Algoritmizace

Algoritmizace

- Proces transformace zadaného problému či úkolu do podoby algoritmu
- Ověření a testování výsledného algoritmu

Základní postupy tvorby algoritmu

- Vždy vycházíme ze schopností systému, pro který algoritmus tvoříme – popisný jazyk (jiný pro člověka nebo stroj)
  - nemusí jít nutně o IT problém
- Snaha použít části algoritmu opakovaně
  - parametrizace algoritmu zadáním vstupních hodnot
  - funkce - činnosti, jejichž výstupem je nový údaj
  - procedury - činnosti bez výstupních údajů, jejich činnost spočívá v ovládání systému
- Ze základních prvků tvoříme složitější konstrukce

Kroky algoritmizace

- Specifikace úkolu
  - co chci řešit a vyřešit
  - co má být výstupem a jaké jsou potřeba vstupy?
  - jaké další předpoklady (omezení) platí?
  - pohled zvenčí
    - algoritmus je v tuto chvíli pouze černou skřínkou
  - příklad
    - mám X Kč a chci mít mobil
    - chci jej koupit v ČR

- Analýza problému
  - jak lze úlohu řešit?
  - koncepční nástin cesty, jak dosáhnout cíle
  - může pomoci dekompozice – rozdělení na dílčí úlohy
  - požadavky na informace a data
  - příklad
    - mobil má umět X funkcí
    - projdu nabídku vybraných e-shopů a vyberu si nejlepší variantu
    - v každém e-shopu provedu výběr

- Vytvoření algoritmu
  - základem schopnosti autora a znalost použitelných příkazů pro cílový systém
  - obvykle návrh shora dolů
    - postupná dekompozice od úkolu k řešení jeho částí
  - návrh zdola nahoru
    - postupné skládání řešení ze základních stavebních prvků (kroků)
    - automatizované postupy
  - příklad
    - konkrétní zápis algoritmu zvoleným způsobem (např. PS Diagram)
    - obtížnější, než se zdá – nutná přesnost

- Testování algoritmu
  - je to správně zapsáno?
    - syntaktické chyby
  - řeší to zadanou úlohu?
    - chyby analýzy – špatně pochopený úkol
    - chyby vstupních dat – špatná, nepřesná nebo nedostupná
  - obvykle náročné
    - „ruční práce“
    - vhodné použít nějakou podporu (PS Diagram)

- Implementace algoritmu – vložení do cílového systému (program)
  - transformace algoritmu do podoby srozumitelné pro cílový systém – např. do programu
  - možné problémy s cílovým systémem
    - nedostatek schopností systému (nebo jiné než jsme čekali),
    - technická nebo např. matematická omezení při výpočtu (0<>0), atd.

- Užití algoritmu
  - může vyžadovat údržbu
    - vnější vlivy – jiné prostředí
    - vnitřní změny – upgrade schopností systému

Algoritmické postupy

- Obecné, nezávislé na konkrétním systému
- Obvykle sekvence (série) kroků, které se mají v daném pořadí provést
  - kroky navazují na předchozí a vytváří předpoklady pro následující
  - lze i hierarchické uspořádání - sekvence algoritmů řešících zvolenou část úlohy
  - příklad: chci jet vlakem do Prahy do divadla
- Dnes i paralelizmus – mimo tento kurz

Algoritmizace problému

- Přímý postup
  - triviální problém, známý postup, dostatek znalostí
- Přeformulování problému
  - zjednodušení, zobecnění, ekvivalentní přeformulování, parametrizace
- Rozklad problému na podproblémy
  - konjunktivní: řešení problému řešením všech podproblémů
  - disjunktivní: řešení problému řešením pouze jednoho z podproblémů
  - repetiční: opakované řešení podproblémů (iterační rozklad) nebo řešení téhož problému se zmenšující se dimenzí (rekurzivní rozklad)

Přeformulování problému

- Zjednodušující přeformulování
  - místo výrazu πr^2 vyhodnocujeme výraz $3{,}14 \cdot r^2$
- Ekvivalentní přeformulování
  - soustavu *n* lineárních rovnic o *n* neznámých postupně transformujeme ekvivalentními úpravami (násobení rovnice nenulovým číslem, přičtení rovnice k jiné rovnici) na soustavu s jednotkovou maticí, což je triviální problém
- Zobecňující přeformulování
  - místo problému nalezení kořenů rovnice $2x^2 - 6x + 3 = 0$ algoritmizujeme problém $ax^2 + bx + c = 0$

