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

Obsah

Techniky návrhu algoritmů **2**

Rekurze, algoritmy prohledávání s návratem, dynamické programování **5**

Rekurze **5**

Prohledávání s návratem (Backtracking) **7**

Dynamické programování **7**

Zametací technika, metoda „rozděl a panuj“ a „prořezávej a hledej“ **9**

Zametací technika **9**

Rozděl a panuj (Divide and Conquer) **9**

Prořezávej a Hledej **10**

Techniky návrhu algoritmů

Algoritmus bývá často definován jako: „přesný návod či postup, kterým lze vyřešit daný typ úlohy.“ Toto je sice na první pohled pravdivá, ale při bližším prozkoumání nepřesná definice. Například některé matematické postupy by této definici vyhovovaly, ale nejsou algoritmy. Přesné znění definice algoritmu zní: „Algoritmus je procedura proveditelná Turingovým strojem.“
Turingův stroj je teoretický model počítače popsaný matematikem Alanem Turingem. Skládá se z procesorové jednotky, tvořené konečným automatem a programu ve tvaru pravidel přechodové funkce a potenciálně nekonečné pásky pro zápis mezivýsledků a vstupů dat.
Tímto končím historický a etymologický úvod a začínám se věnovat vlastnostem algoritmů.

Vlastnosti algoritmů:
Algoritmus obvykle pracuje s nějakými vstupy, veličinami, které jsou mu předány před započetím jeho provádění, nebo v průběhu jeho činnosti. Vstupy mají definované množiny hodnot, jichž mohou nabývat. Algoritmus má alespoň jeden výstup, veličinu, která je v požadovaném vztahu k zadaným vstupům, a tím tvoří odpověď na problém, který algoritmus řeší.
Algoritmus musí být:

- **Konečný**
  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.
- **Deterministický**
  Každý krok algoritmu musí být jednoznačně a přesně definován; v každé situaci musí být naprosto zřejmé, co a jak se má provést, jak má provádění algoritmu pokračovat. Protože běžný jazyk obvykle neposkytuje naprostou přesnost a jednoznačnost vyjadřování, byly pro zápis algoritmů navrženy programovací jazyky, ve kterých má každý příkaz jasně definovaný význam. Vyjádření výpočetní metody v programovacím jazyce se nazývá program.
- **Efektivní**
  Obecně požadujeme, aby algoritmus byl efektivní, v tom smyslu, že požadujeme, aby každá operace požadovaná algoritmem, byla dostatečně jednoduchá na to, aby mohla být alespoň v principu provedena v konečném čase pouze s použitím tužky a papíru.
- **Obecný**
  Algoritmus neřeší jeden konkrétní problém (např. „jak spočítat 3×7“), ale obecnou třídu obdobných problémů (např. „jak spočítat součin dvou celých čísel“).
- **Rezultativní**
  Algoritmus při zadání vstupních dat vždy vrátí nějaký výsledek (může se jednat i jen o chybové hlášení).

Správnost algoritmu
Algoritmus je správný tehdy, když pro všechny údaje splňující vstupní podmínku se proces zastaví a výstupní údaje splňují výstupní podmínku. K ověření správnosti algoritmu nestačí vyzkoušet reakci algoritmu na konečný počet vstupních dat, i když se to tak v praxi často dělá a takové ověření o správnosti algoritmu leccos vypoví. Není ovšem dokázáno, že se algoritmus při neočekávané kombinaci vstupních dat nezhroutí. Pro ověřování správnosti algoritmu neexistuje univerzální metoda, algoritmus by měl být matematicky dokázán sledem předem známých kroků (operací), které nevyvratitelně vedou pro všechna přípustná data ke správnému výsledku úlohy. Je zřejmé, že takový algoritmus je pak korektním řešením úlohy. Algoritmus můžeme považovat za korektní, pokud není opomenuta žádná z možností zpracování dat při průchodu algoritmem. Algoritmus je částečně (parciálně) správný, právě když platí, že pokud skončí, vydá správný výsledek. K dokázání částečné správnosti výpočtu můžeme použít tzv. invariant – tvrzení, které platí po celou dobu výpočtu. Nutné je také ověřit konečnost algoritmu (pro všechna přípustná data algoritmus po konečném počtu kroků skončí).

Konečnost algoritmu
Konečnost algoritmu je často intuitivně zřejmá tím, že se pro vstupní data nemůže algoritmus zacyklit. Matematicky lze dokázat konečnost algoritmu takto: Pokud najdeme způsob, který každý stav výpočtu ohodnotí přirozeným číslem, a ukážeme, že provedením jednoho kroku algoritmu se tato hodnota zmenší, je jasné, že algoritmus po konečném počtu kroků skončí, neboť interval počtu průchodů algoritmem je konečný.

