**BI-SPOL-20 Rekurzivní rozklad problému na podproblémy metodou Rozděl-a-panuj. Rekurze vs iterace. Dynamické programování**

BI-PA1 + BI-AG1

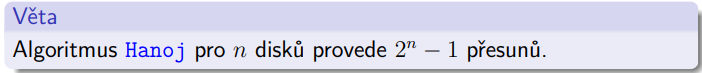
### Rozděl a panuj

* Rekurzivní algoritmus
* Metoda řešení problémů, kdy se problém rozdělí na podproblémy, které se řeší stejným způsobem
* Je to postup řešení problému nad vstupními daty, při kterém
  + se **stejný postup aplikuje na jednu nebo více částí vstupních dat** (čili menší instance problému se řeší stejně jako původní problém)
  + současně se poskytne přímé řešení triviální instance problému
  + řešení celého problému se sestaví z řešení podproblémů
* Řešení dané instance problému se tedy staví na stejném řešení jedné nebo více menších instancí téhož problému (tzn rozdělí se to na menší až triviální a lépe zpracovatelné podproblémy)
* **výhody**:
  + úspornost zápisu kódu,
  + přirozenost (opakování a samopodobnost jsou běžné v přírodě)
  + intuitivnost (explicitní pojmenování toho, co se opakuje v menším)
  + expresivnost (snadné vyjádření složitosti rekurentní rovnicí a ověření korektnosti)
* např. klasické operace s binárními stromy (**BVS**, **AVL**), **binární vyhledávání** **MergeSort**, **QuickSort, DFS, hanojské věže**
* běhové prostředí programovacího jazyka musí zařídit přes systémový zásobník zanořování a návraty z rekurze

**Hanojské věže**

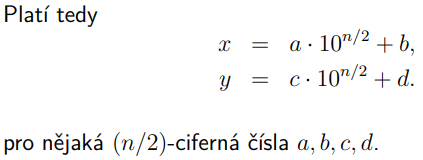
Obsah obrázku text

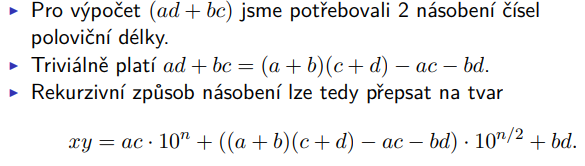
Popis byl vytvořen automaticky



**Karacubův algoritmus**

* rychlejší násobení dvou n-ciferných čísel
* předpokládejme, že n je mocnina dvou
* místo 4 násobení násobíme jen 3 krát
* čísla jsou rozdělená na 2 části stejné délky





* není třeba 4x násobit, stačí 3x + 2x sčítat (což je lineární složitost)
* časová složitost: Θ(n^log3) (oproti O(n^2)) (log3 = cca 1,59)

Obsah obrázku text

Popis byl vytvořen automaticky

**QuickSelect**

* hledání *k*-tého nejmenšího prvku
* postup jako u QuickSortu
* výběr pivota jako skoromediánu (číslo z prostředních dvou čtvrtin seřazené posloupnosti, šance 50%)
* rozdělím na L, S a P část podle pivota
* Obsah obrázku text

  Popis byl vytvořen automaticky

Obsah obrázku text

Popis byl vytvořen automaticky

* Časová složitost záleží na vstupní posloupnosti a volbě pivota
  + Nejhorší případ - Θ().
    - Zvolíme pivota největší prvek na vstupu - |L| = n – 1
    - Pokud navíc bude k = 1, bude se QuickSelect rekurzivně volat na L
    - To se může v nejhorším případě opakovat
  + Pokud zvolíme pivota jako medián bude složitost Θ(n) (najdeme-li ho v lin. čase)
  + Pokud zvolíme pivota jako skoromedián, bude složitost O(n)
    - Ověření, že je to skoromedián trvá Θ(n)
    - Ve střední hodnotě potřebujeme k nalezení skoromediánu 2 pokusy

### Rekurze vs. iterace

#### Rekurze

* rekurzivní funkce je funkce, která v sekvenci příkazů obsahuje svoje volání
  + rekurzivní volání řeší menší (jednodušší) instanci původního problému
  + musí existovat konečná podmínka, kdy rekurze skončí
* rekurzivní funkce jsou přímou implementací rekurzivních algoritmů
  + rekurzivní algo. pracují způsobem „shora dolů“
  + triviální instance problému je implementována přímo
  + rekurzivní volání se použije pro řešení obecné instance problému
* rekurzivní funkce mají sklon k jisté neefektivnosti – například stejné jednodušší problémy mohou být řešeny mnohokrát
* nepřímá rekurze – funkce A volá funkci B, která volá funkci A

př. faktoriál

Obsah obrázku text

Popis byl vytvořen automaticky

#### Iterace

* opakování přes cyklus, který se ukončí splněnou podmínkou, jinak je nekonečná
* iterativní algoritmy pracují „zdola nahoru“ – od instance jednoduchého problému ke komplexnější instanci problému