Rozklad problému

- Postupný návrh algoritmu rozkladem problému na podproblémy
  - zadaný problém rozložíme na podproblémy (procedury, funkce)
  - pro řešení podproblémů zavedeme abstraktní příkazy
  - s pomocí abstraktních příkazů sestavíme hrubé řešení
  - k abstraktním příkazům následně přistupujeme jako k samostatným algoritmům a popisujeme je standardními příkazy

Rozklad problému

- Konjunktivní rozklad
  - aritmetický průměr *n* čísel získáme postupným řešením těchto dvou podproblémů:
    - výpočet součtu zadaných čísel
    - dělení součtu hodnotou n
- Disjunktivní rozklad
  - kořeny kvadratické rovnice získáme v závislosti na hodnotě diskriminantu řešením jednoho z těchto dvou podproblémů:
    - výpočet reálných kořenů
    - výpočet komplexních kořenů

Rozklad problému

- Iterační rozklad (v kombinaci s konjunktivním)
  - výpočet součtu čísel $x_1, x_2, \ldots, x_N$:
    - konjunktivní rozklad - vynulování proměnné S (S = 0)
    - iterační rozklad - pro $I = 1, 2, \ldots, N$ postupně sečítáme hodnoty S a x_I a výsledek ukládáme do S ($S = S + x_I$)
- Rekurzivní rozklad (v kombinaci s disjunktivním)
  - výpočet faktoriálu F přirozeného čísla N ($F = N!$):
    - disjunktivní rozklad:
    - je-li $N = 0$, pak $F = 1$
    - je-li $N > 0$, pak $F = N * (N - 1)!$ … rekurzivní rozklad
- Hierarchický rozklad – procedury a funkce
  - opakované užití v algoritmu

Vazba na algoritmické struktury

- Operační krok
  - přiřazení – A=A+1
  - vstup / výstup - READ (odkud, prom), PRINT (kam, „pokusný text“)
- Sekvence (odpovídá konjunktivnímu rozkladu)
- Větvení (odpovídá disjunktivnímu rozkladu)
- Iterace, cyklus (odpovídá iteračnímu rozkladu)
  - s daným počtem opakování
  - s testem zahájení, s testem ukončení

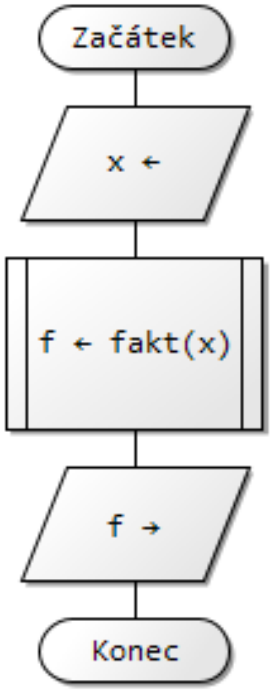
Hra NIM

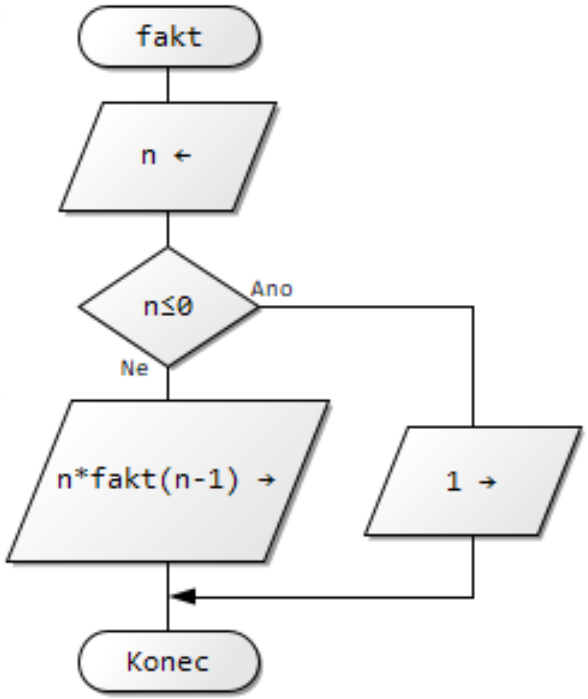
- Rozklad problému na podproblémy ilustrujme na příkladu hry NIM
  - Pravidla:
    - hráč zadá počet zápalek (např. od 15 do 35)
    - pak se střídá se strojem v odebírání; odebrat lze 1, 2 nebo 3 zápalky,
    - prohraje ten, kdo odebere poslední zápalku.
  - Dílčí podproblémy:
    - zadání počtu zápalek
    - odebrání zápalek hráčem
    - odebrání zápalek strojem

