

## Prohledávání do šířky = algoritmus „vlny“

- souběžně zkoušet všechny možné varianty pokračování výpočtu, dokud nenajdeme řešení úlohy  
→ průchod stromem všech možných cest výpočtu do šířky, po vrstvách (v každé vrstvě zkouší všechny možnosti zleva doprava)
- je nutné pamatovat si všechny situace v jedné vrstvě stromu  
→ při větší hloubce je to paměťově velmi náročné (exponenciálně), až nepoužitelné
- použijeme, máme-li nalézt jedno (nejkratší) řešení  
→ vždy najde nejkratší možné řešení co do počtu kroků

- výhodné, existují-li v rozsáhlém prohledávacím stromě nějaká řešení vzdálená od startu jen na málo kroků – prohledá se jen několik horních vrstev stromu (významná časová úspora oproti prohledávání do hloubky)
- nevhodné při úloze nalézt všechna řešení – musí se projít celý strom, a to je při stejné časové složitosti výhodnější provádět do hloubky (paměťově mnohem méně náročné)
- v některých úlohách je nutné prohledávat do hloubky (prohledávání do šířky by se paměťově nezvládlo), v některých je nutné prohledávat do šířky (prohledávání do hloubky by se časově nezvládlo), v některých je to jedno (obě varianty prohledávání jsou paměťově i časově stejně náročné)

## Realizace prohledávání do hloubky a do šířky

Prohledávání do hloubky se zpravidla realizuje rekurzivní procedurou, totéž lze naprogramovat pomocí **zásobníku**:

vlož do zásobníku výchozí stav  
dokud (není zásobník prázdný) a (není nalezeno řešení) opakuj  
    vyjmi ze zásobníku vrchní stav a zpracuj ho  
    vlož do zásobníku jeho všechna možná pokračování

Prohledávání do šířky se realizuje velmi podobně pomocí **fronty**:

vlož do fronty výchozí stav  
dokud (není fronta prázdná) a (není nalezeno řešení) opakuj  
    vyjmi z fronty první stav a zpracuj ho  
    vlož do fronty jeho všechna možná pokračování

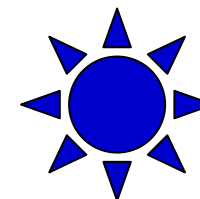
*Příklady:*

### **Proskákání celé šachovnice koněm (bylo)**

- každé možné řešení úlohy má délku 63 kroků,  
použití prohledávání do šířky je paměťově nereálné

### **Nejkratší cesta koněm na šachovnici**

- dána výchozí a cílová pozice, dojít šachovým koněm co nejmenším počtem tahů z výchozího na cílové pole
- použití prohledávání do hloubky je velmi nevhodné – backtracking do hloubky až 63, zatímco vždy existuje řešení délky maximálně 6
- postupujeme do šířky: sousední pole k výchozímu označíme 1, jejich sousedy 2, jejich sousedy 3, atd., dokud nedojdeme na cílové pole (číslo = délka nejkratší cesty ze startu)
- k nalezení vlastní cesty je nutný ještě „**zpětný chod**“  
(cíl → sousední pole s hodnotou o 1 menší → ... → start)



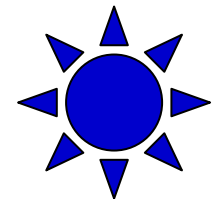
Možnosti realizace:

1. Pole z předchozího kroku vlny (tzn. pole s číslem o 1 menším) nevyhledávat vždy procházením celé šachovnice, ale ukládat si jejich souřadnice navíc do fronty

→ je třeba paměť navíc, ale urychlí to výpočet při „vlně“

2. U každého pole na šachovnici si pamatovat nejen minimální počet kroků, které na něj vedou, ale i souřadnice předchozího pole

→ snadné nalezení předchůdce při zpětném chodu (ale zase je to paměťově i časově náročnější při provádění „vlny“)



