Algoritmizace a programování

RNDr. Miroslav Kubásek, Ph.D.

Obsah předmětu

- Algoritmy
- Programovací jazyky
- OOP
- Matlab
- C#
- Python

Algoritmus a jeho vlastnosti

Problém

Termín mající více významů

Denice 1 (Slovník spisovného jazyka českého)
„Věc k řešení, nerozřešená sporná otázka, otázka k rozhodnutí, nesnadná věc“

Denice 2 (Wikipedia)
„Podmínky nebo situace nebo stav, který je nevyřešený, nebo nechtěný, nebo nežádoucí.“

Problém vyžaduje řešení.
Pro nalezení řešení je nutné pochopit nejdůležitější aspekty problému.

Ne všechny problémy jsme v současné době schopni úspěšně a efektivně vyřešit !!!

Popis problému

Problém lze formálně zapsat takto:

NÁZEV PROBLÉMU: Slovní popis problému
VSTUP: Popis přípustného vstupu (množina vstupních dat).
VÝSTUP: Popis výstupu, tj. výsledku, který je pro daný vstup očekáván.

Musí existovat funkce *f* přiřazující vstupním datům požadovaný výstup.
Nalezení řešení problému → nalezení příslušné funkce *f*.

Denice:
Každý problém P je určen uspořádanou trojicí (IN; OUT; f), kde IN představuje množinu přípustných vstupů, OUT množinu očekávaných výstupů a f: IN → OUT funkci přiřazující každému vstupu očekávaný výstup.

VSTUP/VÝSTUP: Kombinace znaků, celých čísel či přirozených čísel představující kódování.

Algoritmus 1

Algoritmy vznikaly už dávno před zkonstruováním prvních počítačů. Samotné slovo "algoritmus" vzniklo ze jména perského matematika, který žil v 9. století a jmenoval se Mohammed al-Khowarizmí (v latinském přepise Algoritmus). Zabýval se především pravidly pro aritmetické operace s čísly. Například Eukleidův algoritmus pro výpočet největšího společného dělitele dvou přirozených čísel pochází už z 4. až 3. století před naším letopočtem.

Algoritmus je obecný předpis sloužící pro řešení zadaného problému.

Použití:

při konstrukci či analýze algoritmů
pro dokumentaci, pro zachycení myšlenky
setkáváme se s ním i v běžném životě: kuchařský recept, lékařský předpis.

Účel:

názornost a přehlednost pro pochopení při algoritmizaci
jednoznačný převod z textu programu do vývojového diagramu
spíše nevhodné při návrhu celého (složitějšího) programu

Algoritmus 2

Vstupní údaje
informace, ze kterých při řešení úlohy vycházíme, musí splňovat vstupní podmínku

Výstupní údaje
nově získané informace, které jsou výsledkem realizace algoritmu, musí splňovat výstupní údaje

Algoritmus A řeší problém P, pokud libovolnému vstupu x, x $\in$ IN, přiřazuje v konečném počtu kroků (alespoň jeden) výstup y, y $\in$ OUT, tak, že platí:

$$y = f(x)$$

Vlastnosti algoritmu

A) Determinovanost / Jednoznačnost
Algoritmus musí být jednoznačný jako celek i v každém svém kroku. Tohoto stavu nelze dosáhnout přirozenými jazyky, proto jsou pro zápis algoritmu používány formální jazyky.

B) Rezultativnost / Správnost
Algoritmus vede vždy ke správnému výsledku v konečném počtu kroků (vrátí třeba jen chybové hlášení).

C) Hromadnost
Algoritmus lze použít pro řešení stejné třídy problémů s různými vstupními hodnotami. Pro jejich libovolnou kombinaci obdržíme jednoznačné řešení.

D) Opakovatelnost
Při opakovaném použití stejných vstupních dat vždy obdržíme stejný výsledek.

E) Efektivnost
Každý krok algoritmu by měl být efektivní. Krok využívá elementární operace, které lze provádět v konečném čase.

F) Konečnost
Algoritmus se nezacyklí

Řešení problému pomocí algoritmu 1

Základní fáze řešení problému

1) Denice problému
Formulace problému společně s cílem, kterého chceme dosáhnout.
Definujeme požadavky týkající se tvaru vstupních a výstupních dat.

2) Analýza problému
Stanovení kroků a metod vedoucích k jeho řešení.
Rozkládání problému na dílčí podproblémy, jejichž řešení je jednodužší než řešení problému jako celku = dekompozice problému.

3) Sestavení algoritmu
Na základě analýzy navrhneme a sestavíme algoritmus řešící požadovaný problém.
Algoritmus může být popsán v přirozeném i formálním jazyce.

Řešení problému pomocí algoritmu 2

4) Kódování algoritmu
Převod algoritmu do formálního jazyka představovaného zpravidla programovacím jazykem.

5) Ověření správnosti algoritmu
Ověření funkčnosti algoritmu na konečném vzorku dat.
Splnění této podmínky však není postačující k ověření funkčnosti a správnosti algoritmu.
Algoritmus by měl být matematicky dokázán pro všechny možné varianty vstupních dat.

6) Nasazení algoritmu
Splňuje-li algoritmus požadavky na něj kladené ověřené jeho testováním, je možné jej nasadit do ostrého provozu.

Složitost algoritmu

Jeden problém může být řešen sadou různých algoritmů.
Kdyby počítač pracoval s nekonečnou rychlostí, nezáleželo by na volbě algoritmu.

V praxi vykonání instrukce trvá určitou dobu, nejrychlejší je přiřazení, nejpomalejší násobení/dělení. Vyhýbat se mocninám a odmocninám, velmi pomalý výpočet. Různé algoritmy se budou lišit dobou běhu, a to mnohdy velmi významně.

Jak exaktně určit, rychlost algoritmu a množství spotřebované paměti?

Dvě základní kritéria:

Časová složitost algoritmu (Time Complexity)
Popisuje dobu zpracování vstupních dat D algoritmem A v čase T, vyjádřena časovou funkcí t

$$T = t(A(D))$$

Paměťová složitost algoritmu (Space Complexity)
Popisuje paměťovou náročnost algoritmu.

Posuzování složitosti algoritmu 1

Doba běhu algoritmu i paměťová náročnost se mění, existují dva přístupy vyjadřující dobu běhu:

A) V závislosti na velikosti vstupní množiny

Složitost lze definovat jako funkcí velikosti vstupu ***n***.

Závislost doby běhu na ***n*** lze hledat v exaktním tvaru (např. $4n^3-9n^2+20n+27$), což není časté, nebo běžně ve formě odhadu (např. $O(n^3)$).