Druhy algoritmů

Algoritmy můžeme klasifikovat různými způsoby. Mezi důležité druhy algoritmů patří:

- **Rekurzivní algoritmy**, které využívají (volají) samy sebe.
- **Hladové algoritmy** se k řešení propracovávají po jednotlivých rozhodnutích, která, jakmile jsou jednou učiněna, už nejsou dále revidována.
- **Algoritmy typu rozděl a panuj,** dělí problém na menší podproblémy, na něž se rekurzivně aplikují (až po triviální podproblémy, které lze vyřešit přímo), po čemž se dílčí řešení vhodným způsobem sloučí.
- **Algoritmy dynamického programování** pracují tak, že postupně řeší části problému od nejjednodušších po složitější s tím, že využívají výsledky již vyřešených jednodušších podproblémů. Mnoho úloh se řeší převedením na grafovou úlohu a aplikací příslušného grafového algoritmu.
- **Pravděpodobnostní algoritmy** (někdy též probabilistické) provádějí některá rozhodnutí náhodně či pseudonáhodně.
- V případě, že máme k dispozici více počítačů (procesorů nebo jader), můžeme úlohu mezi ně rozdělit, což nám umožní ji vyřešit rychleji; tomuto cíli se věnují **paralelní algoritmy**.
- **Genetické algoritmy** pracují na základě napodobování biologických evolučních procesů, postupným „pěstováním“ nejlepších řešení pomocí mutací a křížení. V genetickém programování se tento postup aplikuje přímo na algoritmy (resp. programy), které jsou zde chápany jako možná řešení daného problému.
- **Heuristické algoritmy** si za cíl nekladou nalézt přesné řešení, ale pouze nějaké vhodné přiblížení; používají se v situacích, kdy dostupné zdroje (např. čas) nepostačují na využití exaktních algoritmů (nebo pokud nejsou žádné vhodné exaktní algoritmy vůbec známy).

Přitom jeden algoritmus může patřit zároveň do více skupin (například může být zároveň rekurzivní a typu rozděl a panuj)

Algoritmus se navrhuje několika možnými způsoby:

- **Shora dolů**
  postup řešení rozkládáme na jednodušší operace, až dospějeme k elementárním krokům.
- **Zdola nahoru**
  z elementárních kroků vytváříme prostředky, které nakonec umožní zvládnout požadovaný problém.
- **Kombinace obou**
  obvyklý postup shora dolů doplníme "částečným krokem" zdola nahoru tím, že se například použijí knihovny funkcí, vyšší programovací jazyk nebo systém pro vytváření programů (CASE).

Nejužívanější metody návrhů algoritmů jsou následující:

Rozděl a panuj
Klasický případ aplikace postupu odshora dolů. Algoritmy typu „rozděl a panuj“ dělí problém na menší podproblémy, na něž se rekurzivně aplikují (až po triviální podproblémy, které lze vyřešit přímo), po čemž se dílčí řešení vhodným způsobem sloučí.
Zpracovává se množina ***V*** složená z ***n*** údajů. Tato množina se rozdělí na ***k*** disjunktních podmnožin, které se zpracují každá zvlášť. Získané dílčí výsledky se pak spojí a odvodí se z nich řešení pro celou množinu ***V***.
Klasickým případem je binární vyhledávání, nebo quicksort.

Hladový algoritmus
Velice přímočarý přístup k řešení určité třídy optimalizačních úloh.
Zpracovává se množina ***V*** složená z ***n*** údajů. Úkolem je najít podmnožinu ***W*** množiny ***V***, která vyhovuje určitým podmínkám a přitom optimalizuje předepsanou účelovou funkci. Jakákoliv množina ***W***, vyhovující daným podmínkám, se nazývá přípustné řešení. Přípustné řešení, pro které nabývá účelová funkce optimální hodnoty, se nazývá optimální řešení.
Hladový algoritmus se skládá z kroků, které budou procházet jednotlivé prvky z ***V***, a v každém kroku rozhodne, zda se daný prvek hodí do optimálního řešení. Prvky ***V*** bude vybírat v pořadí určeném jistou výběrovou procedurou. Výběrová procedura bude založená na nějaké optimalizační míře – funkci, která může být odvozena od účelové funkce. V každém kroku ale musíme dostat přípustné řešení. Jakmile je učiněno takové rozhodnutí, už není dále revidováno. Příkladem je třeba hledání nejkratší cesty grafu.

Dynamické programování

Dynamické programování funguje obdobně jako hladový algoritmus, ale negeneruje se pouze jedna posloupnost. Zkoumají se všechny posloupnosti, které by mohly být optimální a vylučují se ty, které optimální nebudou. Používají se dílčí výsledky, které byly získané již dříve během výpočtu.
Nejjednodušší a nejméně účinnou je metoda hrubé síly – vygenerují se všechny možné posloupnosti a pak se vybere ta nejlepší. Za pomoci dynamického programování se automaticky negenerují ty možnosti, které určitě nejsou optimální.
Opírá se o princip optimality: Optimální posloupnost rozhodnutí má tu vlastnost, že ať je počáteční stav a rozhodnutí jakékoliv, musí být všechna následující rozhodnutí optimální vzhledem k výsledkům rozhodnutí prvního.
Typickým příkladem využití dynamického programování jsou grafové úlohy a jejich příslušné grafové algoritmy.