## **Procházení grafu (např. pro určení souvislosti)**

- v obyčejném neorientovaném grafu projít ze zvoleného výchozího vrcholu všechny dostupné vrcholy
- označujeme všechny již navštívené vrcholy, abychom někam nešli dvakrát (abychom nezačali v grafu chodit v cyklu)
- můžeme rovnocenně použít průchod do hloubky nebo do šířky, liší se pořadím návštěvy jednotlivých vrcholů, ale oba najdou stejné řešení se stejnou (kvadratickou) časovou i paměťovou složitostí

Možnosti realizace:

1. rekurzivní procedura realizující průchod do hloubky
2. zásobník nebo fronta pro řízení průchodu do hloubky resp. do šířky – představují seznam „dluhů“ = čísla těch vrcholů, do nichž jsme při prohledávání už přišli, ale z nichž jsme nezkoušeli jít dál
3. seznam „dluhů“ dokonce nemusí být realizován ani jako zásobník nebo fronta, můžeme z něj vybírat vrchol ke zpracování libovolně (pak průchod grafem neprobíhá do hloubky ani do šířky)

# „Rozděl a panuj“

- metoda rekurzivního návrhu algoritmu (programu)

Problém se rozdělí na dva (příp. více) podproblémy stejného typu, ale menší velikosti, z jejich řešení se pak snadno sestaví řešení původního problému. Každý podproblém je buď už triviální a vyřeší se přímo, nebo se k jeho řešení použije stejný rekurzivní postup.

**Realizace:** rekurzivní procedurou

**Podmínka rozumné (tj. efektivní) použitelnosti metody:**

podproblémy vznikající rozkladem jsou na sobě nezávislé (nevyužívají stejné dílčí podúlohy a jejich řešení)

**Protipříklad:** Fibonacciho čísla rekurzivně

- řešení úlohy se sice skládalo z řešení podobných menších podúloh, ale ty nebyly na sobě nezávislé → opakované výpočty, velmi neefektivní řešení (exponenciální časová složitost)

# Hanojské věže

- 3 kolíky A, B, C
- na A je N disků různé velikosti, seřazené od největšího (dole) k nejmenšímu (nahore)
- kolíky B a C jsou prázdné
- úkol: přenést všechny disky z A na B, mohou se odkládat na C
- podmínka: nikdy nesmí ležet větší disk na menším

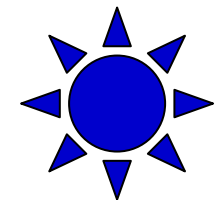
## Řešení:

**PŘENESVĚŽ** (N, A, B, C) = přenes N disků z A na B pomocí C

→ provedeme jediňě takto:

1. **PŘENESVĚŽ** (N-1, A, C, B) - rekurze
2. přenes disk z A na B - triviální akce
3. **PŘENESVĚŽ** (N-1, C, B, A) - rekurze

**Časová složitost:**  $O(2^N)$  – dána charakterem problému, k vyřešení úkolu je nutné provést tolik přesunů.





# Vyhodnocení aritmetického výrazu

metodou Rozděl a panuj

## Postup:

1. ve výrazu nalézt znaménko, které se vyhodnotí až jako poslední (zcela mimo závorky, nejnižší priority, z nich co nejvíce vpravo)
2. výraz rozdělit na dva podvýrazy vlevo a vpravo od znaménka
3. oba podvýrazy vyhodnotit
4. s výsledky obou podvýrazů vykonat poslední operaci určenou znaménkem

## Realizace:

1. lineární průchod výrazem zleva doprava (pokud znaménko mimo závorky neexistuje, odstranit z výrazu vnější závorky a zopakovat)
3. buď je podvýraz triviální (konstanta), nebo se vyhodnotí rekurzivním voláním téhož algoritmu
2. a 4. triviální akce (konstantní složitosti)