#### Základní typy rekurze

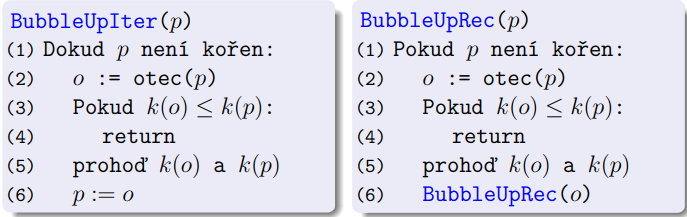
* **koncová**
  + neprovádějí se již další “odložené” operace, snadno převoditelná na iteraci a naopak (např. GCD)
  + rekurzivní volání je posledním příkazem algoritmu, po kterém se už neprovádějí žádné další operace, pouze vrací výsledek. Dané volání se neukládá navíc na zásobník, ale přepisuje existující rámec na zásobníku, proto je výhodnější
* **lineární**
  + jedno nebo více disjunktních rekurzivních volání a provede se jen jedno (např. QuickSelect, BVSFind, BVSMin, BVSInsert, BVSDelete)
  + V těle algoritmu je pouze jedno rekurzivní volání anebo jsou dvě, ale vyskytují se v disjunktních větvích podmíněných příkazů a nikdy se neprovedou současně
  + Strom rekurzivních volání je lineární
* **stromová/kaskádní**
  + má v sobě aspoň dvě rekurzivní volání, které se za určitých okolností provedou, obvyklé pro Rozděl a panuj (např. Fibonacci)
  + V těle algoritmu jsou za sebou aspoň dvě rekurzivní volání, která se za určitých okolností obě provedou. Strom rekurzivních volání je tedy více-ární
  + Obvykle synonymum pro algoritmy Rozděl a panuj

Obsah obrázku text

Popis byl vytvořen automaticky

#### Převod mezi iterací a koncovou rekurzí

* Iterační/podmíněný cyklus se nahradí podmíněným příkazem
* Úprava iterátoru na konci těla cyklu se nahradí rekurzivním voláním na konci podmíněného příkazu



### Dynamické programování

* technika návrhu algoritmů, která je založená na rekurzivním rozkladu problému na podproblémy
* Na rozdíl od klasické metody Rozděl a panuj, metoda DP využívá toho, že se podproblémy během rekurzivního rozkladu **opakují**
* podmínkou použití DP je tedy opakovaný výskyt stejných podproblémů při rekurzivním rozkladu
* DP pak vede na mnohem rychlejší algoritmus než přímočarý rekurzivní rozklad
* Vlastně jde jen o to, že nepočítáme stejné podproblémy vícekrát, ale ukládáme si jejich mezivýsledky – když na něj narazíme znovu, tak už máme výsledek a jedeme dál
  + Pamatujeme si výsledky podproblémů a tak redukujeme počet operací

**Memoizace**

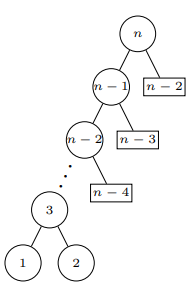
* zapamatování výsledků již spočtených podproblémů, které se mohou použít příště, až algoritmus narazí na stejný podproblém
* výrazně redukuje počet operací k dokončení algoritmu

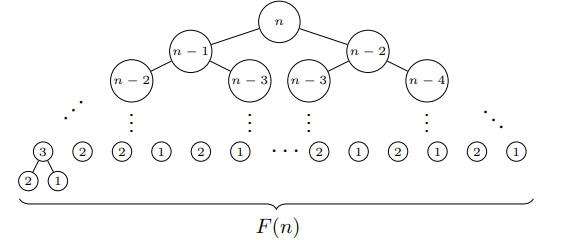
**Princip DP = postup při memoizaci řešení**

* Formulujeme řešení daného problému **rekurzivně rozkladem na řešení podproblémů**.
* Popíšeme řešení pomocí **rekurzivního algoritmu**, který má exponenciální složitost.
* Identifikujeme **opakované** výpočty stejných podproblémů.
* Vytvoříme prázdnou **tabulku** hodnot řešení podproblémů.
* Vložíme do ní hodnoty řešení triviálních instancí.
* Před výpočtem řešení podproblému se podíváme do tabulky.
* Pokud je políčko tohoto podproblému již vypočtené, vezmeme jeho hodnotu.
* Jinak tento podproblém vyřešíme pomocí volání rekurze.
* Po návratu z rekurze zapíšeme výsledek do příslušného políčka tabulky.

#### Fibonacciho čísla

* příklad využití DP
* Výpočet n-2, n-3, atd. se neustále opakuje pro jednotlivé větve
* V DP proto tento podproblém počítáme jen jednou a ukládáme např. do pole.





⇒

DP:

Obsah obrázku text

Popis byl vytvořen automaticky

* Časová složitost je O(n)

Iterativní řešení:

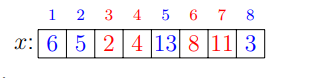
Obsah obrázku text

Popis byl vytvořen automaticky

#### Nejdelší rostoucí posloupnost

* Problém
  + Vstupem je posloupnost x celých čísel x1,…,xn
  + Úkolem je vyškrtnout z ní co nejméně prvků tak, aby zbývající prvky tvořily rostoucí posloupnost
* pro každou buňku si pamatujeme jeho nejdelší posloupnost
* pokud na ní navazujeme, tak už jen přičítáme a nemusíme kontrolovat nic jiného
* přidáme na začátek *-inf*, aby se nám tam uložil nejlepší výsledek
* délu NRP vstupní posloupnosti získáme tak, že zavoláme funkci postupně pro i = 1, …n a vezemem maximum z výsledků
  + lepší řešení je dodefinovat x0 jako nekonečno a zavolat to pouze na to a odečíst 1

Obsah obrázku text

Popis byl vytvořen automaticky

* časová složitost s memoizací je O() – bez ní je to O()

další problémy – editační vzdálenost řetězců, minimální triangulace konvexního mnohoúhelníku – viz přednáška 10