B) V závislosti na vstupní množině dat

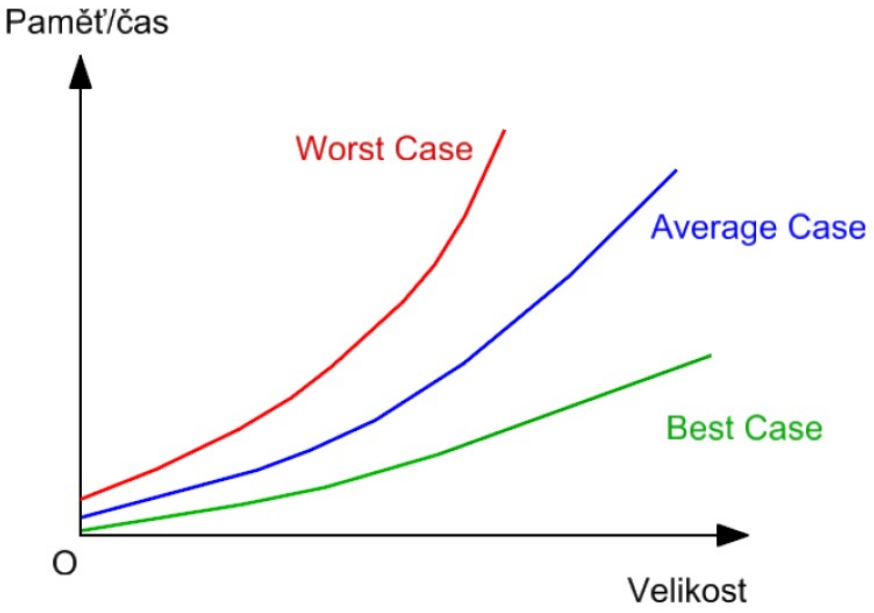
Pro vstupní množinu se stejnou velikostí ***n*** se doba běhu může výrazně měnit

(např. třídící algoritmus se chová výrazně jinak, pokud má na vstupu náhodnou, částečně setříděnou nebo reverzně setříděnou posloupnost dat).

Posuzování složitosti algoritmu 2

Složitost lze posuzovat:

- dle nejhoršího možného případu (Worst Case)
- dle průměrné doby běhu (Average Case)
- dle nejlepšího možného případu (Best Case)



Posuzování složitosti algoritmu 3

Složitost	Vyjádření	Charakteristika
Konstantní	1	Konstantní doba běhu programu. Nezávisí na vstupních datech.
Logaritmická	$\log(n)$	Doba běhu se mírně zvětšuje v závislosti na N. Řešení hledáno opakovaným dělení vstupní množiny na menší množiny (hledání v binárním stromu).
Lineární	n	Doba běhu programu roste lineárně s N. Zpracováván každý prvek, např.cyklus.
$n\log(n)$	$n\log(n)$	Doba běhu roste téměř lineárně. Opakované dělení vstupního problému na menší problémy, které jsou řešeny nezávisle (Divide and Conquer, např. třídění).
Kvadratická	n^2	Doba běhu roste kvadraticky, vhodný pro menší množiny dat. Vnořený cyklus.
Kubická	n^3	Doba běhu roste s třetí mocninou, dvojnásobně vnořený cyklus. V praxi snaha nahrazovat algoritmus předchozími dvěma kategoriemi (Greedy algoritmy)
Exponenciální	2^n	Exponenciální doba běhu. Použitelné pro množiny do n=30 Aplikace v kryptografii.

Ukázka časové složitosti algoritmů

f(n)	n							
	20	40	60	80	100	200	500	1 000
n	20 μs	40 μs	60 μs	80 μs	0.1 ms	0,2 ms	0,5 ms	1 ms
n.log (n)	86 μs	0,2 ms	0,35 ms	0,5 ms	0.7 ms	1,5 ms	4,5 ms	10 ms
n^2	0,4 ms	1,6 ms	3,6 ms	6,4 ms	10,0 ms	40 ms	0,25 s	1 s
n^3	8,0 ms	64 ms	0,22 s	0,5 s	1 s	8 s	125 s	17 min
n^4	0,16 s	2,56 s	13 s	41 s	100 s	27 min	17 h	11,6 dní
2^n	1,00 s	11,7 dní	36 600 let	$3{,}6.10^9$ let				
n!	77 000 let							

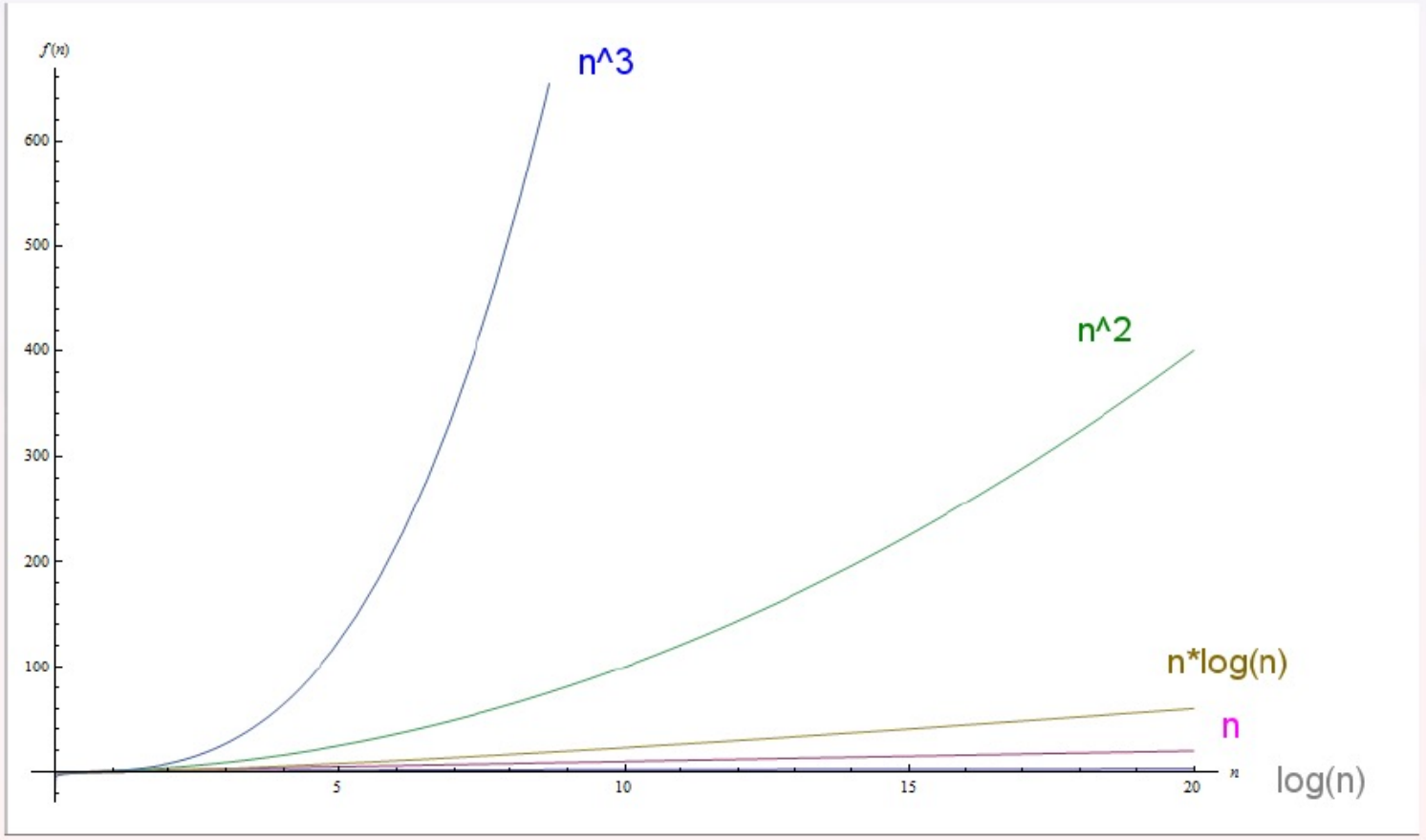
Ukázka doby běhu algoritmů

Ukázka doby běhu algoritmu pro $n = 10^9$.
CPU: Počet operací/s $\frac{10^6}{s}, \frac{10^9}{s}, \frac{10^{12}}{s}$.