### Příklad:

$$\begin{array}{rcl} 5 * ( 3 + 4 * 6 - 8 ) & & 95 \\ 5 \quad ( 3 + 4 * 6 - 8 ) & & \\ \quad 3 + 4 * 6 - 8 & & 19 \\ \quad 3 + 4 * 6 \quad 8 & & 27 \\ \quad 3 \quad 4 * 6 & & 24 \\ \quad \quad 4 \quad 6 & & \end{array}$$

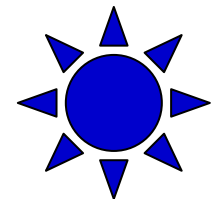
### Časová složitost:

$N$  – délka výrazu (počet znamének)

složitost jako u quicksortu – záleží na struktuře výrazu

- v nejhorším případě  $O(N^2)$

- v nejlepším a průměrném případě  $O(N \cdot \log N)$



# Vnitřní třídění (pokračování)

## ***Rekurzivní metody***

- složitější zápis programu, technika „Rozděl a panuj“
- v průměrném případě optimální časová složitost  $O(N \log N)$ , Quicksort může mít v nejhorším případě složitost  $O(N^2)$
- Quicksort třídí „na místě“, Mergesort potřebuje další pole velikosti  $N$ , oba algoritmy navíc potřebují paměť na realizaci rekurze (tzn. systémový zásobník nebo vlastní zásobník v případě odstranění rekurzivních volání)

# Quicksort

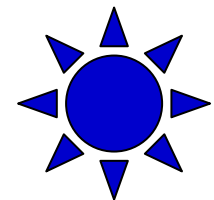
- v průměru nejrychlejší známý třídící algoritmus
- inicializace: tříděným úsekem je celé pole čísel
- v tříděném úseku vybrat jednu hodnotu (označme  $X$ )
- prvky přerovnat tak, aby vlevo byly prvky  $\leq X$  a vpravo prvky  $\geq X$  (lineární časová složitost vzhledem k délce tříděného úseku)
- tím se pole rozdělí na dvě části
- oba vzniklé úseky setřídit rekurzivním voláním téhož algoritmu (pokud nejsou triviální, tzn. délky 1, příp. 2)
- po skončení všech rekurzivních volání je celé pole utříděno

## Realizace:

rekurzivní procedura

parametry = indexy v poli vymezující aktuální tříděný úsek

volána s parametry  $(1, N)$



```

const MaxN = 100; {maximální počet tříděných čísel}

type Pole = array[1..MaxN] of integer;
                                     {tříděná čísla}

var P: Pole;      {uložení tříděných údajů}
    N: 1..MaxN;    {počet prvků v poli P}
    I: integer;

procedure Quicksort1(var P:Pole; Zac,Kon:integer);
{setřídí v poli P úsek od indexu Zac do indexu Kon}
...

begin  {hlavní program}
...
Quicksort1(P,1,N);
...
end.

```

```

procedure Quicksort1(var P:Pole; Zac,Kon:integer);
{setřídí v poli P úsek od indexu Zac do indexu Kon}

var X: integer;      {hodnota pro rozdělení na úseky}
    Q: integer;      {pomocné pro výměnu prvků v poli}
    I,J: integer;    {posouvané pracovní indexy v poli}
begin
    I:=Zac;
    J:=Kon;
    X:=P[(Zac+Kon) div 2];
    {za hodnotu X vezmeme pro jednoduchost
     prostřední prvek ve zkoumaném úseku}
    repeat
        while P[I] < X do I:=I+1;
        while P[J] > X do J:=J-1;

```

```

if I < J then           {vyměnit prvky s indexy I a J}
  begin
    Q:=P[I]; P[I]:=P[J]; P[J]:=Q;
    I:=I+1; J:=J-1;      {posun indexů na další prvky}
  end
else if I = J then
  {indexy I a J se sešly, oba dva ukazují
   na hodnotu X}
  begin
    I:=I+1; J:=J-1      {posun indexů na další prvky
                        - nutné kvůli ukončení cyklu}
  end
until I > J;

{úsek <Zac,Kon> je rozdělen na úseky <Zac,J> a <I,Kon>,
které zpracujeme rekurzivním voláním procedury:}
if Zac < J then Quicksort1(P,Zac,J);
if I < Kon then Quicksort1(P,I,Kon);
end; {procedure Quicksort1}