- Pravidla pro odebírání zápalek strojem, která vedou k vítězství (je-li to možné):
  - počet zápalek nevýhodných pro protihráče je *1*, *5*, *9*, atd., obecně $4n+1$, kde $n >= 0$,
  - stroj musí z počtu p zápalek odebrat x zápalek tak, aby platilo $p - x = 4n + 1$
  - z tohoto vztahu po úpravě a s ohledem na omezení pro x dostaneme $x = (p - 1) \bmod 4$ (*mod* je zbytek po dělení tj. operace *%*)
  - vyjde-li $x=0$, znamená to, že okamžitý počet zápalek je pro stroj nevýhodný a bude-li protihráč postupovat správně, stroj prohraje.

Rekurzivní algorimus

- Rekurzivní algoritmus v některém kroku volá sám sebe
- Rekurzivní metoda v některém příkazu volá sama sebe (i nepřímo)
- Příklad: faktoriál



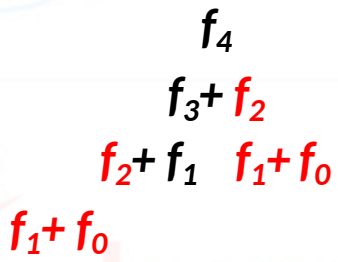


Obecně k rekurzivitě

- Rekurzivní algoritmus předepisuje výpočet "shora dolů" v závislosti na velikosti (složitosti) vstupních dat:
  - pro nejmenší (nejjednodušší) data je výpočet předepsán přímo
  - pro obecná data je výpočet předepsán s využitím téhož algoritmu pro menší (jednodušší) data
- Výhodou rekurzivních metod je jednoduchost a přehlednost
- Nevýhodou může být časová náročnost způsobená např. zbytečným opakováním výpočtu

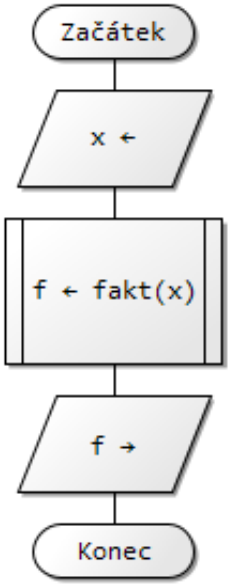
Příklad: Fibonacciho posloupnost {0,1,1,2,3,5,8,13,21,…}

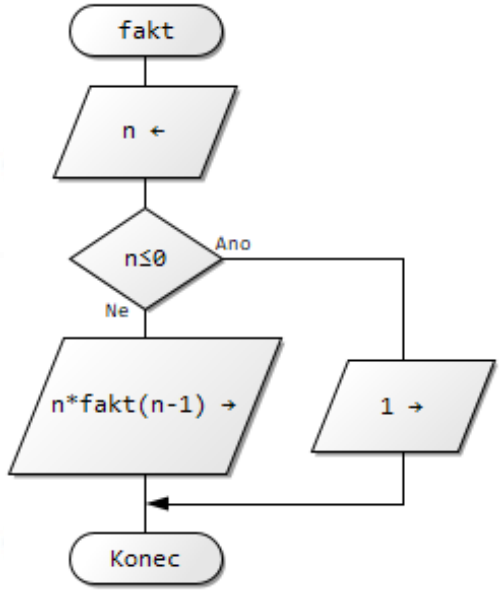
$$f_0 = 0$$
$$f_1 = 1$$
$$f_i = f_{i-1} + f_{i-2} \text{ pro } i > 1$$

