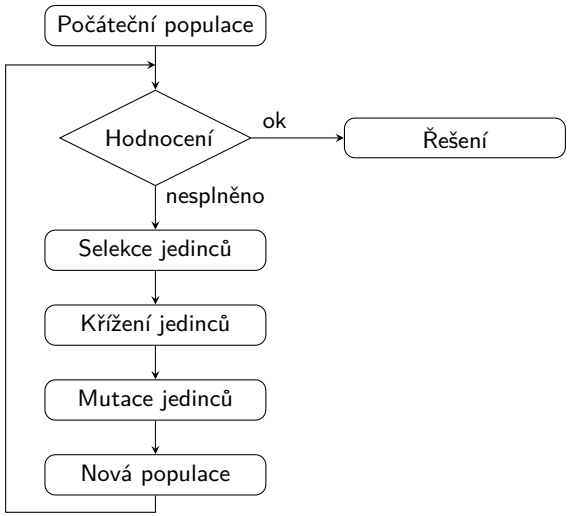
- Přímočaré řešení hrubou silou nemá šanci na úspěch, permutací je příliš mnoho, $n!$. Výsledku bychom se nemuseli dočkat.
- Použít nějakou soustavu rovnic? Jakou? Co by mělo pro pořadí hospod platit?
- Umíme v podstatě jen měřit délku trasy a říci, které ze dvou tras je ta kratší.
- Nešlo by nějak začít u „normálního“ nezajímavého řešení a nějak ho vylepšovat?
- Nešlo by řešení nějak šlechtit?
- Když to funguje v přírodě, proč by to nemohlo fungovat v počítači.
- Řešením je použít darwinovskou evoluci.

## Metoda řešení – evoluční algoritmy

- Evoluční algoritmy se řadí mezi biologicky inspirované algoritmy.
- Zde použitý **genetický algoritmus** je jedním z mnoha evolučním algoritmů.
- Základem je jedinec, jehož genetická informace představuje řešení problému.
- Každý jedinec je ohodnocen svou zdatností – fitness. Snažíme se fitness maximalizovat.
- Algoritmus pracuje s populací jedinců, které přežívá s generace na generaci.
- Tři základní mechanismy pro vývoj populace:
  - Selekce – jedinci jsou vybráni pro křížení na základě své fitness, lepší jedinci mají přednost.
  - Křížení – kombinace genetické informace dvou jedinců.
  - Mutace – malá náhodná změna u nového jedince.
- Do nové generace postupují jen ti nejzdatnější jedinci.
- Algoritmus tedy napodobuje šlechtění rostlin, živočichů atd.



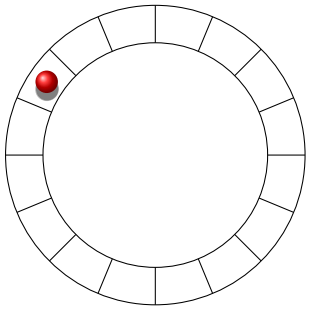
## Genetický algoritmus – schéma






## Selekce

- Jak provést selekci jedinců?
- Náhodně? Zcela náhodně? Prostě ruleta?



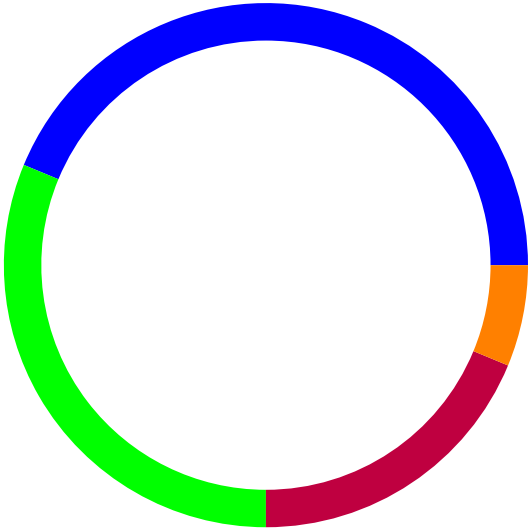
- Asi ne – se potřeba podporovat jedince s vysokým fitness, kvalitnější rodiče budou mít pravděpodobně kvalitní potomstvo.
- Vyřadit ale méně kvalitní jednice úplně z rozmnožování? Můžou nést malý kousek cenné genetické informace.