Složitost	**CPU** $\frac{10^6}{s}$	**CPU** $\frac{10^9}{s}$	**CPU** $\frac{10^{12}}{s}$
Logaritmická složitost	hodiny	vteřiny	okamžitě
Lineární složitost	hodiny	vteřiny	okamžitě
$N \log(N)$	hodiny	vteřiny	okamžitě
Kvadratická složitost	nikdy	roky	týdny
Kubická složitost	nikdy	nikdy	měsíce
Bikvadratická složitost	nikdy	nikdy	roky
Exponenciální složitost	nikdy	nikdy	nikdy
Faktoriální složitost	nikdy	nikdy	nikdy

Grafické znázornění časové složitosti

Vstupní množina: $n \in (0, 20)$



Techniky návrhu algoritmů

Návrh algoritmu prováděn dvěma technikami:

- **Návrh algoritmu Shora dolů,**
- **Návrh algoritmu Zdola nahoru.**

V praxi obě techniky z důvodů větší efektivity často kombinujeme.

Představují analogie postupů používaný při řešení běžných problémů.

Návrh algoritmu „Shora dolů“

Odvozen z postupu, jakým člověk **řeší** složité úkoly v praxi.

Problém rozkládáme na jednodušší kroky, tzv. *dekompozice problému.*

Postup směrem od ***obecného*** ke ***konkrétnímu***.

Postupně tak dospějeme k elementárním krokům, které již zpravidla umíme poměrně snadno vyřešit.

Návrh algoritmu metodou shora-dolů je nejpoužívanějším přístupem.

Příkladem využití návrhu shora dolů může být metoda Divide&Conquer.

Návrh algoritmu „Zdola nahoru“

Opačný přístup než Shora dolů.

Elementární kroky se snažíme za použití abstrakce sdružovat do složitějších celků ve snaze nalézt řešení zadaného problému.

Postup směrem od ***konkrétního*** k ***obecnému***.

Používán převážně při implementaci algoritmu:

1) Nejprve vytváříme nejjednodušší komponenty.
2) Jednoduché komponenty spojujeme ve složitější komponenty.
3) Jako poslední vytváříme zpravidla hlavní program.

Zdroje chyb v algoritmu

1) Chyba vstupních dat

Nepřesné určení vstupních dat, např. nepřesné měření fyzikálních parametrů. Tuto chybu označujeme jako počáteční chybu.

2) Chyba modelu

Algoritmus vychází ze zjednodušeného modelu skutečnosti. Některé předpoklady nemusí platit, nebo platí pouze přibližně. K možnému ovlivnění výsledku dochází také zanedbáním některého z parametrů.

3) Chyba matematických metod

V řadě případů používáme přibližné numerické řešení (např. výpočet prvních x členů nějaké řady, aproximace funkčního vztahu, atd). Lze je částečně odstranit použitím „přesnějšího“ řešení.

4) Chyba z početních operací

Vznikají zaokrouhlením hodnot při početních operacích, jsou neodstranitelné. Při návrhu matematického modelu je nutno použít takové metody, při kterých nedochází ke kumulaci zaokrouhlovacích chyb.

Návrh algoritmů

Základní pojmy 1

Syntaxe příkazu vždy popisuje, jak tento příkaz správně bez chyby vytvořit.

Sémantika popisuje význam tohoto příkazu.

Proměnná (variable) je objekt, který má pevně stanovené označení a nese určitou hodnotu. Tato hodnota se může v průběhu programu měnit. Pro označení proměnných se používají jména složená z písmen a číslic, první však musí být písmeno.

Konstanta (constant) je také pojmenovaný objekt určité hodnoty, na rozdíl od proměnné se hodnota konstanty nemůže měnit. Konstanta pi například může obsahovat hodnotu 3.14.
Proměnná může získávat hodnotu dvěma způsoby: načtením vstupní hodnoty pomocí vstupního bloku nebo přiřazovacím příkazem

Proměnná, která dosud nezískala žádnou hodnotu, má nedefinovaný obsah.

Základní pojmy 2

Přiřazovací příkaz je základem všech algoritmů zapsaných jak pomocí diagramů, tak i v libovolném programovacím jazyce. Syntaxe přiřazovacího příkazu je **proměnná:=výraz**. Symbol přiřazení se nachází uprostřed, často se bude jednat o znak :=. Výrazy mohou obsahovat konstanty i proměnné, aritmetické operátory, kulaté závorky. Základní aritmetické operátory jsou +,-,*,/.

Sémantika přiřazovacího příkazu je následující:
Nejprve se vyhodnotí hodnota výrazu na pravé straně přiřazovacího příkazu. Tato hodnota je pak přiřazena proměnné uvedené na levé straně příkazu. Předchozí hodnota této proměnné je nenávratně ztracena.

Příklad několika přiřazení (: a:=0 b:=a+5 c:=2*(b+4) a:=a+1 vysvětlení: do proměnné a je vložena hodnota nula, do b se přiřadí hodnota o 5 větší, tedy 5, v c bude 2*(5+4)=18 a nakonec do a bude přiřazena hodnota o jedničku větší, než má teď, tedy 1

Základní pojmy 3

Podmínka (condition) je logický výraz, jehož hodnotou je pravda nebo nepravda. Říkáme, že podmínka platí nebo neplatí.

Základní relační operátory jsou <,>,=,<=,>=,<>.
Ve výrazu lze použít proměnné, konstanty i kulaté závorky. Obecně v podmínce může být výraz vlevo i vpravo, navíc se dají sestavovat složené podmínky pomocí logických spojek. Toho si ale užijete až v konkrétním programovacím jazyce.

Sémantika je následující:
Nejprve se vyhodnotí podmínka. Pokud platí, provede se příkaz (nebo více příkazů) uvedený ve větvi označené symbolem +. Pokud podmínka neplatí, provede se příkaz (nebo více příkazů) uvedený ve větvi označené symbolem -. Příkazem uvnitř může být přiřazovací příkaz, vstup nebo výstup dat i další podmíněný příkaz. V posledním případě mluvíme o složeném podmíněném příkazu nebo prostě o podmínce v podmínce. Jedna z větví v rozhodovacím bloku může být prázdná, neobsahuje žádný příkaz. Obvykle se jedná o větev označenou -. Takový podmíněný příkaz nazýváme neúplný. Úplný podmíněný příkaz má v každé větvi alespoň jeden příkaz.

Znázorňování algoritmů

Algoritmus lze znázorňovat mnoha způsoby.

Nejčastěji je používáno:

- **Grafické vyjádření algoritmu**
- **Textové vyjádření algoritmu**

Grafické vyjádření algoritmu

Algoritmus je popsán formalizovanou soustavou grafických symbolů.

Používány vývojové diagramy nebo strukturogramy.

Výhody: přehlednost, názornost, znázornění struktury problému, poskytuje informace o postupu jeho řešení.

