Metody hledání řešení a optimalizace

- Problémy řešené na počítači jsou natolik různorodé, že **neexistuje obecný návod**, jak navrhnout paměťově a operačně **efektivní algoritmus**

- Přesto se používá řada základních přístupů a obratů, které bývají úspěšné u široké třídy algoritmů – označující se jako **programovací metody** či **programátorská paradigmata**

- Většina dále uvedených metod (jedná se pouze o výběr ze všech možných) je součástí metod probíraných v dalších kapitolách

Metoda „rozděl a panuj“

- **Metoda „rozděl a panuj“** patří mezi nejúspěšnější metodu tvorby efektivních algoritmů

- **Princip metody**
  - Nechť $V(n)$ je řešení, které obdržíme aplikací algoritmu **A** na prostor vstupních dat $P(n)$, $n \in \mathbf{N}$
  - Algoritmus **A** pracuje **rekurzivním opakováním kroků**:
    - rozdělení prostoru $P(n)$ na podprostory $P(n_i)$, kde platí $P(n)=\bigcup P(n_i)$ pro $i=1,2,\ldots k$
    - aplikace algoritmu **A** na podprostory $P(n_i)$, tím se získají řešení $V(n_i)$
    - sloučením řešení $V(n_i)$ se získá řešení $V(n)$

- **Předpoklady použití**
  - výsledné řešení $V(n)$ lze skutečně získat sloučením dílčích řešení $V(n_i)$
  - sloučení řešení umíme – sloučení může být netriviální (např. konvexní obálka množiny bodů)

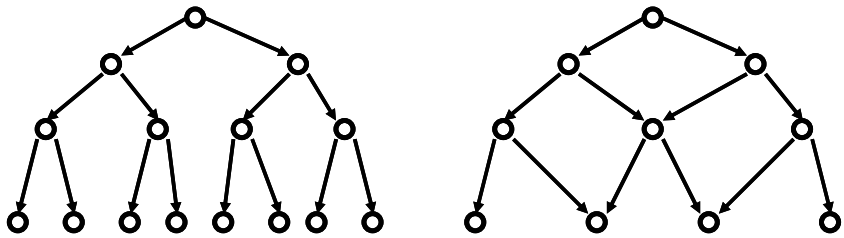
- **Operační a prostorová složitost**
  - použití metody vede většinou k rekurzivnímu zápisu algoritmu – postup „**shora dolů**“
  - složitost odpovídá **složitosti rekurzivního programování** (polynomiální či logaritmická)

- **Příklady použití**
  - algoritmy řazení **Quick-sort**, **Merge-sort**

Tabelační metoda – dynamické programování

- **Princip metody**
  - prostor vstupních dat $P(n)$ je opět rozdělen na dílčí podprostory
  - často se stává, že mnohá z **dílčích řešení problémů jsou stejná**
  - na rozdíl od metody „rozděl a panuj“ se problém v případě tabelační metody řeší **„zdola nahoru“**

- **výsledky dílčích řešení se uchovávají** (tabelují) a vícenásobně se používají při řešení na vyšších úrovních



„rozděl a panuj“ **tabelační metoda**

- **Dynamické programování**
  - mezivýsledky se uchovávají alespoň v **dvojrozměrných datových strukturách** (polích)
  - řešení jsou vázána optimalizační podmínkou – hledáme **optimální řešení dílčích úloh**, jejichž kombinací získáme cílové řešení

- **Operační a paměťová složitost**
  - **operační složitost** je (i výrazně) **nižší** než u metody „rozděl a panuj“
  - **paměťová složitost** je však **vyšší** – uložení tabelovaných hodnot
    - dynamické programování vyžaduje alespoň dvourozměrná pomocná pole

- **Použití metody**
  - častá aplikace **v optimalizačních a grafových úlohách** (hledání optimální cesty apod.), které jinak vedou k algoritmům s exponenciální operační složitostí

