Zadání: **Techniky návrhu algoritmů. Rekurze, algoritmy prohledávání s návratem, dynamické programování. Zametací technika, metoda „rozděl a panuj“ a „prořezávej a hledej“.**

Základní pojmy

- **Algoritmus** - Výpočetní postup řešení výpočetního problému.
- **Výpočetní problém** - Obecný úkol zpracovat vstupní data na výstupní data, která budou mít zadané vlastnosti (např. vstupní data jsou dvě čísla, výstupní data jsou součet těchto čísel)
- **Korektnost algoritmu** - Pro všechna přípustná vstupní data (všechny instance výpočetního problému) musí korektní algoritmus skončit v **konečném** čase se **správným výstupem** - tedy „Algorimus řeší zadaný problém“. Když se bavíme o algoritmech, téměř vždy uvažujeme korektní algoritmy.
- **Techniky návrhu algoritmů** (Design paradigms) - Základní vzory, jak přistoupit k návrhu algoritmu (např. prohledavání s návratem, dynamické programování, atd.)

Algoritmus

Při návrhu a popisu algoritmů se zabýváme jeho **technikami návrhu** (obecná předloha, jak na to), složitostí a korektností.

Dělení algoritmů

Algoritmy můžeme dělit několika způsoby:

- **Podle implementace**
  - **Rekurzivní/Iterativní** (tato kvalita se uvažuje i v řazení dle techniky návrhu) - Rekurzivní algoritmus volá sám sebe, dokud se nesplní určité podmínky, při iterativním se opakuje určitý blok kódu. Záleží na problému, zda je lépe řešitelný rekurzivně nebo iterativně. Každý rekurzivní algoritmus lze ale přepsat na iterativní (koncovou rekurzi lze dokonce přepsat na iterativní bez použití zásobníku).
  - **Deterministické/Nedeterministické** - Deterministický algoritmus pro stejná vstupní data vykoná vždy naprosto stejné kroky, tudíž výstup bude vždy stejný. Nedeterministický algoritmus může pro stejná data projít různými kroky a mít i různé výsledky. Například při Quick Sortu pro výběr pivotu použiji nedeterministickou heuristiku, která vybere „nějaký“ vhodný.
  - **Sériový/Paralelní/Distribuovaný** - Kroky algoritmu jsou vykonávány po sobě - sériově, současně - paralelně(rozdělení na podúlohy a vykonávání např. ve více vláknech), nebo distribuovaně (paralelizace mezi více počítači). Algoritmy, kterými se zde zabýváme, jsou sériové, ale mnoho z nich je možné paralelizovat.
  - **Exaktní/Aproximativní** - Pro některé problémy nejsou známy exaktní algoritmy, ale využívají se aproximativní algoritmy hledající přibližné řešení blízké opravdovému řešení.
- **Podle techniky návrhu**
- **Podle složitosti a třídy složitosti**

Techniky návrhu

Většina algoritmů využívá ve svém návrhu jeden nebo více z následujících vzorů:

1. Rekurze

Rekurze je metoda zápisu algoritmu, kde se stejný postup opakuje na částech vstupních dat.

V oblasti matematiky pojem rekurze chápeme jako definování objektu pomocí sebe sama. Využívá se například pro definici přirozených čísel, stromových struktur a některých funkcí.

V imperativním programování rekurze představuje opakované vnořené volání stejné funkce (podprogramu), v takovém případě se hovoří o rekurzivní funkci.

Prakticky všechny dnešní VPJ povolují rekurzivní programování.

➢ **Rekurze - pravidla**

1. Musí být definována podmínka pro ukončení rekurze.
2. V každém kroku rekurze musí dojít ke zjednodušení problému.
3. V algoritmu se nejprve musí ověřit, zda nenastala koncová situace. Když ne, provede se rekurzivní krok.

➢ **Rekurze – použití v programování**

a) při definování datové struktury

Datová struktura obsahuje prvek, který má stejnou strukturu jako definovaná datová struktura. Definujeme tak nekonečnou datovou strukturu konečným způsobem (například u dynamických struktur).

b) při vykonávání programu

Obecná úloha se rozkládá na dílčí úlohy, které se vyřeší stejným způsobem. Rozklad zastaví omezující podmínka.