Nevýhody: náročnost konstrukce grafických symbolů a jejich vzájemných vztahů, obtížná možnost dodatečných úprav postupu řešení vedoucí často k „překreslení" celého postupu, technika není vhodná pro rozsáhlé a složité problémy,

Vývojové diagramy

Jeden z nejčastěji používaných prostředku pro znázorňování algoritmu. Tvořeny značkami ve formě uzavřených rovinných obrazců, do kterých jsou vepisovány slovní či symbolickou formou jednotlivé operace. Tvary a velikosti značek jsou dány normami.

Značky jsou spojeny přímými nebo lomenými spojnicemi čarami a znázorňují tak posloupnosti jednotlivých kroků. Čáry mohou být orientované zavedením šipek, neměly by se křížit. Pokud již ke křížení dojde, měly by být čáry zvýrazněny tak, aby bylo jednoznačně patrné, odkud a kam směřují.

Vývojový diagram čteme ve směru shora dolů.

Výhody: názornost, přehlednost

Nevýhody: pracnost a složitost konstrukce, složité diagramy se nevejdou na jednu stránku a stávají se méně přehlednými, malé možnosti pozdějších úprav.

Komponenty vývojového diagramu

Start/konec algoritmu:

Počáteční a koncový krok algoritmu.

Vstup/Výstup:

Načtení dat potřebných pro běh programu, uložení dat.

Předzpracování dat:

Incializace proměnných.

Zpracování dat:

Dochází k transformaci dat. Jeden vstup a jeden výstup, nesmí dojít k rozvětvení programu.

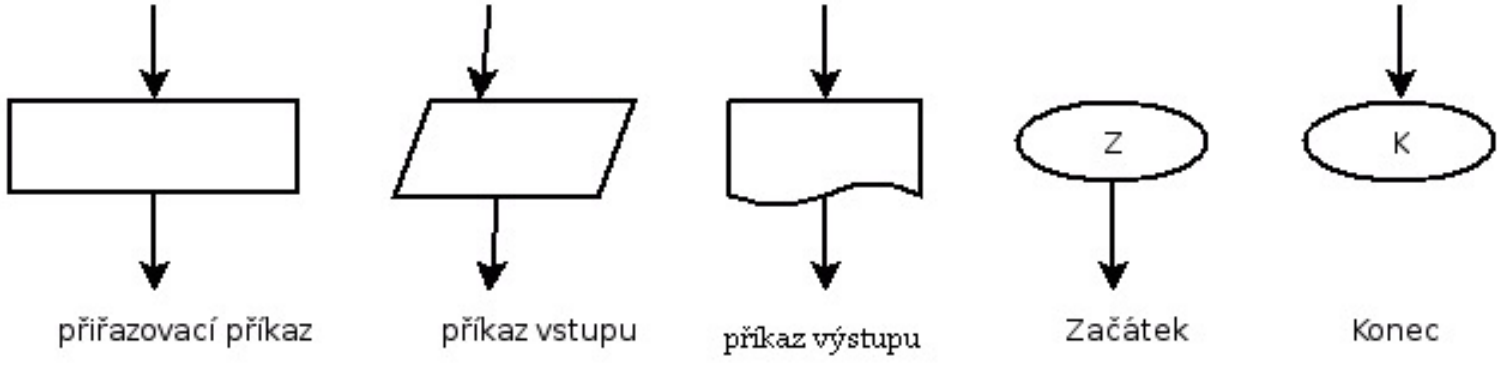
Rozhodovací blok:

Větvení na základě vstupní podmínky. Je-li splněna, pokračuje se větví (+), v opačném případě větví (-).

Podprogram:

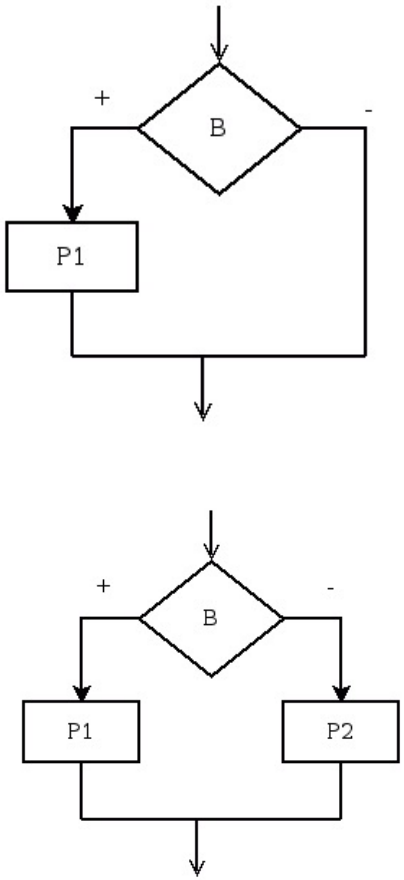
Samostatná část programu tvořená větším počtem kroků.