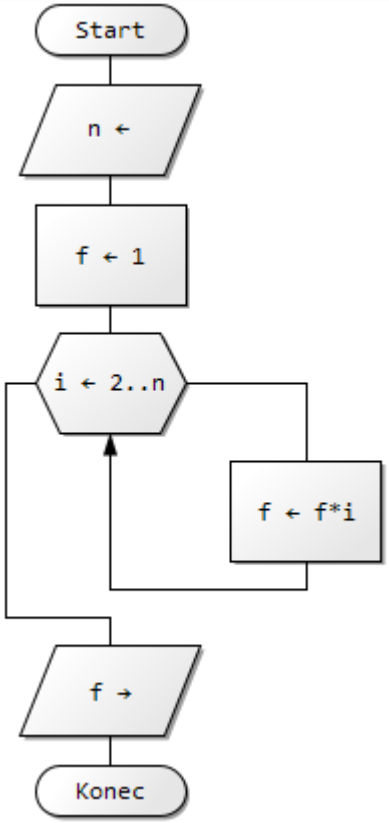


Od rekurze k iteraci

- Řadu rekurzívních algoritmů lze nahradit iteračními, které počítají výsledek "zdola nahoru", tj, od menších (jednodušších) dat k větším (složitějším)







- Pozn.: Rekurzivitu lze odstranit pomocí tzv. zásobníku

Problémy algoritmizace

- Rámec (rozsah) popisovaného problému
  - co všechno do algoritmu zahrneme?
  - algoritmus může být velmi jednoduchý, pokud nebereme v úvahu všechny možné (i málo pravděpodobné) situace
    - pravidlo 20 / 80
  - optimální je minimální rozsah naplňující uspokojivě (např. s určitou pravděpodobností či kvalitou) daný cíl

- Náročnost a efektivita
  - cíle se často dá dosáhnout různými způsoby
    - příklad: dřevorubec
  - nároky na zdroje
    - čas – drobnou úpravou algoritmu může být řešení výrazně zkráceno
    - lidé - algoritmy nejsou jen pro stroje
    - energie – souvisí s dobou řešení
    - finance – jiné promítnutí energií
    - vstupy – různá cena informace
    - paměť – nutná kapacita

Jiné pohledy na algoritmizaci

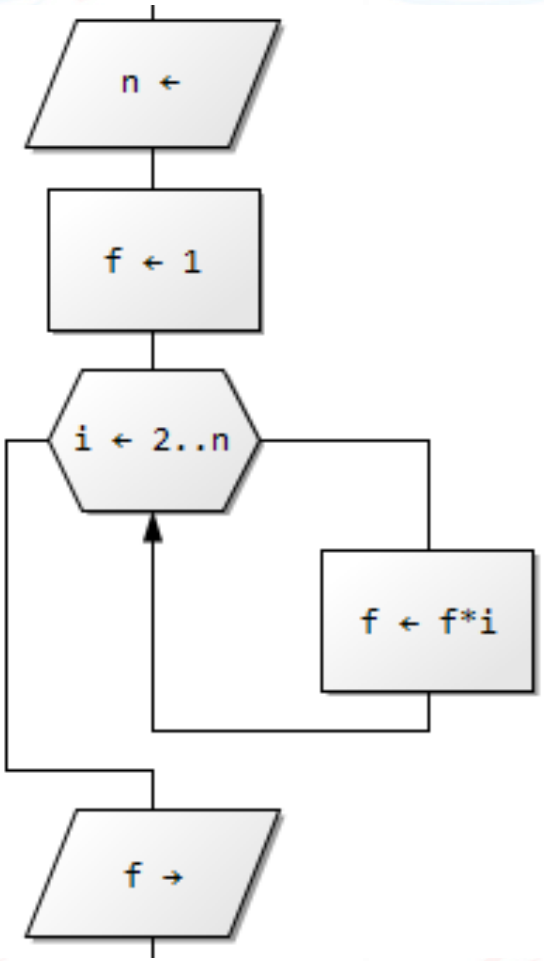
- Na tvorbu algoritmu může být pohlíženo i jako na hru mezi autorem a daným úkolem.
- Autor vybírá své tahy ze sady příkazů podle známých pravidel s cílem vytvořit sekvenci, která povede k vítězství (vyřešení úlohy).
- Tvorba algoritmu je i prohledávání stavového prostoru.
- Prostor možností (stupeň volnosti systému) je obrovský (při zahrnutí výběru proměnných → ∞ variant), výrazně větší než u šachů.