## Selekce (pokrač.)

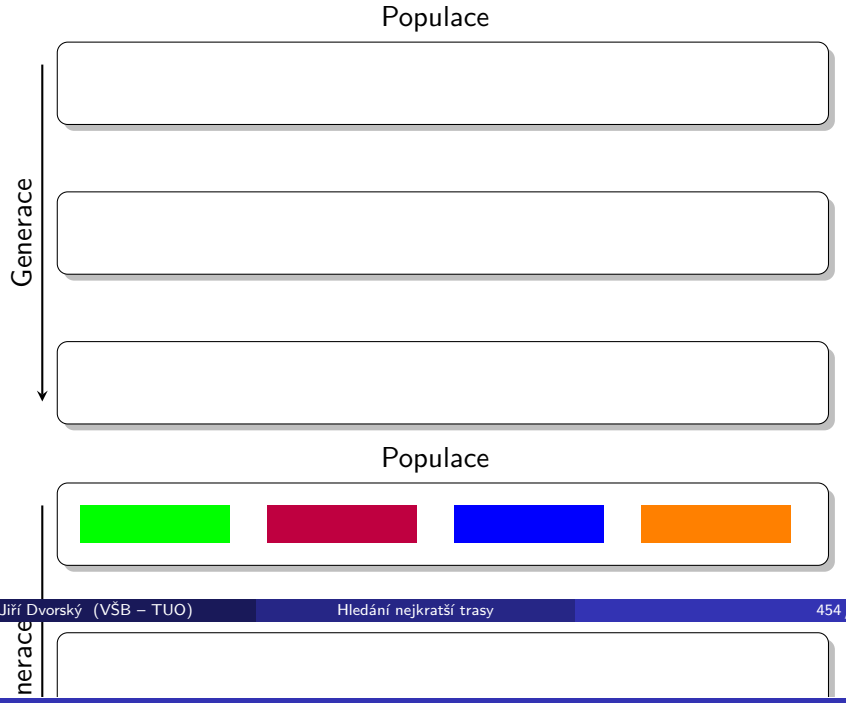
- Jen s kvalitními jedinci nám hrozí uváznutí na jednom místě, všechna řešení si budou podobná, nebudeme schopni provést větší změnu v řešení.
- Řešením se ukazuje ruletové kolo s různě velkými výsečemi. Velikost výšeče odpovídá fitness jedince.

## Selekce (pokrač.)





## Křížení jedinců






## Mutace jednice






## Jak reprezentovat jedince a potřebné operace?

- Trasa po hospodách je dána permutací – jedinec tedy bude reprezentován touto permutací.
- Zdatnost jedince bude dána délkou této trasy. Otázka maximalizace fitness.
- Jak provést křížení permutací? A jak provést mutaci v permutaci?
- Mutace by šla řešit prohozením dvojice čísel v permutaci.
- Při přímočaré reprezentaci může dojít při křížení k opakování hodnot v permutaci.
- Řešení nabízí reprezentace pomocí tzv. náhodných klíčů.
  - Pro každý prvek v permutaci vygenerujeme náhodné číslo v rozsahu $\langle 0, 1)$.

$$\begin{pmatrix} 0,7155 & 0,2547 & 0,8695 & 0,4034 & 0,7228 & 0,7678 \\ 0 & 1 & 2 & 3 & 4 & 5 \end{pmatrix}$$

## Jak reprezentovat jedince a potřebné operace? (pokrač.)

- Permutaci získáme jejich setříděním.

$$\begin{pmatrix} 0,2547 & 0,4034 & 0,7155 & 0,7228 & 0,7678 & 0,8695 \\ 1 & 3 & 0 & 4 & 5 & 2 \end{pmatrix}$$

- Křížení – spojíme dvě části posloupnosti náhodných klíčů a klíče setřídíme.
- Mutace je přímočará – nepatrně změníme jeden k klíčů a setřídíme.



## Fitness jedince

- GA maximalizuje fitness.
- My chceme ale naopak co nejkratší trasu!
- Jak tedy převést mimimalizaci délky trasy na maximalizaci?
- Vhodnou formulací fitness funkce.
- Musíme použít přímo délku trasy? Nemůžeme ji nějak transformovat?
- Velice jednoduše – převrácené číslo, „jedna lomeno“.
- Čím kratší trasa bude, tím bude její převrácená hodnota větší.