Bloky vývojového diagramu

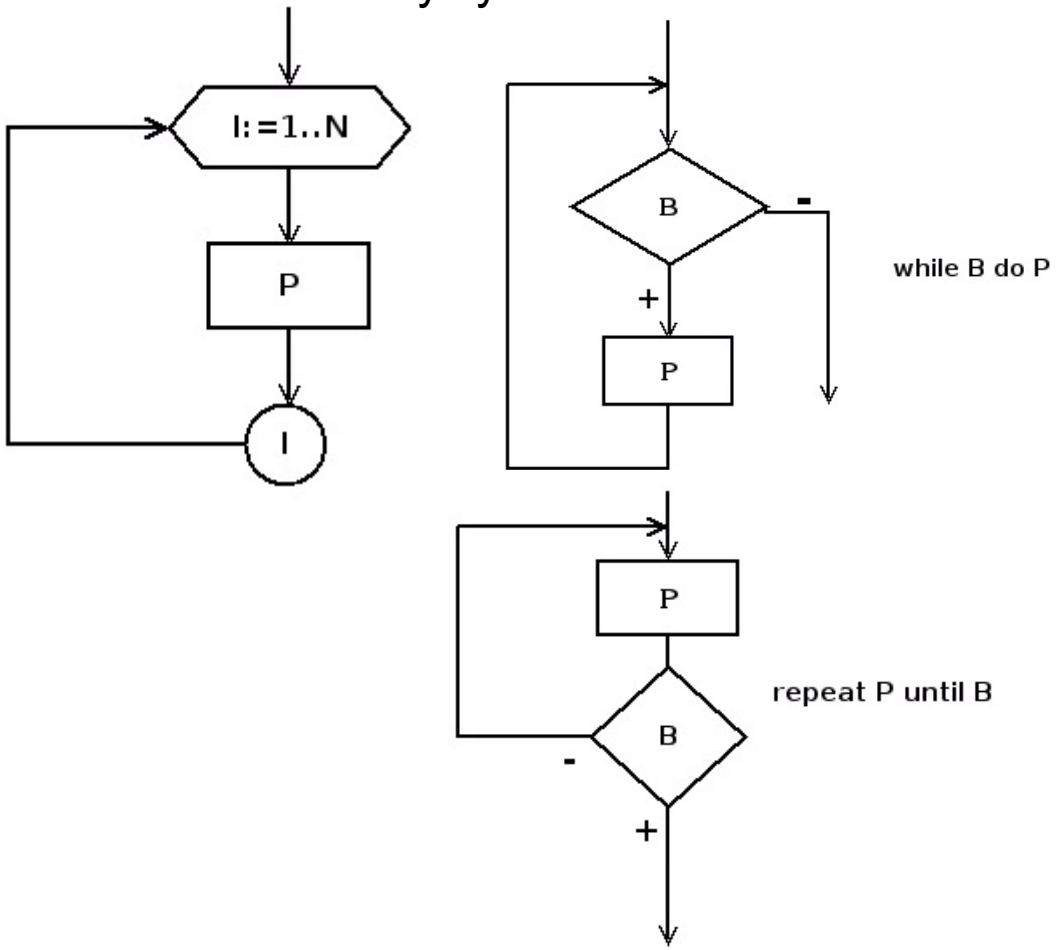


Základní konstrukce

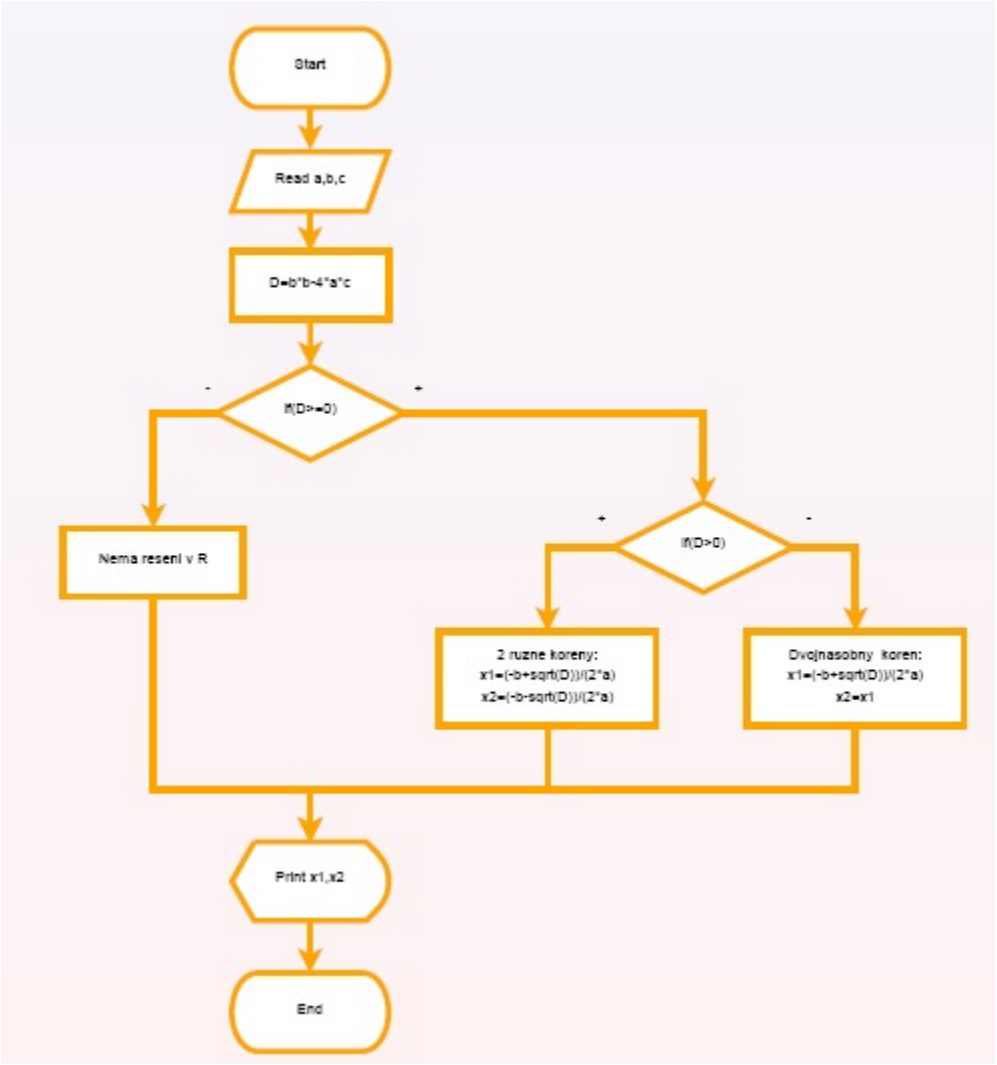
Větvení



Cykly



Vývojových diagram řešení kvadratické rovnice



Strukturogram

Úspornější znázornění algoritmu, kombinace grafického a textového popisu.
Tvořen obdélníkovou tabulkou, do řádku zapisujeme postup kroku symbolickou či slovní formou v poradí, v jakém budou provádeny. Záhlaví tabulky obsahuje název algoritmu nebo dílčího kroku.

Výhody:

Přehlednější způsob znázornění.

Lze ho aplikovat i na složitější problémy.

Jednoznacný a snadný prepis do formálního jazyka.

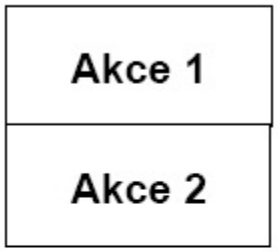
Nevýhody:

Pracnost konstrukce, složité strukturogramy se nevejdou na jednu stránku.

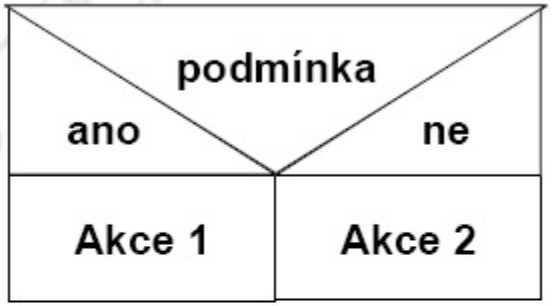
Malé možnosti pozdějších úprav.

Strukturogram – základní tvary

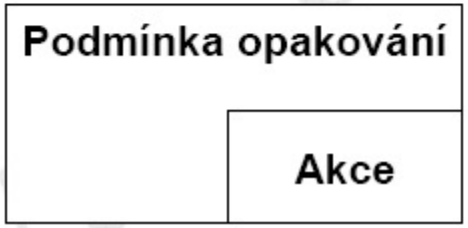
- **sekvence**



- **selekce**



- **iterace**



Strukturogram - příklad

<table>
<tr><td colspan="4">Řešení kvadratické rovnice</td></tr>
<tr><td colspan="4">Čti a, b, c</td></tr>
<tr><td colspan="4">Spočti diskriminant D=b*b -4*a*c</td></tr>
<tr><td colspan="4">Je D>0?</td></tr>
<tr><td rowspan="4"></td><td>Ano</td><td colspan="2">Ne</td></tr>
<tr><td>x1=(-b+sqrt(D))/(2*a)</td><td colspan="2">Je D=0 ?</td></tr>
<tr><td>x2=(-b-sqrt(D))/(2*a)</td><td rowspan="3"></td><td>Ano</td><td>Ne</td></tr>
<tr><td>Tisk 2 kořeny:
x1, x2</td><td>x1=(-b+sqrt(D))/(2*a)
Tisk 1 kořen:
x1</td><td rowspan="2">Tisk:
Nemá v R řešení.</td></tr>
</table>

Textové vyjádření algoritmu

Tento způsob vyjádření algoritmu se v současné době používá nejčastěji. Zápis algoritmu prostřednictvím formalizovaného jazyka. Využíván pesudokód nebo PDL (Program Description Language).

Výhody:

Přehlednost zápisu.

