**Základní vlastnosti algoritmů**

* Konečnost (finitnost)
* Obecnost (hromadnost, masovost, univerzálnost)
* Determinovanost
* Výstup (resultativnost)
* Vstup
* Efektivita

**Konečnost**

Každý algoritmus musí skončit v konečném počtu kroků. Tento počet kroků může být libovolně velký (podle rozsahu a hodnot vstupních údajů), ale pro každý jednotlivý vstup musí být konečný. Postupy, které tuto podmínku nesplňují, se mohou nazývat výpočetní metody. Speciálním příkladem nekonečné výpočetní metody je reaktivní proces, který průběžně reaguje s okolním prostředím. Někteří autoři však mezi algoritmy zahrnují i takovéto postupy.

**Obecnost**

Algoritmus neřeší jeden konkrétní problém (např. „jak spočítat 3×7"), ale obecnou třídu obdobných problémů (např. „jak spočítat součin dvou celých čísel"), má širokou množinu možných vstupů.

**Výstup**

Algoritmus má alespoň jeden výstup, veličinu, která je v požadovaném vztahu k zadaným vstupům, a tím tvoří odpověď na problém, který algoritmus řeší (algoritmus vede od zpracování hodnot k výstupu).

**Vstup**

**Efektivita**

To nás bude zde velmi zajímat. Je to jedno z měřítek kvality algoritmů (efektivita, účinnost) – jaký je vztah mezi počtem kroků (časem provedení) a velikostí vstupu (instance). Jaký je vztah mezi velikosti paměti a dalšími sledovanými hodnotami, jaká je síťová složitost (komunikační) v závislosti na dalších veličinách? Lze to nějak vyjádřit jednoduchým vztahem? Dá se to „zrychlit“?

**Klasifikace algoritmů**

**Rekurzivní**

* Funkce volá sama sebe
* Využívá se například pro definici přirozených čísel, stromových struktur a některých funkcí.
* Musí mít podmínku ukončení, jinak funkce pojede do nekonečna
* programovací jazyk musí umožňovat volání podprogramu ještě před ukončením jeho předchozího volání.
* Po každém kroku volání sebe sama musí dojít ke zjednodušení problému. Pokud nenastane koncová situace, provede se rekurzivní krok
* Každá rekurze jde přepsat pomocí for loopu

**Dělení rekurze**

**První dělení:**

* Přímá rekurze nastává, pokud podprogram volá přímo sám sebe.
* Nepřímá (vzájemná) rekurze je situace, kdy vzájemné volání podprogramů vytvoří „kruh". Např. v příkazové části funkce A je volána funkce B, ve funkci B voláme funkci C, která volá zpět funkci A (a kruh se uzavřel)

**Druhé dělení:**

* Lineární rekurze nastává, pokud podprogram při vykonávání svého úkolu volá sama sebe pouze jednou. Vytváří se takto lineární struktura postupně volaných podprogramů.
* Stromová rekurze nastává, pokud se funkce nebo procedura v rámci jednoho vykonání svého úkolu vyvolá vícekrát. Vzniklou strukturu je možné znázornit jako strom. Pro dvě volání v jednom průchodu vzniká binární strom, pro tři ternární strom, atd. Při kombinaci zmíněných typů rekurze lze docílit velmi komplikovaných struktur (již z charakteru takovéhoto volání podprogramů nelze docílit jiné, než symetrické struktury)

**Rozděl a panuj (Divide and Conquer)**

* Algoritmy pro práci s daty, které řeší problém rozdělením řešené úlohy na dílčí části (podproblémy), nad kterými se provádí algoritmická operace.
* Metoda se implementuje rekurzivně nebo iterativně.
* Typickými představiteli metody rozděl a panuj jsou algoritmy třídění MergeSort, QuickSort, nebo výpočet rychlé Fourierovy transformace (FFT).

**Základní myšlenka:**

Problém je rozdělen na několik menších, ale podobných dílčích problémů (často stejné velikosti nebo přibližně stejné velikosti). Tyto dílčí problémy jsou řešeny rekurzivně, a poté jsou jejich řešení kombinována, aby se získalo řešení původního problému.

**Struktura:**

1) Rozdělení: Rozdělí problém na dvě nebo více částí (obvykle na poloviny).

2) Řešení dílčích problémů: Každý dílčí problém se řeší rekurzivně.

3) Kombinace: Kombinují se výsledky, aby vzniklo konečné řešení.

**Sniž a vyřeš (Decrease and Conquer)**

* Problém je redukován na menší problém menší velikosti
* Algoritmy jako Insertion Sort nebo Binary Search, kde je problém postupně zmenšován a řešen. Například při binárním vyhledávání je zmenšováno hledané pole o polovinu.

**Základní myšlenka**:

Problém je redukován na menší problém menší velikosti, obvykle jen o jednu nebo konstantní část, a tento menší problém je řešen přímo nebo rekurzivně. Poté se řešení menšího problému použije k vyřešení původního problému.

**Struktura**:

1) Redukce problému: Zmenší problém (obvykle jen o jednu jednotku).

2) Řešení zmenšeného problému: Tento zmenšený problém je řešen (rekurzivně nebo přímo).

3) Kombinace: Výsledek se použije k dosažení řešení původního problému.

**Algoritmy dynamického programování**

* Dynamické programování se používá pro problémy, které mají překrývající se podproblémy a optimální podstrukturu.
* Příklad: Fibonacciho posloupnost, kde by rekurzivní přístup počítal stejnou hodnotu mnohokrát, zatímco dynamické programování počítá každou hodnotu jen jednou a uloží ji pro další použití.