## Tour de pub Olomouc 2017 – analýza zadání

**Nástroje pro hledání nejkratší soutěžní trasy**

Musíme vymyslet následující:

- způsob, jak reprezentovat trasu, ✓
- způsob, jak určit délku libovolné trasy a ✓
- algoritmus řešení úlohy. ✓



## Soubor GlobalConstants.h

```
1  #pragma once
2
3  const int TotalNumberOfPubs = 16;
4
5  const int NumberOfMutablePubs = TotalNumberOfPubs - 1;
6
7  const int StartFinishPubIndex = TotalNumberOfPubs - 1;
8
9  const int DistanceMatrix[TotalNumberOfPubs][TotalNumberOfPubs] =
10 {
11    {0, 100, 160, 320, 1400, 1600, 890, 1700, 1800, 2600, 820, 110, 480,
          900, 650, 1600},
12    { 100, 0, 80, 280, 1500, 1700, 970, 1700, 1900, 2700, 890, 190, 560,
          980, 730, 1500 },
13    { 160, 80, 0, 310, 1500, 1800, 1000, 1800, 1900, 2800, 900, 270,
        640, 1100, 810, 1500 },
14    { 320, 280, 310, 0, 1700, 1500, 760, 1500, 2000, 2700, 1100, 210,
        550, 780, 600, 1300 },
```

## Soubor GlobalConstants.h (pokrač.)

```
15    { 1400, 1500, 1500, 1700, 0, 2600, 1900, 2600, 780, 1700, 580, 1500,
          1100, 1900, 1500, 3100 },
16    { 1600, 1700, 1800, 1500, 2600, 0, 750, 60, 2800, 3200, 2100, 1600,
          1600, 840, 1300, 1000 },
17    { 890, 970, 1000, 760, 1900, 750, 0, 770, 2000, 2600, 1400, 870,
          860, 150, 510, 1900 },
18    { 1700, 1700, 1800, 1500, 2600, 60, 770, 0, 2800, 3200, 2100, 1600,
          1600, 860, 1300, 1000 },
19    { 1800, 1900, 1900, 2000, 780, 2800, 2000, 2800, 0, 950, 1000, 1900,
          1400, 2100, 1700, 3500 },
20    { 2600, 2700, 2800, 2700, 1700, 3200, 2600, 3200, 950, 0, 2000,
          2600, 2200, 2600, 2300, 3900 },
21    { 820, 890, 900, 1100, 580, 2100, 1400, 2100, 1000, 2000, 0, 880,
          620, 1400, 980, 2300 },
22    { 110, 190, 270, 210, 1500, 1600, 870, 1600, 1900, 2600, 880, 0,
          460, 880, 630, 1400 },
23    { 480, 560, 640, 550, 1100, 1600, 860, 1600, 1400, 2200, 620, 460,
          0, 880, 570, 1700 },
```



## Soubor GlobalConstants.h (pokrač.)

```
24    { 900, 980, 1100, 780, 1900, 840, 150, 860, 2100, 2600, 1400, 880,
          880, 0, 540, 1500 },
25    { 650, 730, 810, 600, 1500, 1300, 510, 1300, 1700, 2300, 980, 630,
          570, 540, 0, 1700 },
26    { 1600, 1500, 1500, 1300, 3100, 1000, 1900, 1000, 3500, 3900, 2300,
          1400, 1700, 1500, 1700, 0 }
27 };
28
29 const int PopulationSize = 250;
30
31 const int NumberOfGenerations = 100;
32
33 const double MutationProbability = 1.0 / NumberOfMutablePubs;
34
35 const double MutationDelta = 0.2;
```



## Soubor Random.h

```
1  #pragma once
2
3  #include <cstdlib>
4
5  using namespace std;
6
7  inline double Random()
8  {
9     double dRet = rand();
10    dRet /= RAND_MAX + 1;
11    return dRet;
12 }
13
14 inline int Random(const int UpperLimit)
15 {
16    return rand() % UpperLimit;
17 }
```



## Soubor Utils.h

```
1  #pragma once
2
3  inline void Exchange(int& a, int& b)
4  {
5     int aux = a;
6     a = b;
7     b = aux;
8  }
9
10 inline void Exchange(double& a, double& b)
11 {
12    double aux = a;
13    a = b;
14    b = aux;
15 }
```

## Soubor Individual.h

```
1 #pragma once
2
3 #include <sstream>
4 #include <string>
5 #include "GlobalConstants.h"
6 #include "Random.h"
7 #include "Utils.h"
8
9 using namespace std;
10
11 struct Individual
12 {
13    double RandomKeys[NumberOfMutablePubs];
14    int Route[NumberOfMutablePubs];
15    int RouteLength;
16    double Fitness;
17 };
18
19
```