Složitost algoritmů

Časová složitost algoritmů

- Důležitou vlastností algoritmu je časová náročnost výpočtů podle daného algoritmu
  - nezískává se měřením doby výpočtu pro různá data, ale analýzou algoritmu, jejímž výsledkem je stanovení časové složitosti algoritmu
- Časová složitost algoritmu vyjadřuje závislost času potřebného pro provedení výpočtu na rozsahu (velikosti) vstupních dat
- Čas se měří počtem provedených operací, přičemž trvání každé operace se chápe jako bezrozměrná jednotka

Příklad - faktoriál



1

1

1 + (n-1)

n

1

$C(n) = 1 + 1 + 1 + n - 1 + n + 1$

$C(n) = 3 + 2n$

Časová složitost algoritmů

- Doba výpočtu obvykle nezávisí jen na rozsahu vstupních dat, ale též na konkrétních hodnotách
- Obecně proto rozlišujeme časovou složitost v nejlepším, nejhorším a průměrném případě

- Analýza
  - nejlepší případ: n = 1
    Cmin(n) = 5
  - nejhorší případ: obecný počet prvků *n*
    Cmax(n) = 3 + 2n
  - průměrný případ:
    Cprum(n) = 4 + n

Časová složitost algoritmů

- Přesné určení počtu operací při analýze složitosti algoritmu bývá velmi problematické
  - problém s určením průměru - uvádí jen nejhorší případ
- Zpravidla nás nezajímají detaily, ale tendence růstu při zvětšujícím se *n*
  - výrazy udávající složitost lze pak zjednodušit: stačí uvažovat pouze složky s nejvyšším řádem růstu a i u nich lze zanedbat multiplikativní konstanty
- Řád růstu časové složitosti výpočtu faktoriálu je ***n*** - časová složitost je lineární
- Časovou složitost definujeme pomocí tzv. asymptotické složitosti *O()*

Časová složitost algoritmů

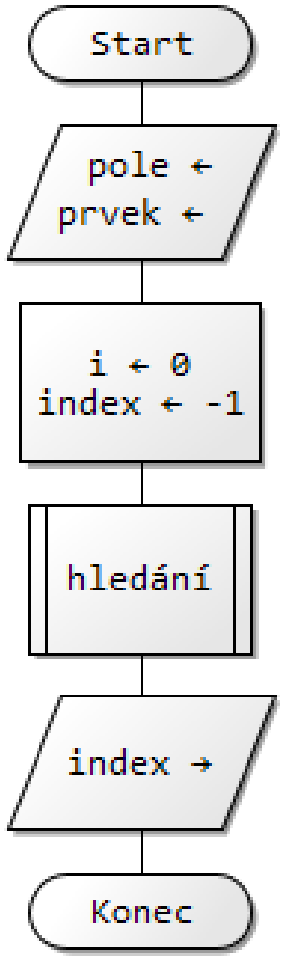
Tabulka udávající dobu výpočtu pro různé časové složitosti za předpokladu, že *1*operace trvá *1* μs a pro rostoucí rozsahy dat ***n***