Tato programovací technika se hodí pouze pro určitý omezený okruh úloh, u nichž je rozdělení na menší nezávislé podúlohy stejného charakteru možné a přirozené. Je-li použitelná, vede zpravidla k poměrně krátkému a efektivnímu programu.

➢ **Výhody**

- Úsporná - Rekurze dává možnost definovat nekonečnou množinu objektů konečným příkazem a zápis kódu je kratší a možnost ,
- přirozená: opakování a samopodobnost jsou v přírodě běžné,
- intuitivní: explicite pojmenovává to, co se opakuje v menším,
- expresivní: rekurzivní specifikace umožňuje snadnou analýzu složitosti a ověření správnosti

➢ **Nevýhody**: Rekurzivní volání spotřebovává paměť na zásobníku (pro předání parametrů + návratových hodnot, vytvoření lokálních proměnných) - spotřebovaná paměť je úměrná hloubce stromu rekurze. Dynamická alokace systémového zásobníku a ukládání parametrů na něj navíc představuje časovou režii.

➢ **Typy rekurze**

- ***Koncová rekurze*** - Rekurzivní volání je posledním příkazem v rekurzivní funkci - po něm již nejsou prováděny žádné další operace

  Příklad:
  V rekurzivním výpočtu největšího společného dělitele je jedna větev fce GCD (n, m) definována jako GCD(n,m) = GCD(m, Remainder(n, m))

- ***Vnořená*** - Rekurzivní funkce, jejíž argumenty jsou specifikovány rekurzivně

  Příklad:
  V Ackermannově funkci je jedna větev A(m − 1,A(m, n − 1))

- ***Kaskádní rekurze*** - V těle funkce jsou vedle sebe alespoň dve rekurzivní volání

  Příklad:
  V rekurzivním výpočtu Fibonacciho čísla je jedna větev F(n − 1) + F(n − 2)

- ***Lineární rekurze*** - V těle funkce je pouze jedno rekurzivní volání anebo jsou dvě, ale vyskytují se v disjunktních větvích podmíněných příkazu a nikdy se neprovedou současně

  Příklad:
  Při výpočtu faktoriálu je pro n != 1 výpočet n * Fac(n − 1)

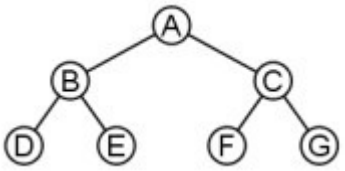
➢ **Rekurze vs. Iterace**

Rekurzivní programování má základní oporu v teoretické informatice, kde bylo dokázáno, že:
- Každou funkci, která může být implementovaná na vonNeumannovském počítači, lze vyjádřit rekurzivně bez použití iterace.
- Každá rekurzivní funkce se dá vyjádřit iterativně s použitím zásobníku (implicitní zásobník se stane viditelný uživateli).

Koncovou rekurzi lze vždy nahradit iterací bez nutnosti zásobníku.

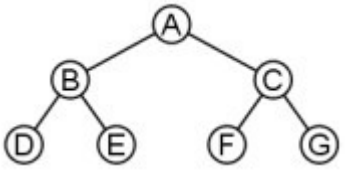
➢ **Rekurzivní průchody stromem**

- **Do hloubky** (DFS - Depth-First-Search) lze procházet třemi různými způsoby podle pořadí zpracování uzlu samotného a procházení jeho potomků
  - PreOrder - při vstupu do uzlu je nejdříve zpracován uzel samotný a následně jsou projiti jeho potomci



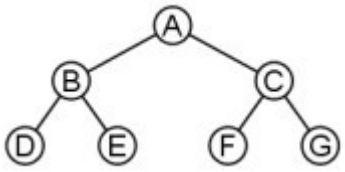
Výsledek: **ABDECFG**

  - InOrder - uzel je zpracován po zpracování levého podstromy a před zpracováním pravého podstromu (tedy jen pro binární stromy)



Výsledek: **DBEAFCG**

- PostOrder - uzel je zpracován až po projití levého i pravého podstromu



Výsledek: **DEBFGCA**

- **Do šířky** (BFS - Breadth-first-Search) procházíme nejblizší sousedy, poté sousedy sousedů, atd.