Hledání s návratem (backtracking)

Hledání s návratem založené na prohledávání stavového stromu problému. Též se nazývá metoda pokusů a oprav, metoda zpětného sledování, metoda prohledávání do hloubky.
Metodu je možné použít v případě, že řešením je vektor (x1,...,xn) jehož jednotlivé složky vybíráme z množiny Si, $x_i \in S_i$. Zpravidla je třeba najít n-tici, která optimalizuje nějakou účelovou funkci P(x1,...,xn). Můžou se ale také hledat všechny n-tice, které tuto podmínku splňují. Metoda vytváří n-tice jednu složku po druhé. Přitom používá účelovou funkci (nebo nějakou vhodnou pomocnou funkci) a pro každou nově vytvořenou složku testuje, zda by taková n-tice vůbec mohla být optimální nebo splňovat dané podmínky. Jestliže pro nějaké xi žádaný vektor (x1,...,xi) nemůže být optimální, není třeba už žádný takový vektor testovat a vezmeme další možnou hodnotu i-té složky. Pokud jsou vyčerpány všechny hodnoty i-té složky, vrátí se metoda zpět o jeden krok a zkouší další možnou hodnotu xi-1.
Příkladem je třeba problém osmi dam, nebo chůze koně celou šachovnicí.

Rekurze, algoritmy prohledávání s návratem, dynamické programování

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)

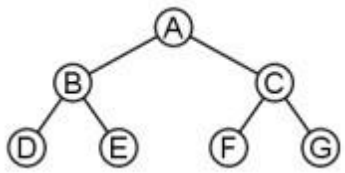
Rekurze vs. Iterace

- 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

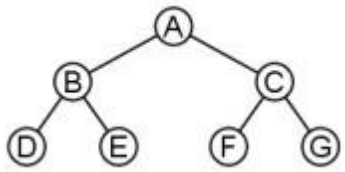
Do hloubky (DFS -Depth-First-Search)
lze procházet třemi různými způsoby podle pořadí zpracování uzlu samotného a procházení jeho potomků

PreOrder -při vstupu do uzlu je nejdříve zpracován uzel samotný a následně jsou projiti jeho potomci



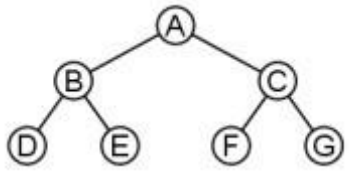
Výsledek: **ABDECFG**

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 nejbližší sousedy, poté sousedy sousedů, atd.

Příklad

Základní ukázky rekurze:
Funkce **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)
}
```

Funkce **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)
 }
```

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
Program pro kontrolu řešitelnosti Sudoku
http://www.algoritmy.net/article/1351/Sudoku

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.

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.

Používají se následující 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 řešení 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!
if n = 1 return 1
Toto je rozpis rekurzivního volání funkce fib pro fib(5). Všimněte si, že hodnota fib(2) byla počítána třikrát!

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

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.

Zametací technika, metoda „rozděl a panuj“ a „prořezávej a hledej“

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)
- 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.

Rozděl a panuj (Divide and Conquer)

Založena na opakovaném rozdělování problému na menší a jednodušší podproblémy, které jsou svým řešením podobné původnímu problému.
Důsledek návrhu algoritmu technikou ***Shora dolů***.
Dělení prováděno dokud není řešení podproblému triviální (tj. většinou přímé).
Takové řešení umíme zpravidla nalézt bez složitých výpočtů.
Podproblémy řešeny nezávisle na sobě a poté jejich řešení spojena v celek, čímž získáme řešení původního problému.
Techniky řešení:

- Rekurze
- Iterace

Hodí se pro dekomponovatelné úlohy, tj. úlohy rozdělitelné na podúlohy stejného nebo podobného typu.
Postup řešení 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

Výhody

Algoritmus je poměrně krátký a rychlý.
Efektivní řešení řady problémů.

Nevýhody

Značná paměťová složitost, pro rozsáhlá vstupní data mají algoritmy D&C vysoké nároky na paměť.
časová složitost $\Theta=(n \log n)$.
Možnost vzniku nekonečné rekurze.

Použití

- Třídící algoritmy (QuickSort)
- Vyhledávací algoritmy (Půlení intervalu).
- Datové struktury (Binární strom).
- atd

Prořezávej a Hledej

Prořezávej a hledej (Prune and search) je typ algoritmu založený na vyřazování neperspektivních dat – redukci velikosti problému. 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

Pokud hledáme n-té nejvyšší číslo v neseřazeném poli, tak by řešením zajisté bylo pole seřadit a podívat se na zadaný index. Toto řešení ovšem není příliš efektivní. Lepším řešením je upravit například Quicksort tak, aby se po rozdělení pole dle pivota prohledávala pouze ta část, která obsahuje řešení - Quicksort v každém svém kroku umístí pivota na korektní místo v seřazeném poli, není proto problém rozhodnout, ve které části se nachází ono hledané číslo.

Příklad implementace: http://www.algoritmy.net/article/158/Prorezavej-a-hledej