## Soubor Individual.h (pokrač.)

```
20 Individual CreateNewIndividual();
21
22 Individual CreateOffspringIndividual(const Individual& Parent1, const
       Individual& Parent2);
23
24 void PerformCrossover(const double ParentRandomKeys1[], const double
       ParentRandomKeys2[], double OffspringRandomKeys[]);
25
26 void PerformMutation(double RandomKeys[]);
27
28 void CreateRoute(const double RandomKeys[], int Route[]);
29
30 int ComputeRouteLength(const int Route[]);
31
32 double ComputeFitness(const int RouteLength);
33
34 string IndividualToString(const Individual& Individual);
35
36 inline void Exchange(Individual& a, Individual& b)
```



## Soubor Individual.h (pokrač.)

```
37 {
38    Individual aux = a;
39    a = b;
40    b = aux;
41 }
```



## Soubor Individual.cpp

```
1 #include "Individual.h"
2
3 Individual CreateNewIndividual()
4 {
5    Individual I;
6    for (int i = 0; i < NumberOfMutablePubs; i++)
7    {
8       I.RandomKeys[i] = Random();
9    }
10   CreateRoute(I.RandomKeys, I.Route);
11   I.RouteLength = ComputeRouteLength(I.Route);
12   I.Fitness = ComputeFitness(I.RouteLength);
13   return I;
14 }
15
16 Individual CreateOffspringIndividual(const Individual& Parent1, const
       Individual& Parent2)
17 {
18    Individual Offspring;
```

## Soubor Individual.cpp (pokrač.)

```
19    PerformCrossover(Parent1.RandomKeys, Parent2.RandomKeys, Offspring.
          RandomKeys);
20    PerformMutation(Offspring.RandomKeys);
21    CreateRoute(Offspring.RandomKeys, Offspring.Route);
22    Offspring.RouteLength = ComputeRouteLength(Offspring.Route);
23    Offspring.Fitness = ComputeFitness(Offspring.RouteLength);
24    return Offspring;
25 }
26
27 void PerformCrossover(const double ParentRandomKeys1[], const double
       ParentRandomKeys2[], double OffspringRandomKeys[])
28 {
29    int breakpoint;
30    do
31    {
32       breakpoint = Random(NumberOfMutablePubs);
33    } while (breakpoint == 0);
34    for (int i = 0; i < NumberOfMutablePubs; i++)
35    {
```



## Soubor Individual.cpp (pokrač.)

```
36       OffspringRandomKeys[i] = i < breakpoint ? ParentRandomKeys1[i] :
             ParentRandomKeys2[i];
37    }
38 }
39
40 void PerformMutation(double RandomKeys[])
41 {
42    int i = Random(NumberOfMutablePubs);
43    int sgn = Random() < 0.5 ? -1 : 1;
44    RandomKeys[i] += sgn * MutationDelta * RandomKeys[i];
45 }
46
47 void CreateRoute(const double RandomKeys[], int Route[])
48 {
49    double TempKeys[NumberOfMutablePubs];
50    for (int i = 0; i < NumberOfMutablePubs; i++)
51    {
52       TempKeys[i] = RandomKeys[i];
53       Route[i] = i;
```



## Soubor Individual.cpp (pokrač.)

```
54    }
55    bool exchange;
56    do
57    {
58       exchange = false;
59       for (int i = 0; i < NumberOfMutablePubs - 1; i++)
60       {
61          if (TempKeys[i] > TempKeys[i + 1])
62          {
63             Exchange(TempKeys[i], TempKeys[i + 1]);
64             Exchange(Route[i], Route[i + 1]);
65             exchange = true;
66          }
67       }
68    } while (exchange);
69 }
70
71 int ComputeRouteLength(const int Route[])
72 {
```



## Soubor Individual.cpp (pokrač.)

```
73    int pub1 = Route[0];
74    int pub2 = Route[NumberOfMutablePubs - 1];
75    int length = DistanceMatrix[StartFinishPubIndex][pub1] +
          DistanceMatrix[pub2][StartFinishPubIndex];
76    for (int i = 0; i < NumberOfMutablePubs - 1; i++)
77    {
78       pub1 = Route[i];
79       pub2 = Route[i + 1];
80       length += DistanceMatrix[pub1][pub2];
81    }
82    return length;
83 }
84
85 double ComputeFitness(const int RouteLength)
86 {
87    return 1.0 / RouteLength;
88
89 }
90
```

## Soubor Individual.cpp (pokrač.)

```
91 string IndividualToString(const Individual& Individual)
92 {
93    ostringstream os;
94    os << "Random keys (";
95    os << Individual.RandomKeys[0];
96    for (int i = 1; i < NumberOfMutablePubs; i++)
97    {
98       os << ", " << Individual.RandomKeys[i];
99    }
100   os << ")" << endl;
101   os << "Route (";
102   os << Individual.Route[0];
103   for (int i = 1; i < NumberOfMutablePubs; i++)
104   {
105      os << ", " << Individual.Route[i];
106   }
107   os << ")" << endl;
108   os << "RouteLength " << Individual.RouteLength << endl;
109   os << "Fitness " << Individual.Fitness << endl;
```