Metoda „ořezávej a hledej“

- **Princip metody**
  - metoda „ořezávej a hledej“ **ignoruje části** stavového prostoru (ořezává), **kde řešení neexistuje**.
  - Řešení se hledá ve zbytku prostoru

- **Použití metody**
  - algoritmy hledání ve všech typech **vyhledávacích stromů**
    - každý uzel stromu rozděluje prostor na několik podprostorů – řešení se hledá pouze v jednom z podstromů a ostatní se ignorují

- **Operační a paměťová složitost**
  - vede obyčejně k efektivní operační složitosti za cenu zvýšení paměťové složitosti
  - často vyžaduje **předzpracování** (konstrukce stromu)

Metoda zarážky

- **Princip metody**
  - metoda zarážky umožňuje snížit operační složitost cyklického opakování výpočtu (příkazy **while**, **repeat**) **zjednodušením podmínky** opakování nebo ukončení cyklu
  - tyto podmínky jsou často vázány na splnění dvou **událostí**:
    - u_1 – vyčerpání prostoru cyklu
    - u_2 – nalezení řešení před vyčerpáním prostoru
    - podmínka u_1 bývá obvykle splněna po celou dobu neúspěšného hledání
  - **příklad** - cyklus pro hledání prvku **x** v poli **A** o **n** prvcích

```
while (i<=n) and (A[i]<>x) do i:=i+1;
nalezen:=(i<=n)
```

  - **metoda zarážky** spočívá v tom, že:
    - prostor cyklu **rozšíříme o jeden prvek** (zarážku), který bude představovat neúspěšné řešení
    - v podmínce pokračování cyklu **vynecháme** dílčí podmínku u_1
    - případ neúspěšného řešení (hledání) **ošetříme vně příkazu cyklu**
  - upravený algoritmus:

```
A[n+1]:=x;              {nastavení zarážky}
while (A[i]<>x) do i:=i+1;
nalezen:=(i<(n+1))
```

    - při dostatečně velkém **n** jsme použitím zarážky snížili operační složitost až **o 1/3**

- Metoda je použitelná **pro statické datové struktury** (pole) i **pro dynamické** (spojové seznamy)

Metoda odstranění opakovaných výpočtů

- **Princip metody**
  - v programech se často setkáme se situací, kdy se **opakují výpočty nad stejnými daty**
  - výpočet se provede pouze jednou a výsledek uloží **do pomocné proměnné**, kterou pak opakovaně použijeme

- **Příklady použití**
  - **výpočet**: $q = (\sin x - \cos y) + \sqrt{(\sin x - \cos y)}$
    - *řešení*:

```
POM:=sin(x)-cos(y);
Q:=POM+sqrt(POM);
```

- **skrytá duplicita výpočtu** ukládací (mapovací) funkce při přístupu k prvku pole – `C[i,j]`

```
for i:=1 to n do for j:=1 to n do begin
  C[i,j]:=0;
  for k:=1 to n do C[i,j]:=C[i,j]+A[i,k]*B[k,j]
end;
```

  - indexovanou proměnnou (`C[i,j]`), jejíž indexy jsou invariantní k řídící proměnné cyklu (`k`), nahradíme pomocnou proměnnou (`POM`)

```
for i:=1 to n do for j:=1 to n do begin
  POM:=0;
  for k:=1 to n do POM:=POM+A[i,k]*B[k,j];
  C[i,j]:=POM
end;
```

Metoda předzpracování vstupních dat

- **Princip metody**
  - rozdělíme úlohu na **etapu předzpracování**, kde vstupní data zpracujeme do struktur, nad nimiž budou následné **operace** (etapa výpočtu) **snáze realizovatelné**

  - **běžné varianty předzpracování:**
    - **seřazení vstupních dat** před hledáním hodnoty v těchto datech
    - všechny druhy **vyhledávacích stromů**
    - **technika ohraničujících těles** - tvarově složité geometrické objekty jsou uzavřeny do jednoduchých těles (kvádry u 3D prostoru, obdélníky u 2D)