```

## **Paměťová složitost: $O(N)$**

třídění probíhá na místě v poli

navíc je ale třeba paměť na realizaci rekurzivních volání (zásobník)

## **Časová složitost:**

*nejhorší případ*

- za  $X$  vždy vybereme nejmenší nebo největší prvek v úseku
- postupně procházíme úseky délky  $N, N-1, N-2, \dots$ ,  
celkem tedy práce  $N + (N-1) + (N-2) + \dots + 1 = N.(N+1)/2 \dots O(N^2)$

*nejlepší případ*

- za  $X$  vždy vybereme prostřední hodnotu (medián) v úseku
- procházíme 1 úsek délky  $N$ , 2 úseky délky  $N/2$ , 4 úseky délky  $N/4, \dots$ ,  
celkem hloubka rekurze  $\log N$  a na každé hladině rekurze práce  $N$   
(součet délek  $K$  úseků, každý dlouhý  $N/K$  prvků)  $\dots O(N \log N)$

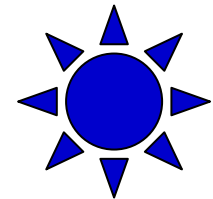
*průměrný případ*

- lze dokázat, že  $O(N \log N)$  jako v nejlepším případě



## Quicksort - implementace bez rekurzivní procedury

- použití rekurzivní procedury lze nahradit vlastním zásobníkem
- zásobník „dluhů“ = seznam úseků, které je ještě třeba dotřídit
- místo rekurzivního volání → vložení úseku do zásobníku
- v cyklu se postupně vybírají ze zásobníku jednotlivé dluhy a řeší se (čímž zase vznikají nové, menší dluhy)
- pro programátora více práce při psaní programu
- výsledný kód může být úspornější
  - \* na čas (úspora za režii rekurzivních volání)
  - \* na paměť (při dobré organizaci práce stačí zásobník logaritmické výšky)



```

const MaxN = 100;           {maximální počet tříděných čísel}
      MaxNdiv2 = 50;        {= MaxN div 2 (velikost zásobníku)}

type  Pole = array[1..MaxN] of integer;    {tříděná čísla}

var P: Pole;                {uložení tříděných údajů}
    N: 1..MaxN;              {počet prvků v poli P}
    Zasob: array[1..MaxNdiv2] of
        record Zac,Kon: integer end;
        {zásobník úseků čekajících na zpracování}
    Vrchol: 0..MaxN;          {vrchol zásobníku}
    I: integer;

procedure Quicksort2(var P:Pole; Zac,Kon:integer);
{setřídí v poli P úsek od indexu Zac do indexu Kon}
...

```

```

procedure Quicksort2(var P:Pole; Zac,Kon:integer);
{setřídí v poli P úsek od indexu Zac do indexu Kon}
var X: integer;      {hodnota pro rozdělení na úseky}
    Q: integer;      {pomocné pro výměnu prvků v poli}
    Z,K: integer;    {začátek a konec zkoumaného úseku}
    I,J: integer;    {posouvané pracovní indexy v poli}
begin
    Zasob[1].Zac:=Zac;
    Zasob[1].Kon:=Kon;
    Vrchol:=1;      {celý tříděný úsek vložen do zásobníku}
    while Vrchol > 0 do {zásobník je neprázdný}
        begin
            Z:=Zasob[Vrchol].Zac;
            K:=Zasob[Vrchol].Kon;
            Vrchol:=Vrchol-1; {odebrán jeden úsek ze zásobníku}
            I:=Z; J:=K;
            X:=P[(I+J) div 2];
            {za hodnotu X vezmeme pro jednoduchost
             prostřední prvek ve zkoumaném úseku}