## Soubor Individual.cpp (pokrač.)

```
110   return os.str();
111 }
```



## Soubor GeneticAlgorithm.h

```
1  #pragma once
2
3  #include <ctime>
4  #include <fstream>
5  #include <iostream>
6  #include "GlobalConstants.h"
7  #include "Individual.h"
8  #include "Random.h"
9
10 using namespace std;
11
12 struct PopulationStatistics
13 {
14    int BestRoute[NumberOfMutablePubs];
15    int BestRouteLength;
16    double BestFitness;
17
18    int MeanRouteLength;
19    double MeanFitness;
```



## Soubor GeneticAlgorithm.h (pokrač.)

```
20
21    int WorstRouteLength;
22    double WorstFitness;
23 };
24
25
26 void PerformGeneticAlgorithm();
27
28 void ReportPopulation(const Individual Population[], const int PopSize,
       const string& ReportFilePath);
29
30 void ComputeStatistics(const Individual Population[],
      PopulationStatistics& Stat);
31
32 void ReportStatistics(const PopulationStatistics Stats[], const string&
       ReportFilePath);
33
34 void CreateInitialPopulation(Individual Population[]);
35
```

## Soubor GeneticAlgorithm.h (pokrač.)

```
36 void CreateRouletteWheel(const Individual Population[], double
       RouletteWheel[]);
37
38 void PerformSelection(const double RouletteWheel[], int& Parent1, int&
       Parent2);
39
40 void CopyPopulation(const Individual SourcePopulation[], Individual
       DestinationPopulation[]);
41
42 int TurnRouletteWheel(const double RouletteWheel[]);
43
44 void QuickSort(Individual A[], const int Left, const int Right);
```



## Soubor GeneticAlgorithm.cpp

```
1 #include "GeneticAlgorithm.h"
2
3 void PerformGeneticAlgorithm()
4 {
5    Individual Population[PopulationSize];
6    PopulationStatistics PopStat[NumberOfGenerations + 1];
7    srand((unsigned int)time(NULL));
8    CreateInitialPopulation(Population);
9    QuickSort(Population, 0, PopulationSize - 1);
10   ReportPopulation(Population, PopulationSize, "InitialPopulation.txt"
         );
11   ComputeStatistics(Population, PopStat[0]);
12   for (int g = 1; g <= NumberOfGenerations; g++)
13   {
14      cout << "Generation #" << g << endl;
15      cout << "Best individual" << endl;
16      cout << IndividualToString(Population[0]);
17      cout << endl << endl;
18      double RouletteWheel[PopulationSize];
```



## Soubor GeneticAlgorithm.cpp (pokrač.)

```
19      CreateRouletteWheel(Population, RouletteWheel);
20      Individual TempPopulation[2 * PopulationSize];
21      CopyPopulation(Population, TempPopulation);
22      for (int i = 0; i < PopulationSize; i++)
23      {
24         int Parent1Index, Parent2Index;
25         PerformSelection(RouletteWheel, Parent1Index, Parent2Index);
26         TempPopulation[i + PopulationSize] = CreateOffspringIndividual(
               Population[Parent1Index], Population[Parent2Index]);
27      }
28      QuickSort(TempPopulation, 0, 2 * PopulationSize - 1);
29      CopyPopulation(TempPopulation, Population);
30      ReportPopulation(Population, PopulationSize, "NewPopulation" +
            to_string(g) + ".txt");
31      ComputeStatistics(Population, PopStat[g]);
32   }
33   cout << "Finish" << endl;
34   cout << "-----------------------------------------------" << endl;
35   cout << "Best individual" << endl;
```



## Soubor GeneticAlgorithm.cpp (pokrač.)

```
36   cout << IndividualToString(Population[0]);
37   cout << endl;
38   ReportStatistics(PopStat, "PopulationStatistics.txt");
39 }
40
41
42 void ReportPopulation(const Individual Population[], const int PopSize,
       const string& ReportFilePath)
43 {
44   ofstream Output(ReportFilePath);
45   for (int i = 0; i < PopSize; i++)
46   {
47      Output << "Individual #" << i << endl;
48      Output << IndividualToString(Population[i]);
49      Output << endl;
50   }
51   Output.close();
52 }
53
```