Jednoznačnost jednotlivých kroku.

Snadný přepis do programovacího jazyku (nejblíže ke skutečnému programovacímu jazyku).

Možnost pozdější modifikace postupu řešení.

Textové vyjádření - příklad

Algoritmus 3: Kvadratická rovnice (a,b,c)

```
1: Read (a,b,c)
2: D=b*b-4*a*c
2: If D > 0:
3:         x1=(-b+sqrt(D))/(2*a)
4:         x2=(-b- sqrt(D))/(2*a)
5: Else if D = 0
6:         x1=(-b+sqrt(D))/(2*a)
7:         x2=x1
8: else
9:         x1=∅, x2=x1
10: Print (x1,x2)
```

Příklady

1. Sestrojte algoritmus, který vytiskne absolutní hodnotu zadaného čísla. Udělej podle definice absolutní hodnoty
   Sestrojte algoritmus, který načte tři čísla a vytiskne největší z nich.
2. Je dáno přirozené číslo. Rozhodněte, zda je jednociferné, dvouciferné či víceciferné
3. Sestavte vývojový diagram, který pozdraví podle zadané denní doby vyjádřené v hodinách. Dopoledne do 12 hodin pozdraví Dobré dopoledne, do 18 hodin Dobré odpoledne, později Dobrý večer.
4. Sestavte vývojový diagram, který vypíše, kolik minut uplynulo mezi dvěma časovými údaji, např. mezi 9:30 a 11:15 uplynulo 105 minut.
5. Sestavte algoritmus pro výpočet zbytku po dělení dvou přirozených čísel. (Využijte cyklus while.)
6. Načítej čísla tak dlouho, dokud nebude zadána nula. Zadaná čísla sečti a vytiskni výsledek.
7. Je dána posloupnost kladných celých čísel zakončená nulou. Určete, kolik je v ní lichých čísel dělitelných třemi.
8. Sestrojte algoritmus, který vypočítá faktoriál zadaného přirozeného čísla. (Pozor 0! = 1)
9. Určete, zda je zadané číslo prvočíslo. (Předpokládejte zadání přirozeného čísla.)

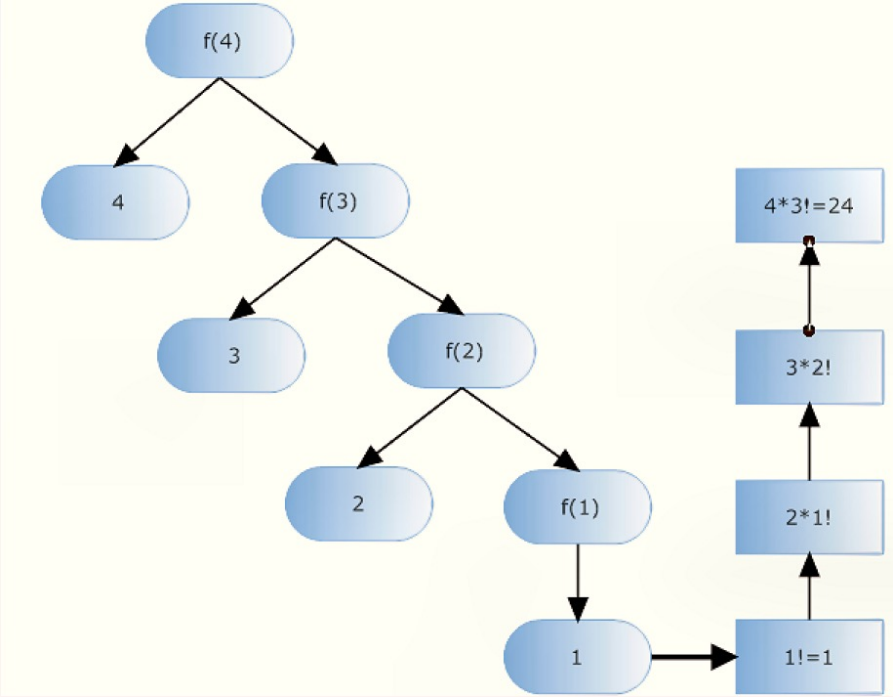
Příklady

1. Sestrojte algoritmus, který vytiskne všechny dělitele zadaného přirozeného čísla.
2. Sestrojte algoritmus, který načte N čísel a vytiskne, kolik jich je lichých.
3. Sestrojte algoritmus, který sečte čísla od 1 do N. N bude zadáno.
4. Sestrojte algoritmus, který načte 100 čísel a zjistí kolik z nich je kladných.
5. Sestrojte algoritmus, který načte n čísel a vypočítá jejich aritmetický průměr.
6. Sestrojte algoritmus, který načte n čísel a zjistí největší z nich.
7. Sestrojte algoritmus, který vypočítá n-tou mocninu čísla x.
8. Sestrojte algoritmus, který načte n čísel a zjistí kolik z nich je lichých.
9. Sestrojte algoritmus, který načte číslo n a vytiskne všechny jeho dělitele.
10. Sestrojte algoritmus, který načte dvě čísla a vrátí zbytek po celočíselném dělení prvního čísla druhým.
11. Sestrojte algoritmus, který vytiskne čtyři zadaná čísla podle velikosti.
12. Bude zadáno N čísel. Vytiskněte číslo s největší absolutní hodnotou.
13. Nalezněte největší mocninu dvojky, která je menší než zadané přirozené číslo.
14. Určete, zda je posloupnost 10 prvků rostoucí, klesající nebo není monotónní.
15. Číslo ve dvojkové soustavě převeďte do soustavy desítkové.

Příklady

1. Sestavte a nakreslete diagram algoritmu pro porovnání obsahu množiny A a množiny B . Prvky množiny A i B jsou celá čísla, počet prvků množiny A je N a množiny B je M. Pokud množiny mají společný prvek (číslo) zobrazte je na výstupním zařízení.
2. Sestavte a nakreslete diagram algoritmu pro součet matic A a B. Prvky množiny A i B jsou reálná čísla, počet prvků množiny A je N×N a množiny B je M×M. Množinu C zobrazte na výstupním zařízení.
3. Sestavte a nakreslete diagram algoritmu pro součin matice *A a vektoru B.* Prvky množiny *A i B jsou reálná čísla, počet prvků množiny A je N×N a množiny B je M.* Množinu *C zobrazte na výstupním zařízení.*