```

```

repeat
  while P[I] < X do I:=I+1;
  while P[J] > X do J:=J-1;
  if I < J then          {vyměnit prvky s indexy I a J}
    begin
      Q:=P[I]; P[I]:=P[J]; P[J]:=Q;
      I:=I+1; J:=J-1;    {posun indexů na další prvky}
    end
  else if I = J then
    {indexy I a J se sešly, oba dva ukazují
     na hodnotu X}
    begin
      I:=I+1; J:=J-1     {posun indexů na další prvky
                        - nutné kvůli ukončení cyklu}
    end
until I > J;

```

```

{úsek <Z,K> je rozdělen na úseky <Z,J> a <I,K>,
které vložíme do zásobníku k dalšímu zpracování:}
if Z < J then
    begin
        Vrchol:=Vrchol+1;
        Zasob[Vrchol].Zac:=Z;
        Zasob[Vrchol].Kon:=J
    end;
if I < K then
    begin
        Vrchol:=Vrchol+1;
        Zasob[Vrchol].Zac:=I;
        Zasob[Vrchol].Kon:=K
    end
end
end; {procedure Quicksort2}

```

# Mergesort (třídění sléváním)

- rozdělení pole na dvě stejně velké části
- každou z nich setřídíme  
(buď je triviální, nebo rekurzivním voláním téhož algoritmu)
- obě setříděné části pole slít dohromady = merge  
(lineární časová složitost vzhledem k délce tříděného úseku)

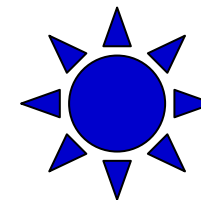
**Paměťová složitost:**  $O(N)$

algoritmus potřebuje druhé pomocné pole

navíc je ale třeba paměť na realizaci rekurzivních volání (zásobník)

**Časová složitost:**

procházíme 1 úsek délky  $N$ , 2 úseky délky  $N/2$ , 4 úseky délky  $N/4$ , ..., celkem hloubka rekurze  $\log N$  a na každé hladině rekurze práce  $N$  (součet délek  $K$  úseků, každý dlouhý  $N/K$  prvků) ...  $O(N \cdot \log N)$



```

procedure Mergesort(var P,Q:Pole; Zac,Kon:integer);
{setřídí v poli P úsek od indexu Zac do indexu Kon}
{Q je pomocné odkládací pole pro realizaci slévání}

var Stred:integer; {index prostředku tříděného úseku}
    i,j,k: integer;      {pomocné indexy pro slévání}

begin
    Stred:=(Zac+Kon) div 2;
    {konec levého úseku při rozdělení úseku <Zac,Kon>}
    if Zac < Stred then Mergesort(P,Q,Zac,Stred);
                                {utřídí levý úsek}
    if Stred+1 < Kon then Mergesort(P,Q,Stred+1,Kon);
                                {utřídí pravý úsek}

```

```

{slévání obou úseků:}
i:=Zac;           {levý úsek}
j:=Stred+1;       {pravý úsek}
k:=Zac;           {výsledek}
while (i<=Stred) and (j<=Kon) do
    {slévání setříděných úseků do pomocného pole Q}
    begin
        if P[i]<=P[j] then begin Q[k]:=P[i]; i:=i+1 end
            else begin Q[k]:=P[j]; j:=j+1 end;

        k:=k+1
    end;
    while i<=Stred do    {kopírování zbytku levého úseku}
        begin Q[k]:=P[i]; i:=i+1; k:=k+1 end;
    while j<=Kon do      {kopírování zbytku pravého úseku}
        begin Q[k]:=P[j]; j:=j+1; k:=k+1 end;

    {zbývá přenést setříděný úsek <Zac,Kon> zpět z Q do P:}
    for k:=Zac to Kon do
        P[k]:=Q[k]
end;    {procedure Mergesort}

```