## Soubor GeneticAlgorithm.cpp (pokrač.)

```
54 void ComputeStatistics(const Individual Population[],
       PopulationStatistics& Stat)
55 {
56    for (int i = 0; i < NumberOfMutablePubs; i++)
57    {
58       Stat.BestRoute[i] = Population[0].Route[i];
59    }
60    Stat.BestRouteLength = Population[0].RouteLength;
61    Stat.BestFitness = Population[0].Fitness;
62    Stat.MeanRouteLength = 0;
63    Stat.MeanFitness = 0;
64    for (int i = 0; i < PopulationSize; i++)
65    {
66       Stat.MeanRouteLength += Population[i].RouteLength;
67       Stat.MeanFitness += Population[i].Fitness;
68    }
69    Stat.MeanRouteLength /= PopulationSize;
70    Stat.MeanFitness /= PopulationSize;
71    Stat.WorstRouteLength = Population[PopulationSize - 1].RouteLength;
```



## Soubor GeneticAlgorithm.cpp (pokrač.)

```
72    Stat.WorstFitness = Population[PopulationSize - 1].Fitness;
73 }
74
75 void ReportStatistics(const PopulationStatistics Stats[], const string&
       ReportFilePath)
76 {
77    ofstream Output(ReportFilePath);
78    for (int i = 0; i <= NumberOfGenerations; i++)
79    {
80       Output << i << ";";
81       Output << "(";
82       Output << Stats[i].BestRoute[0];
83       for (int j = 1; j < NumberOfMutablePubs; j++)
84       {
85          Output << ", " << Stats[i].BestRoute[j];
86       }
87       Output << ");";
88       Output << Stats[i].BestRouteLength << ";";
89       Output << Stats[i].BestFitness << ";";
```



## Soubor GeneticAlgorithm.cpp (pokrač.)

```
90       Output << Stats[i].MeanRouteLength << ";";
91       Output << Stats[i].MeanFitness << ";";
92       Output << Stats[i].WorstRouteLength << ";";
93       Output << Stats[i].WorstFitness << ";";
94       Output << endl;
95    }
96    Output.close();
97 }
98
99
100 void CreateInitialPopulation(Individual Population[])
101 {
102    for (int i = 0; i < PopulationSize; i++)
103    {
104       Population[i] = CreateNewIndividual();
105    }
106 }
107
```



## Soubor GeneticAlgorithm.cpp (pokrač.)

```
108 void CreateRouletteWheel(const Individual Population[], double
       RouletteWheel[])
109 {
110    double sum = 0;
111    for (int i = 0; i < PopulationSize; i++)
112    {
113       sum += Population[i].Fitness;
114    }
115    for (int i = 0; i < PopulationSize; i++)
116    {
117       RouletteWheel[i] = Population[i].Fitness / sum;
118    }
119    for (int i = 1; i < PopulationSize; i++)
120    {
121       RouletteWheel[i] += RouletteWheel[i - 1];
122    }
123 }
124
```

## Soubor GeneticAlgorithm.cpp (pokrač.)

```
125 void PerformSelection(const double RouletteWheel[], int& Parent1, int&
      Parent2)
126 {
127    Parent1 = TurnRouletteWheel(RouletteWheel);
128    do
129    {
130       Parent2 = TurnRouletteWheel(RouletteWheel);
131    } while (Parent1 == Parent2);
132 }
133
134 void CopyPopulation(const Individual SourcePopulation[], Individual
      DestinationPopulation[])
135 {
136    for (int i = 0; i < PopulationSize; i++)
137    {
138       DestinationPopulation[i] = SourcePopulation[i];
139    }
140 }
141
```



## Soubor GeneticAlgorithm.cpp (pokrač.)

```
142 int TurnRouletteWheel(const double RouletteWheel[])
143 {
144    double r = Random();
145    for (int i = 0; i < PopulationSize; i++)
146    {
147       if (r < RouletteWheel[i])
148       {
149          return i;
150       }
151    }
152    return 0;
153 }
154
155 void QuickSort(Individual A[], const int Left, const int Right)
156 {
157    int i = Left;
158    int j = Right;
159    double pivot = A[(Left + Right) / 2].Fitness;
160    do
```



## Soubor GeneticAlgorithm.cpp (pokrač.)

```
161    {
162       while (A[i].Fitness > pivot)
163          i += 1;
164       while (pivot > A[j].Fitness)
165          j -= 1;
166       if (i <= j)
167       {
168          Exchange(A[i], A[j]);
169          i += 1;
170          j -= 1;
171       }
172    } while (i <= j);
173    if (Left < j)
174       QuickSort(A, Left, j);
175    if (i < Right)
176       QuickSort(A, i, Right);
177 }
```