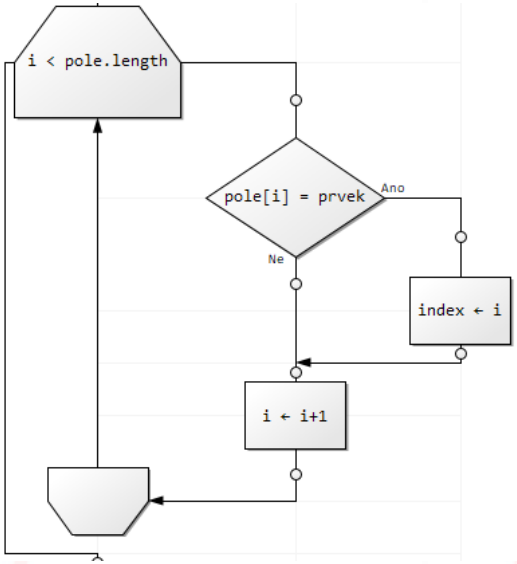
O() \ n	10	20	40	60	500	1000
$log_2\ n$	3,3μs	4,3μs	5,3μs	5,9μs	9μs	10μs
n	10μs	20μs	40μs	60μs	0,5ms	1ms
$n\ log_2\ n$	33μs	86μs	0,2ms	0,35ms	4,5ms	10ms
n^2	0,1ms	0,4ms	1,6ms	3,6ms	0,25s	1s
n^3	1ms	8ms	64ms	0,2s	125s	17min
n^4	10ms	160ms	2,56s	13s	17h	11,6dní
2^n	1ms	1s	12,7 dní	36000 let		
$n!$	3,6s	77000 let				

Z tabulky vyplývá, že algoritmy s horší než polynomiální složitostí jsou prakticky neproveditelné – tzv. NP úplné (nedeterministicky polynomiální), např. problém obchodního cestujícího tj. nalezení nejkratší cesty, která prochází právě jednou všemi místy ze zadané množiny (permutace uzlů grafu – $C(n!)$)

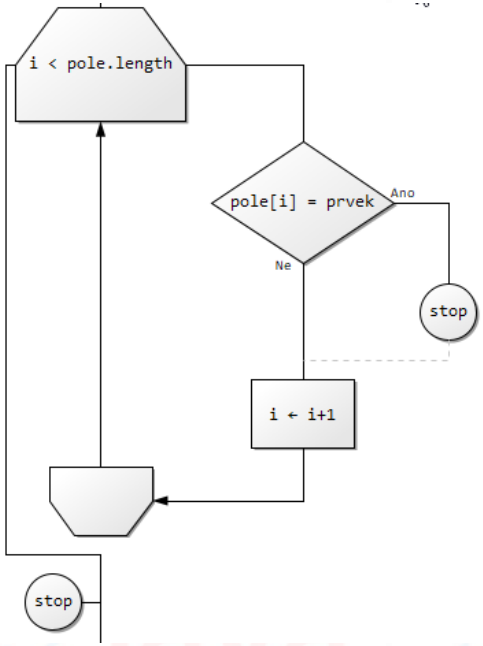
Hledání v poli

- Klasická ukázka různé složitosti algoritmů pro stejný úkol
- Sekvenční hledání v poli lze urychlit pomocí předčasného ukončení cyklu a dále pomocí zarážky
- Cyklus budeme provádět do prvního výskytu hledané hodnoty
  - s tím spojena určitá omezení
- Za předpokladu, že pole není zaplněno až do konce, uložíme do prvního volného prvku hledanou hodnotu a cyklus pak může být řízen jedinou podmínkou