➢ **Příklad**

Základní ukázky rekurze:

- TreeSize(node) vrátí počet uzlů ve stromu určeného kořenem node

```
function TreeSize(node)
{
 if (node = NIL) then return(0);
 return(TreeSize(node.left) + TreeSize(node.right) + 1)
}
```

- TreeDepth(node) vrátí hloubku stromu určeného kořenem node

```
function TreeDepth(node)
{
 if (node = NIL) then return(0);
 return(max(TreeDepth(node.left),TreeDepth(node.right)) + 1)
}
```

2. Prohledávání s návratem (Backtracking)

Backtracking je postup, kdy se inkrementálně vytváří částeční kandidáti na řešení problému a jakmile se zjistí, že z určitého částečného kandidáta nemůže být vytvořeno řešení k problému, je tento kandidát zahozen (již se z něj nevytváří další kandidáti). Backtracking si lze představit jako strom částečných kandidátů , každý další uzel je doplněním předešlého o jeden rozšiřující krok blíže k celkovému řešení. (Jedná se o představu, v praxi se takový strom nevytváří a následně neprochází, ale je realizován například rekurzivním sestupem). Tento strom pak backtracking prochází **do hloubky** a v případě, že určitý uzel nemůže být řešením, nejsou procházeny větve z něj vedoucí.

Variací backtrackingu je **backjumping**, který se po vyloučení uzlu nevrací o jednu úroveň výš, ale na základě určité heuristiky o více úrovní výše.

➢ **Ukázka**

V ukázce jsou použity funkce:

- *root(P)* - kořenový (např. prázdné) částečný kandidát,
- *reject(P, c)* - vrátí true, pokud částečné řešení **c** nemůže být součástí správného řešení k **P**,
- *accept(P, c)* vrátí true, pokud **c** je úplné a správné řešení k **P**,
- *output(P, c)* výstup řešení z algoritmu,
- *first(P, c)* vrátí první řešení rozšířené o jeden rozšiřující krok od **c**,
- *next(P, s)* vrátí další rozšířené řešení od **c** po rozšířeném řešení **s**

```
procedure bt(c)
    if reject(P,c) then return
    if accept(P,c) then output(P,c)
    s ← first(P,c)
    while s ≠ Λ do
      bt(s)
      s ← next(P,s)
```

Realizace backtrackingu pro vstupní problém P bude spuštěna voláním *bt(root(P))*. Tato implementace najde všechna řešení. Pro terminaci při prvním nalezeném řešení stačí ukončit algoritmus po prvním zavolání funkce *output*

Dalším ukázkovým příkladem backtrackingu je řešení problému osmi dam (8 dam rozestavěných na šachovnici tak, aby se neohrožovaly). Zde se postupně konstruuje řešení pro prvních k řad šachovnice a k dam. Pokud nějaké řešení nevyhovuje, můžeme veškeré další podřešení využívající rozestavění tohoto neplatného řešení vyloučit.

3. Dynamické programování

Tento postup je založen na rozdělení problému na podproblémy, které se ovšem mohou překrývat (v tom smyslu, že při rozkladu (zpravidla rekurzivním) mohou vzniknout stejné podproblémy). Během řešení jednotlivých podproblémů se ukládají výstupy tak, aby bylo možné tyto výstupy použít, pokud se vyskytne stejný podproblém dále ve výpočtu.