## Soubor main.cpp

```
1 #include "GeneticAlgorithm.h"
2
3 void main()
4 {
5    PerformGeneticAlgorithm();
6 }
```

## Výsledky – statistické údaje

- Celkový počet možných tras: $15! = 1,30767 \times 10^{12}$ tj. $\approx 1,31$ bilionu.
- Celkem existuje 1833 navzájem různých, tj. unikátních, délek tras.
- Nejkratší trasa měří 10090 m, nejdelší 28450 m – rozdíl činí 18360 m.
- Průměrná délka trasy je cca 22194 m.
- Medián délek tras má hodnotu 22335 m.
- Modus délek tras má hodnotu 2 499 847 424 pro délku trasy 22600 m. Modus označuje nejčastější hodnotu v datech.

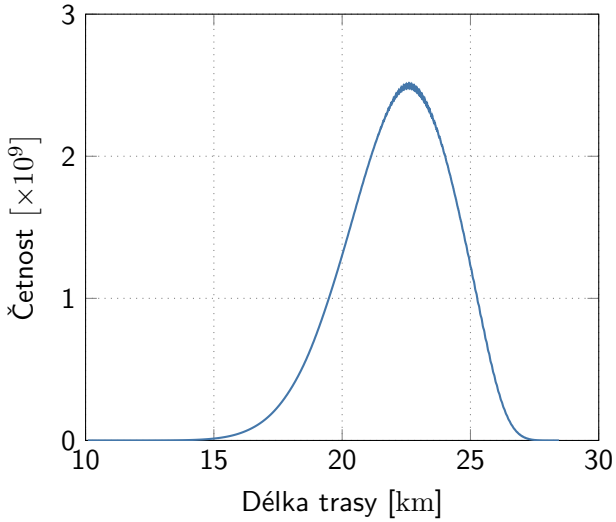


## Tabulka četností

| Délka trasy [m] | Četnost |
|---|---|
| 10090 | 2 |
| 10110 | 2 |
| 10120 | 2 |
| 10140 | 2 |
| ⋮ | ⋮ |
| 22590 | 2 493 603 794 |
| 22600 | 2 499 847 424 |
| 22610 | 2 492 130 770 |
| ⋮ | ⋮ |
| 28410 | 132 |
| 28420 | 70 |
| 28430 | 68 |
| 28440 | 14 |
| 28450 | 10 |