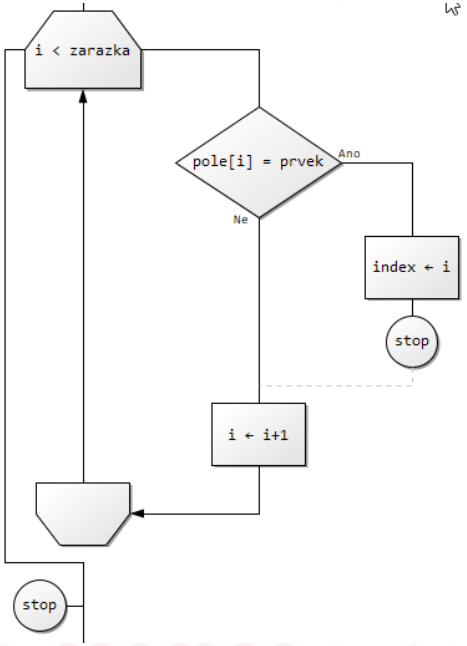




klasika



přerušení cyklu



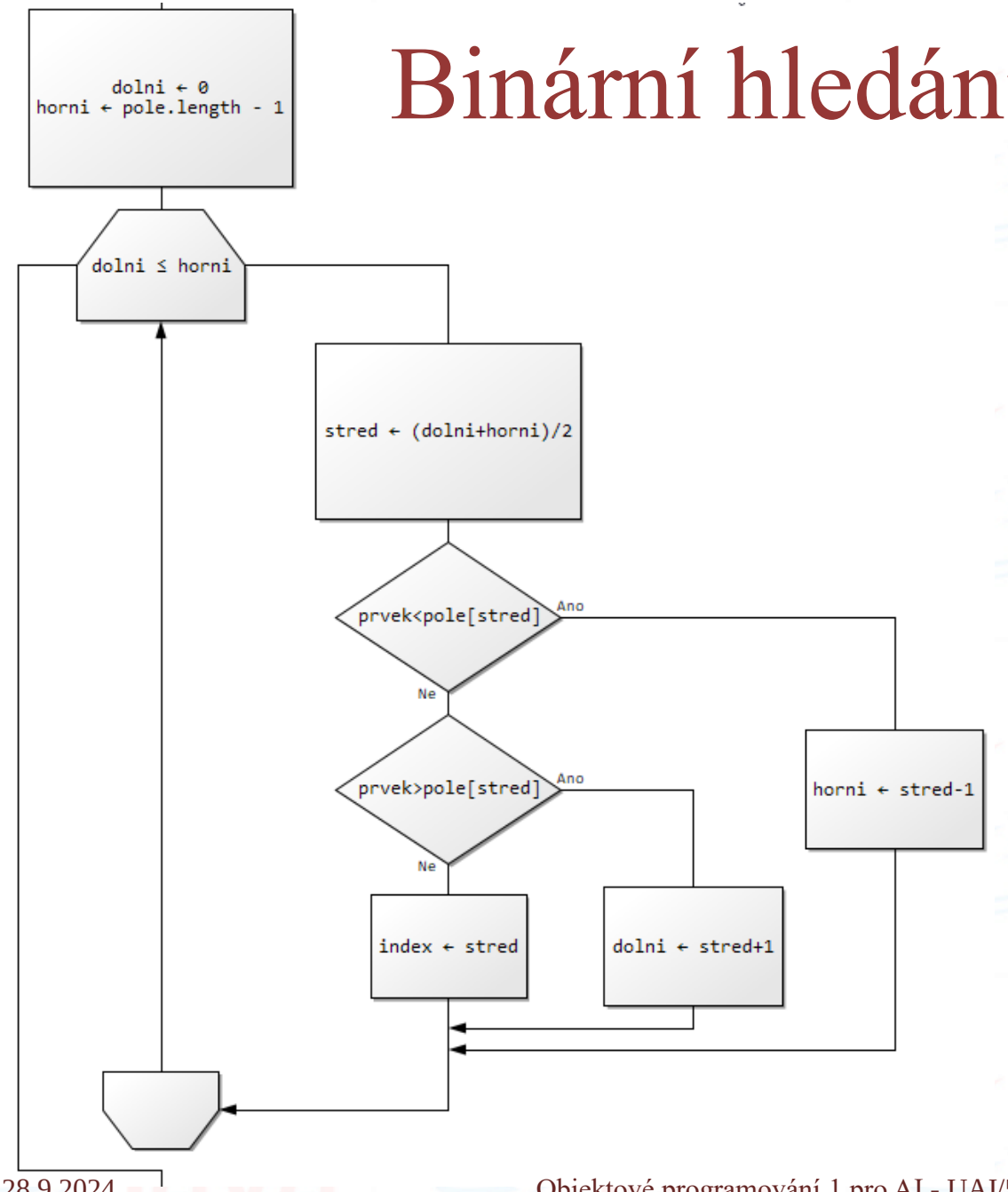
přerušení cyklu se zarážkou

Ušetříme část průchodů cyklem, avšak časová složitost zůstane ***O(n)*** a nejde tedy o významné urychlení