**Základní myšlenka:**

Místo toho, aby se řešily ty samé dílčí problémy vícekrát (jako v rekurzivních algoritmech), dynamické programování je řeší jen jednou a jejich řešení ukládá pro pozdější použití (tzv. memoizace nebo tabulkování).

**Struktura:**

1) Rozdělení problému: Podobně jako u „Rozděl a panuj“ je problém rozdělen na menší dílčí problémy, ale s tím rozdílem, že se tyto problémy často opakují.

2) Uložení mezivýsledků: Dynamické programování ukládá výsledky dílčích problémů do tabulky (například pole nebo matici), aby se zamezilo opakovaným výpočtům.

3) Využití mezivýsledků: Při řešení větších podproblémů se využívají uložené výsledky menších podproblémů, což zvyšuje efektivitu.

**Rozdíly**

* Rozděl a panuj: Rozdělí problém na několik nezávislých podproblémů, které řeší samostatně a jejichž řešení následně kombinuje.
* Sniž a vyřeš: Snižuje problém na menší verzi sebe sama (obvykle o jeden krok) a řeší jej postupně.
* Dynamické programování: Řeší problémy, které mají překrývající se podproblémy, a ukládá dílčí řešení, aby se předešlo opakovanému výpočtu.

**Optimalizace a čas**:

* Dynamické programování je optimalizováno pro problémy, kde se dílčí problémy opakují. Tím šetří čas uložením výsledků a opakovaným použitím již vypočtených dílčích problémů.
* Rozděl a panuj a Sniž a vyřeš neřeší podproblémy opakovaně, ale nezabývají se optimalizací ukládání dílčích výsledků. Pokud se podproblémy opakují, musí být vyřešeny znovu, což může vést k horší časové složitosti.

**Ukládání výsledků**:

* Dynamické programování aktivně ukládá výsledky dílčích problémů (memoizace nebo tabulkování) pro pozdější použití.
* Rozděl a panuj a Sniž a vyřeš obvykle neukládají mezivýsledky – každý podproblém je řešen na místě, což může vést k opakovaným výpočtům.

**Struktura dat:**

* Pokud mají dílčí problémy tendenci se opakovat, dynamické programování je efektivnější. •
* Rozděl a panuj a Sniž a vyřeš jsou vhodnější tam, kde lze problém efektivně rozdělit na nezávislé nebo neopakující se dílčí problémy

**Nedeterministické algoritmy**

* Pravděpodobnostní algoritmy (někdy též probabilistické, nebo stochastické) provádějí některá rozhodnutí náhodně či pseudonáhodně.
* Snaží se najít řešení rychleji nebo řešení těžko řešitelných problémů (NP, NPC problémů – viz budoucí lekce).

**Základní myšlenka**:

* Pravděpodobnostní algoritmus se může náhodně rozhodovat mezi různými možnostmi, jak pokračovat.
* Pro stejný vstup může dávat takový algoritmus různé výsledky, které mohou být dokonce nesprávné. Mnohdy se tedy na daném vstupu spustí pravděpodobnostní algoritmus vícekrát, aby se s větší pravděpodobností dospělo ke správnému výsledku.

**Možné varianty** (ne všechny):

1) Výpočetní strom je binární, v každém uzlu se provede hod mincí.

2) V každém výpočetním uzlu je definováno pravděpodobnostní rozložení na hranách.

3) Na začátku se vybere náhodně deterministický algoritmus, který provede výpočet (nebo místo startu – vložení dat).

**Hladové (greedy) algoritmy**

* Hladový algoritmus (anglicky “greedy search”) je jedním z možných způsobů řešení optimalizačních úloh v matematice a informatice.
* V každém svém kroku vybírá lokální nejlepší řešení, přičemž existuje šance, že takto nalezne nejlepší řešení globální.

**Základní myšlenka:**

* Hladový algoritmus se uplatní v případě, kdy je třeba z množiny určitých objektů vybrat takovou podmnožinu, která splňuje jistou předem danou vlastnost, a navíc má minimální (případně maximální) ohodnocení.
* Jakmile je učiněno rozhodnutí, již se algoritmus nevrací a rozhodnutí nereviduje! Proto je „hladový“.

**Příklady použití**:

1) Hledání cesty v grafu. Přijedu na křižovatku. Vím kolik je doteď uražená vzdálenost. Mám dvě možnosti. Doleva (5km do nejbližšího města) nebo doprava (3,5 km). Zvolím možnost doprava (doposud ujetá vzdálenost + tato možnost má nejmenší ohodnocení). Pak se ale může stát že v další uzlu je 30km objížďka – již se nevracím na předchozí rozhodovací bod.

2) Balení batohu na trek (kufru na dovolenou). Jak naskládat věci s co minimální hmotností a maximálním přínosem (a kufr/batoh 237 654x “nepřebalovat“?

**Heuristické algoritmy**

* Heuristický algoritmus si za cíl neklade nalézt přesné řešení, ale pouze nějaké vhodné přiblížení.
* Používá se v situacích, kdy dostupné zdroje (např. čas) nepostačují na využití exaktních algoritmů (nebo pokud nejsou žádné vhodné exaktní algoritmy vůbec známy).

**Základní myšlenka:**

* Heuristikou nazýváme postup, který, i když neprobere všechny možnosti, je schopný v některých případech podat výsledek. Mnohdy nutno spustit opakovaně pro jistou garanci dosažení kvalitních výsledků.
* Tyto algoritmy jsou rozvíjeny a často používány pro řešení složitých problémů. Deterministické algoritmy jsou v řešení složitých problémů velmi špatně použitelné, protože jejich náročnost (zejména časová) roste, s lineárně rostoucím rozsahem problému, exponenciálně.
* Velmi často se setkáváme s metodami kombinujícími heuristické algoritmy s deterministickými.
* Nezaměňovat s čist ě nedeterministickými algoritmy.