Rekurse - faktoriál



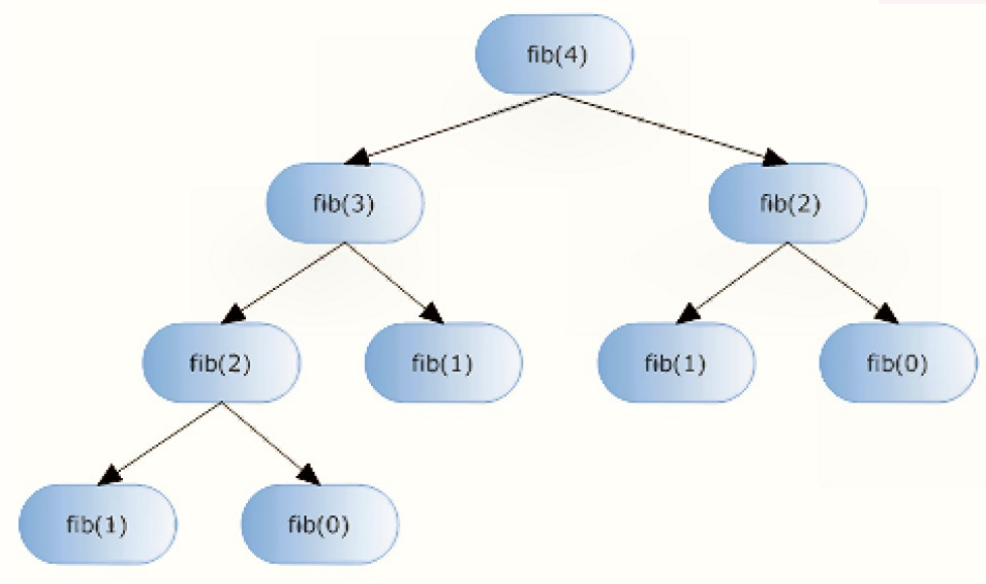
Výpočet faktoriálu s použitím rekurze:

```
int f(int n)
 {
  if (n>1)  //Ukoncovaci podminka
    {
      return n*f(n-1);
    }
  else return 1; //Ukonceni rekurze
 }
```

Rekurse – fibonacciho číslo

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

```
int  fib(int n)
 {
   if (n>1)  //Podminka ukonceni rekurze
     {
       return fib(n-1)+fib(n-2);
     }
   else return 1; //Ukonceni rekurze
 }
```



Fibonacciho číslo – bez rekurze

```
int fib(int n)
 {
  int fi=1;
  int fii=1;
  for (int i=0;i<n;i++)
   {
      fi = fi + fii; //Vypocet predchoziho clenu
      fii = fi - fii; //Vypocet nasledujiciho clenu
   }
  return fii;
 }
```

Nalezení maxima

Dána neuspořádaná posloupnost prvků $x = \{x_i\}, i = 1, ..., N$.
Nalezněte hodnotu $x_{max} = \max(x_i)$.

Varianty:

- Bez použití rekurze.
- S použitím rekurze.

Nalezení minima bez použití rekurze:

- hodnotu x_{max} inicializujeme prvním prvkem posloupnosti, tj. $x_{max} = x[0]$.
- porovnáme x_{max} s ostatními členy posloupnosti. Pokud $x_{max} > x[i]$, pak $x_{max} = x[i]$.
- po provedení porovnání se všemi prvky $x_{max} = \max(x_i)$.

Složitost algoritmu $O(N)$.

Nalezení maxima - klasicky

```
double maximum(double [] x)
 {
   double max=x[0];   //Inicializace maxima
   for (int i=0;i<c.length;i++)
    {
       //Prirazeni nejvetsiho cisla jako maxima
       if max>x[i] max=x[i];
    }
   return max;
 }
```

Nalezení maxima - rekurzí

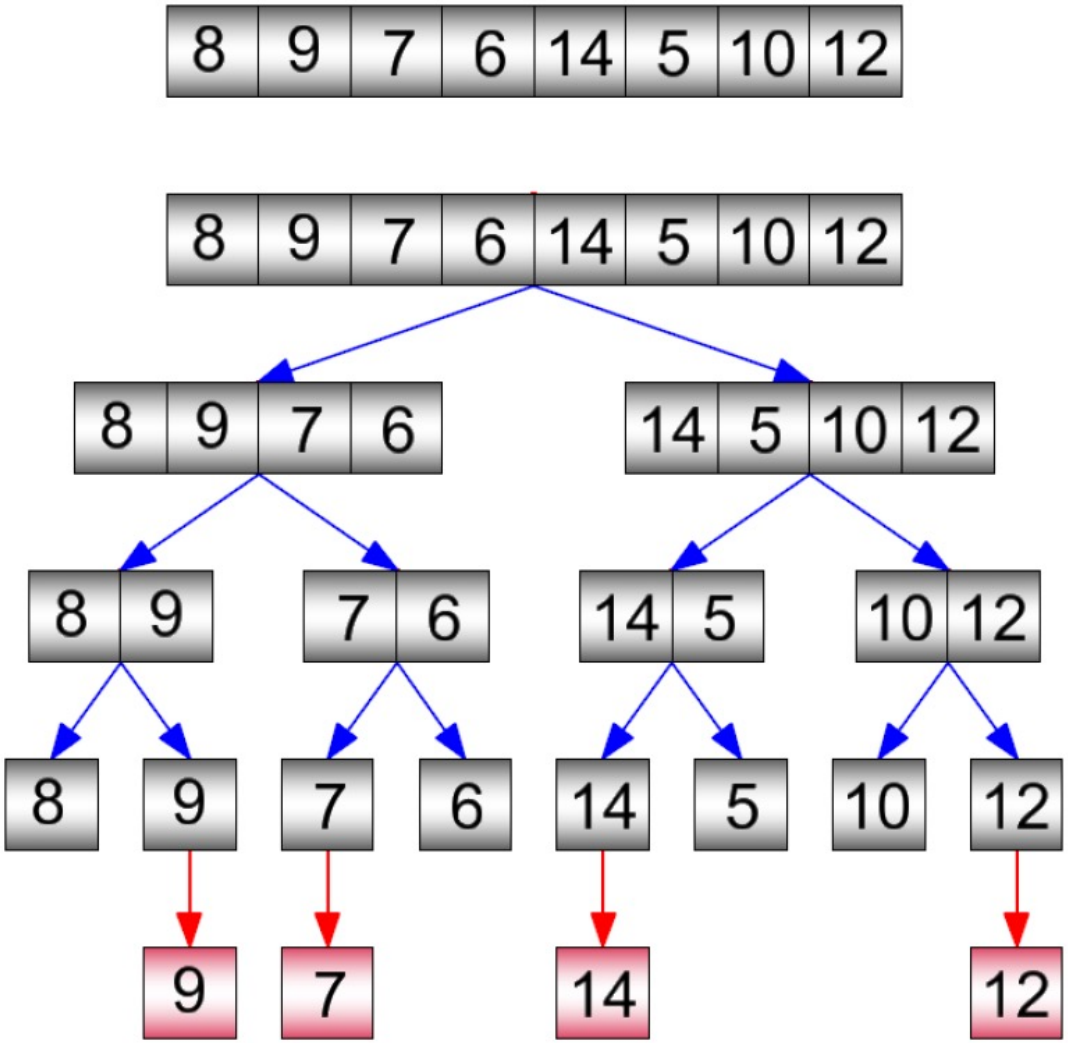
Úloha je dekomponovatelná na úlohy stejné třídy:

1. Hledání maxima v posloupnosti n prvků lze převést na dva stejné podproblémy, a to hledání maxima ve dvou posloupnostech tvořených $n/2$ prvky: $\{x_0, x_1, \ldots, x_{n/2}\}$ a $\{x_{n/2+1}, x_{n/2+2}, \ldots, x_{n-1}\}$.
2. Hledání maxima v těchto posloupnostech lze řešit stejným způsobem, a to rozdělením každé z nich na dvojici posloupností s $n/4$ prvky, atd...
3. Pro $n = 2$ je funkce maximum volána pro posloupnosti tvořené dvěma za sebou následujícími prvky x_i, x_{i+1}.
4. Pro $n = 1$ vrací funkce přímo hodnotu prvku $\{x_i\}$.