Lze rozlišit dva přístupy:

- ***Shora-dolů*** - Problém se rozloží na podproblémy (rekurzivně) a řešení podproblémů jsou zapamatovávána, pokud se při výpočtu dalších podproblémů vyskytnou znovu
- ***Zdola-nahoru*** - Najdou se všechny možné podproblémy, ty se vyřeší, zapamatuje se jejich řešení, které bude následně případně využito při rešení větších podproblémů. Tento redukuje rekurzi, ale může být problematické určit, jaké jsou možné podproblémy

➢ **Příklad**

Při počítání Fibonacciho řady jednoduchou rekurzivní funkcí níže jsou podřady počítány pro každé volání celé znova. Náročnost tohoto postupu je exponenciální!

```
function fib(n)
     if n = 0 return 0
     if n = 1 return 1
     return fib(n − 1) + fib(n − 2)
```

Toto je rozpis rekurzivního volání funkce fib pro fib(5). Všimněte si, že hodnota fib(2) byla počítána třikrát!

```
 1. fib(5)
 2. fib(4) + fib(3)
 3. (fib(3) + fib(2)) + (fib(2) + fib(1))
 4. ((fib(2) + fib(1)) + (fib(1) + fib(0))) + ((fib(1) + fib(0)) + fib(1))
 5. (((fib(1) + fib(0)) + fib(1)) + (fib(1) + fib(0))) + ((fib(1) + fib(0)) +
fib(1))
```

Při použití dynamického programování uložíme výsledek podproblému, který řešíme a pokud bude při dalším volání potřeba, není nutné podproblém řešit znovu - zde si do datové struktury map ukládáme spočítanou hodnotu n-tého prvku Fibonacciho řady, a pokud je v budoucnu znovu potřeba, není nutné ji počítat.

```
var m := map(0 → 0, 1 → 1)
function fib(n)
    if map m does not contain key n
        m[n] := fib(n − 1) + fib(n − 2)
    return m[n]
```

➢ **Optimalizační úlohy**

Dynamické programování se často využívá v optimalizačních úlohách, kde hledáme **optimální řešení**. Postup při tvoření algoritmu založeném na dynamickém programování je následující:

- Charakterizace struktury optimálního řešení (jak je optimální řešení definováno - určení parametrů řešení atd.)
- Rekurzivně definujeme hodnotu optimálního řešení (jak se pozná optimální podřešení)
- Efektivní výpočet hodnoty optimálního řešení (shora-dolu, zdola-nahoru)
- Struktura optimálního řešení z podřešení - pouze pokud je struktura potřeba - někdy stačí jen výsledná hodnota)

➢ **Typické problémy**

- Floydův algoritmus hledání nejkratších cest v grafech.
- Hledání nejdelšího společného podřetězce dvou řetězců.
- Výpočet Levenshteinovy vzdálenosti mezi 2 textovými řetězci.
- aj.

4. Zametací technika

Je postup z výpočetní geometrie, kdy postupujeme mezi objekty zleva doprava (zpravidla x souřadnice) a postupně zpracováváme místa (vyjádřené souřadnicí po které postupujeme), která nás zajímají. Pro představu se používá idea svisle zametací přímky, která představuje onu hranici posouvající se zleva doprava.

Základní charakterizující body jsou:

- Suneme svislou přímku (scanline, SL, zametací přímku) zleva doprava nad množinou objektů
- Přímkou posouváme na souřadnice, které nás zajímají (např. mezi body na grafu - nikoliv po x++)
- Souřadnice, na které se chceme dostat, jsou v prioritní frontě (tu nazýváme postupový plán, x-struktura, B)
- O již projitých místech/souřadnicích si ukládáme informace (nazývá se y-struktura nebo T)

➢ **Příklad**

Hledám minimální body (takové, kde v levém dolním segmentu nejsou žádné body)

- 1. Seřadím body dle x a uložím do x-struktury
- 2. Pamatuji si v y-struktuře jednu hodnotu - minimální (nejspodnější) y souřadnici. (Prvotní hodnota je nekonečno)
- 3. Zametací přímku umisťují do body x-struktury (tedy procházím zleva doprava body a v každém provedu aktualizaci y-struktury)
- 4. Aktualizuji y-strukturu - pokud je aktuální souřadnice menší než hodnota v y-struktuře, nahradím.