Princip půlení intervalu

- Pro některé problémy lze sestavit algoritmus založený na principu opakovaného půlení intervalu
  - opakovaně se zmenšuje rozsah dat na polovinu
  - časová složitost takového algoritmu je logaritmická (dělíme-li n opakovaně 2, pak po $\log_2 n$ krocích dostaneme číslo menší nebo rovno 1)
- Při hledání prvku pole lze použít princip opakovaného půlení v případě, že pole je seřazené, tj. hodnoty jeho prvků tvoří monotónní posloupnost
  - zjistíme hodnotu y prvku ležícího uprostřed zkoumaného úseku pole
  - je-li hledaná hodnota $x = y$, je prvek nalezen
  - je-li $x < y$, budeme hledat v levém úseku
  - je-li $x > y$, budeme hledat v pravém úseku
- Takovéto hledání se nazývá též binární hledání (binary search), časová složitost je $O(\log_2 n)$
  - základ logaritmu ale není pro určení složitosti podstatný

Binární hledání



Řazení pole

- Algoritmy řazení pole jsou algoritmy, které přeskupí prvky pole tak, aby upravené pole bylo seřazené
  - pole *p* je vzestupně seřazené, jestliže platí:
    p[i-1] <= p[i] *pro i = 1..počet prvků pole – 1*
  - pole p je sestupně seřazené, jestliže platí:
    p[i-1] >= p[i] *pro i = 1.. počet prvků pole – 1*

- Následující metody vzestupně řadí všechny prvky pole daného parametrem:

bubbleSort(...)	řazení zaměňováním
selectSort(...)	řazení výběrem
insertSort(...)	řazení vkládáním
mergeSort(...)	řazení slučováním

Bublinkové řazení (BubbleSort)

- Při řazení zaměňováním postupně porovnáváme sousední prvky a pokud jejich hodnoty nejsou v požadované relaci, vyměníme je; to je třeba provést několikrát - prohledávaný interval se postupně zkracuje zprava

- Pseudokód řešení

```
// největší prvek na pravý konec intervalu,
pak zkrácení intervalu o 1
a znovu  for (n=pole.length-1; n>0; n--)
for (i=0; i<n; i++)
if (pole[i]>pole[i+1]) "vyměň pole[i] a pole[i+1]"
```

- Časová složitost algoritmu BubbleSort je $O(n^2)$

Výběrové řazení (SelectSort)

- Při řazení výběrem se opakovaně hledá nejmenší prvek
  - prohledávaný interval se postupně zkracuje zleva

- Pseudokód řešení

```
for (i=0; i<pole.length-1; i++) {
   "najdi nejmenší prvek mezi
   pole[i] až pole[pole.length-1]";
 "vyměň hodnotu nalezeného prvku s
 pole[i]";
}
```

- Časová složitost algoritmu SelectSort: $O(n^2)$

Řazení vkládáním (InsertSort)

- Pole je řazeno opakovaným vkládáním prvku do již seřazeného úseku pole s posunem větších prvků vpravo

- Psedudokód řešení

```
for (n=1; n<pole.length; n++) {
 "úsek pole od pole[0] do pole[n-1] je seřazen"
 "vlož do tohoto úseku délky n hodnotu pole[n]"
}
```

- Časová složitost algoritmu InsertSort je $O(n^2)$

Slučování (merging)

- Problém slučování lze obecně formulovat takto:
  - ze dvou seřazených (monotónních) posloupností ***a*** a ***b*** máme vytvořit novou posloupnost obsahující všechny prvky z ***a*** i ***b***, která je rovněž seřazená
- Příklad

  a: 2 3 6 8 10 34
  b: 3 7 12 13 55
  výsledek: 2 3 3 6 7 8 10 12 13 34 55

- Neefektivní řešení
  - vytvoříme pole, do něhož zkopírujeme prvky *a*, pak *b*, a pak seřadíme
  - ale to není slučování!
- Princip slučování
  - postupně porovnáváme prvky zdrojových posloupností a do výsledné posloupnosti přesouváme menší z nich
  - nakonec zkopírujeme do výsledné posloupnosti zbytek první nebo druhé posloupnosti

Řazení slučováním (MergeSort)

- Efektivnější algoritmy řazení mají časovou složitost ***O(n.log$_2$ n)***
- Např. řazení slučováním,
  - založen na opakovaném slučování seřazených úseků do úseků větší délky
- Lze jej popsat rekurzívně:
  - řazený úsek pole rozděl na dvě části
  - seřaď levý úsek a pravý úsek
  - přepiš řazený úsek pole sloučením levého a pravého úseku