---

5. Ukončení rekurzivního volání.
6. Nalezení maxima z dvojice za sebou následujících prvků ve všech posloupnostech a jejich vrácení $max = max(x_i, x_{i+1})$.
7. Nalezení maxim z těchto maxim:
   atd...$max = max(max(x_i, x_{i+1}), max(x_{i+2}, x_{i+3}))$, atd...

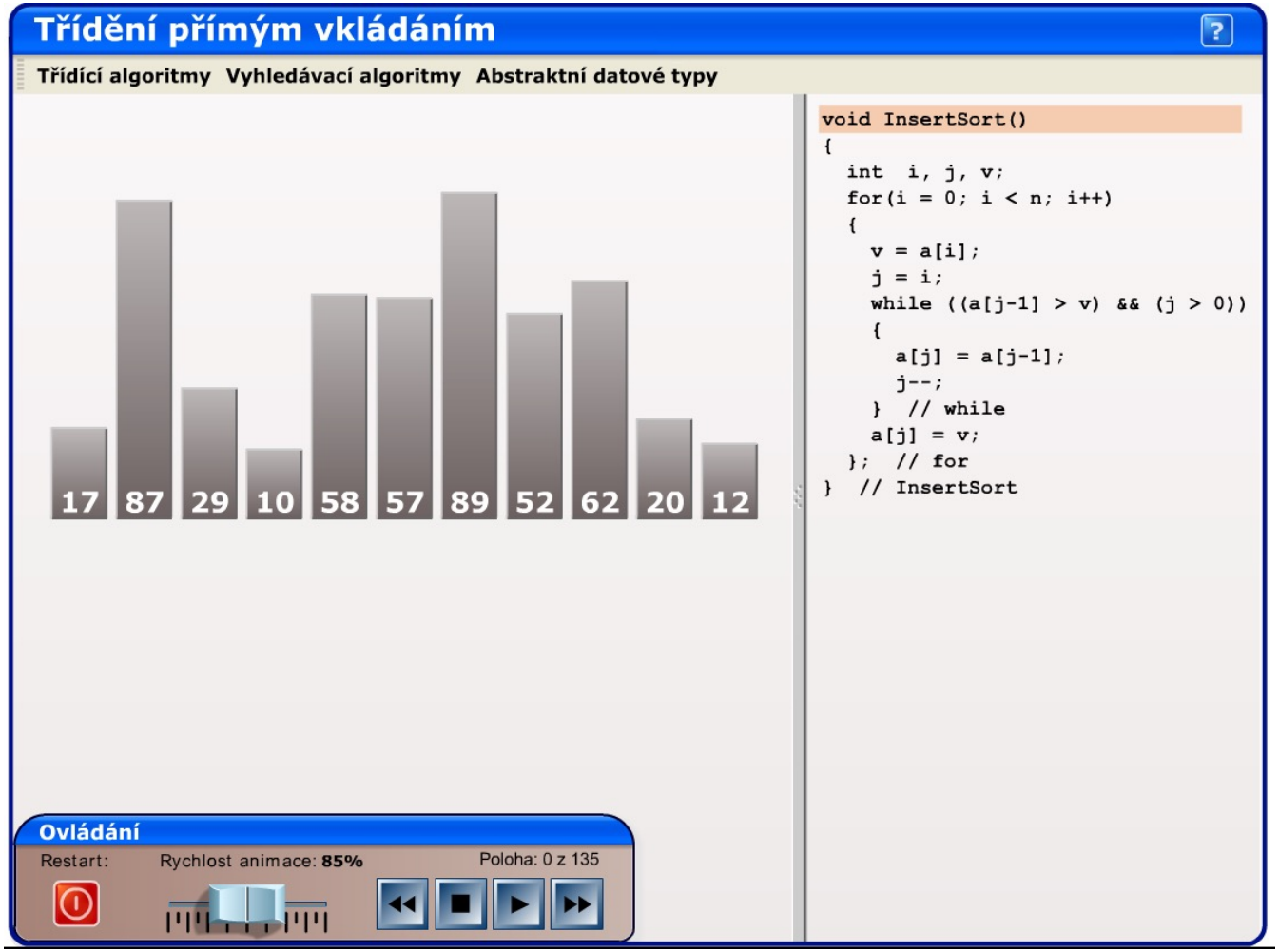
Nalezení maxima – rekurzí – dělení intervalu



Nalezení maxima – rekurzí - algoritmus

```
double maximum(double [] pole, int od, int do)
 {
      if (od == do) return pole[od];  //Ukonceni rekurze
      mid=(od+do)/2;  //Nalezeni prostredniho prvku
      levy = maximum(pole, od, mid);  //Rekurze,levy interval
      pravy = maximum(pole, mid+1,do); //Rekurze, pravy interval
      if (levy > pravy)  // Ktery z prvku je vetsi?
          return levy;   //Vrat levy prvek
     else
          return pravy;  //Vrat pravy prvek
 }
```

Základní techniky algoritmů
vizuální ukázky typů algoritmů



Programovací jazyky

Dělení programovacích jazyků 1

Nižší programovací jazyky

jsou jazyky primitivní, jejichž instrukce odpovídají příkazům procesoru. To znamená, že procesor bude vykonávat ty instrukce, které programátor napíše. Jsou závislé na svém procesoru a nepřenositelné na jiný procesor.
V praxi to vypadá tak, že programátor musí vypisovat vše. Pak i jednoduchý program má neúměrně složitý zdrojový kód. Výhodou je, že programátor má takto přístup i k funkcím počítače, které by měl ve vyšším programovacím jazyce nedosažitelné. Lépe tedy využije jeho schopnosti.

Patří sem:

- strojový kód (to, co uvidíte, když otevřete obsah „exe“ souboru v textovém editoru)
- jazyk symbolických adres (Assembler) – je velice blízký strojovému kódu

Dělení programovacích jazyků 2

Vyšší programovací jazyky

jsou podstatně srozumitelnější, jejich struktura je logická, nejsou závislé na strojových principech počítače. Do strojového kódu se převádějí kompilátorem (případně se rovnou spouštějí interpretrem).

V praxi je vyšší programovací jazyk vše, co není Assembler (například jazyk C++, Pascal, Basic, Delphi..)

Dělení programovacích jazyků 3

Programovací jazyky dále dělíme na :

- **kompilované**
- **interpretované**

Kompilované jazyky

jsou nejdříve celé přeloženy a až potom mohou být spuštěny. Jsou rychlejší, mají vyšší nároky na formální správnost kódu. Překládají se kompilátorem, výsledkem překladu je (většinou) .exe soubor. Patří sem většina klasických programovacích jazyků. Teoreticky může mít jeden programovací jazyk verzi jak interpretovanou, tak i kompilovanou.

Interpretované jazyky

jsou překládány až za běhu programu. Jsou pomalejší, ale nemají tak velké požadavky. Překládají se interpretrem, ten instrukce zároveň při překladu provádí. Hlavní nevýhodou těchto jazyků je, že se musejí vždy spouštět v interpretru. Do této skupiny patří například Basic, skriptovací jazyky (PHP, Python,Perl ...).

Dělení programovacích jazyků 4

Další způsoby dělení programovacích jazyků

Programovací jazyky můžeme dále dělit na
jazyky, které podporují: programování strukturované (např. Pascal)

objektově orientované programování (OOP) – např. Visual Basic, Delphi