5. Rozděl a panuj (Divide and Conquer)

Tento postup je založen na několikanásobném rekurzivním volání sebe sama na několik podmnožin vstupních dat, má tři části:

- Rozděl (Divide) - Rozděl řešení problému na několik podproblémů stejného typu na podmnožinách vstupních dat
- Panuj (Conquer) - Podproblémy jsou řešeny rekurzivně řešeny stejným algoritmem. Dostatečně malé podproblémy jsou vyřešeny přímou metodou (např. pole o jednom prvku je již seřazené).
- Spoj (Combine) - Výsledky podproblémů jsou při návratu z rekurze spojovány do větších podřešení a na konci je celkové řešení

Typickým příkladem je ***Merge Sort*** či ***Quick Sort***.

➢ Grafika

Přístup rozděl a panuj jsme také probírali v rámci výpočetní geometrie pro hledání konvexní obálky. Algoritmus vypadá takto:

1. Vybraným způsobem rozdělíme množinu bodů X na dvě - obsahující body nalevo a napravo od určité souřadnice x
2. Rekurzivně zpracujeme tyto dvě podmnožiny (pro dva body je obálka přímka mezi nimi..)
3. Spojíme obálky obou podmnožin - vytvoříme horní most a to tak, že
   - Procházíme body na horní straně obálek (nejlepé začneme od nejvyšších bodů na obou obálkach) - máme vždy vybrány dva body, mezi kterými je imaginární přímka
   - Pokud bod, který procházíme, leží **nad přímkou** spojující dva vybrané body, nahradíme vybraný bod v příslušné obálce aktuálně procházeným
   - Jakmile už nelze nalézt žádný bod, který toto splňuje, našli jsme horní most
4. Spojíme obálky obou podmnožin - vytvoříme dolní most - analogicky k hornímu. Máme konvexní obálku.

6. Prořezávej a Hledej

Tento postup optimalizace hledání je založen na postupném vyřazování části prohledávaných dat a tím redukováním složitosti hledání. Toto paradigma je velmi podobné algoritmům typu rozděl a panuj (divide and conquer), zásadní rozdíl je ovšem v tom, že při prořezávání neprocházíme všechny větve, ale pouze ty, které pro nás dávají smysl.

➢ Příklad

Hledáme i-té nejnižší číslo v neseřazeném seznamu čísel. Jednoduchý postup je seřadit seznam a vybrat i-tý prvek - tento postup však není nejefektivnější.

Využijeme dělící funkce použité z quicksortu, která na daném poli vybere pivot a prvky menší než pivot přehází do levé části pole a prvky větší do pravé části. Z pozice pivotu v tomto poli můžeme určit, zda hledat i-té nejnižší číslo v levé nebo pravé části - v té rekurzivně opakujeme postup, druhou části se pak dále nemusíme zabývat.

Příklad operuje nad globálním polem *array* a využívá následující funkci: *int RSPLIT(l,r)* - v podpoli určeným indexy *l* a *r* vybere pivot a prohází prvky pole tak, aby prvky menší než pivot byly nalevo a větší napravo. Návratová hodnota je index pivotu.

```
int[] array; //
int RSELECT(int l, r, i) // Vrátí index i-tého nejmenšího prvku
  q = RSPLIT(l, r); m = q - l + 1;
```

```
    if i < m then return RSELECT(l, q - 1, i) // Vlevo je více prvků než i -
tedy i-tý nejmenší musí ležet tam
    elseif i = m then return q // Vlevo je právě i prvků - pivot je i-tý
nejmenší
    else return RSELECT(q + 1, r, i - m) // Vlevo je méně prvků než i - i-tý
nejmenší musí ležet v pravé podčastí. i je zmenšeno o počet menších prvků, které
jsme již našli (nalevo od pivotu)
    endif .
```

Zdroje

- Přednášky Y36DSA https://service.felk.cvut.cz/courses/X36DSA/LectureNotes.html
- Wikipedia - http://en.wikipedia.org
- Prune and Search http://www.cs.duke.edu/courses/fall05/cps230/L-03.pdf
- http://www.nti.tul.cz/~vrany/ads/prednaska02_2009.pdf