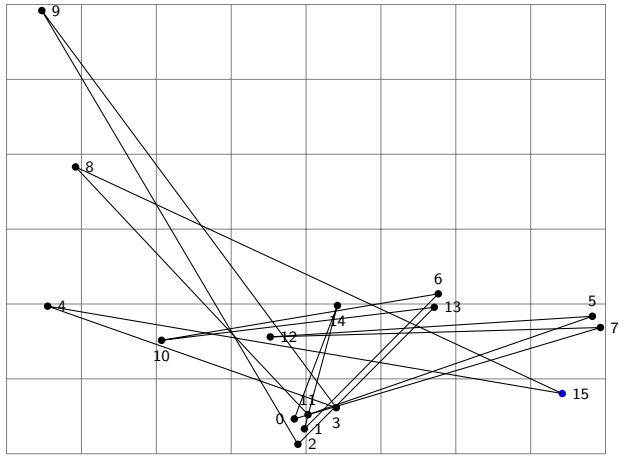


## Histogram délek tras

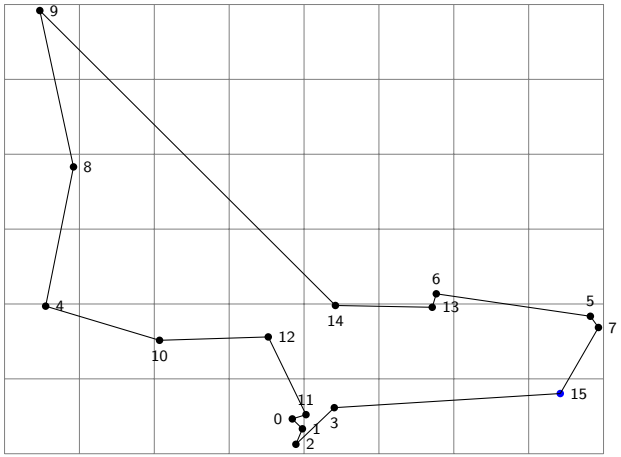





## Nejhorší trasa, 28450m

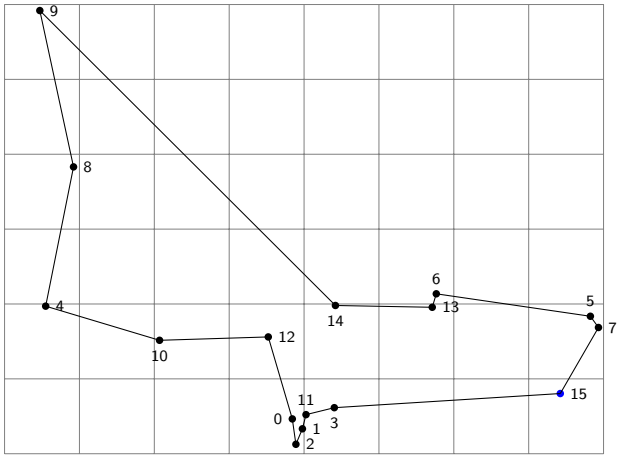


## Nejlepší trasa, 10090m






## Nejlepší trasa nalezená pomocí GA, 10150m



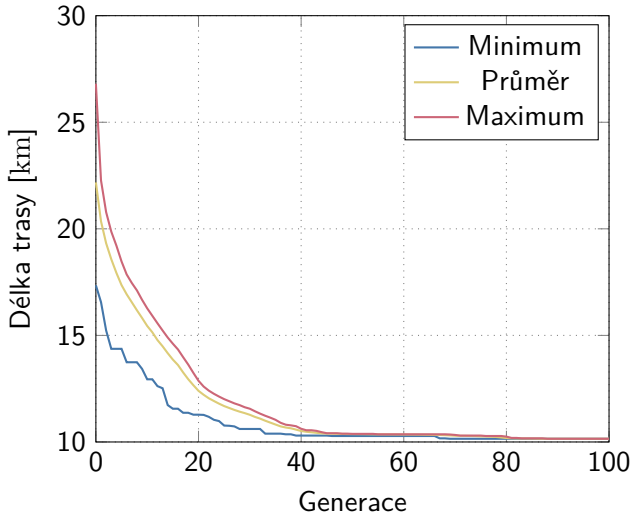



## Animace hledání řešení pomocí GA

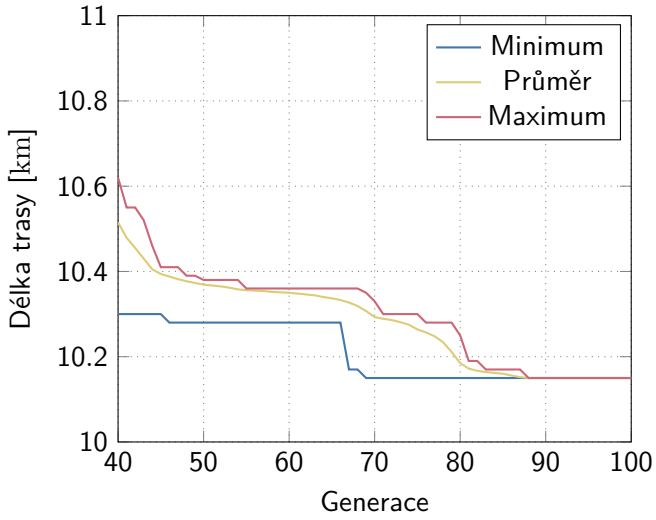
Animace jsou v poznámkách k prezentaci vynechány.



## Evoluce řešení v GA

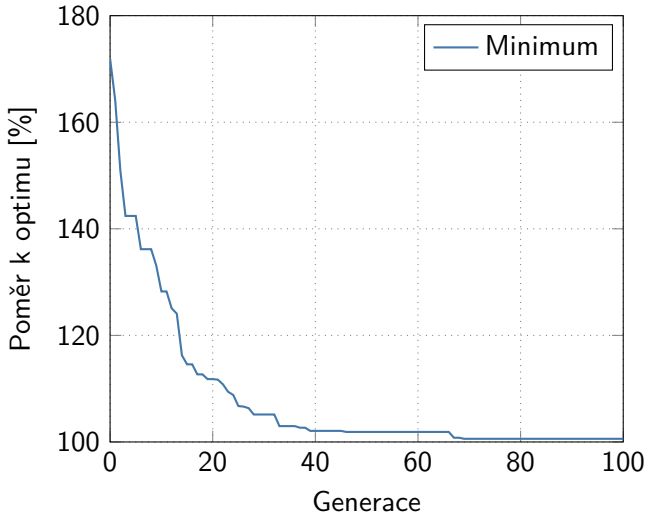


## Evoluce řešení v GA – detail






## Evoluce řešení v GA






Děkuji za pozornost