Algoritmy vyhledávání a řazení

Zatím nad lineární datovou strukturou (polem) …

Vyhledávání

Vyhledávací problém

- je dáno Universum (množina prvků) U
- je dána konečná množina prvků $X \subset U$ (vyhledávací prostor)
- mějme prvek $x \in U$
- vyhledávací problém (Najdi(x,X)) je definován jako rozhodovací problém:
  - Najdi(x,X): jestliže $x \in X$, pak najdi=1, jinak najdi=0

- algoritmy vyhledávání závisejí na vyhledávacím prostoru a jeho reprezentaci
- druhy vyhledávacích problémů:
  - dynamický lexikon
    - vyhledávací prostor se v průběhu zpracování mění (vložení, rušení, náhrada prvku)
    - příklad: autorský katalog knihovny
  - statický lexikon
    - vyhledávací prostor se v průběhu zpracování nemění
    - příklad: telefonní seznam

- pro jednoduchost se omezíme na čísla typu int (univerzum U = int)
- množina X, ve které provádíme hledání, bude reprezentována polem čísel (zatím, poznáme i jiné výhodnější reprezentace, např. strom)
- jako výsledek hledání při programování zpravidla potřebujeme znát index nalezeného prvku (resp. ukazatel)

Sekvenční vyhledávání

- používá se při vyhledávání v poli (seznamu), které je neseřazené
- princip: sekvenčně (v cyklu) procházím prvky pole, dokud prvek nenaleznu nebo neprojdu celé pole

- budeme předpokládat následující pole

```
#define MAX 100
int pole[MAX];
```

- implementujeme vyhledávací funkci, která zjistí, zda je v poli, obsahující `n` prvků, hledaný prvek `x;` v kladném případě vrátí funkce index prvku, v případě neúspěchu hodnotu -1

```
int hledej(int x, int *pole, int n)
/* x je hledaný prvek, n je počet prvků
  pole */
{
  int i=0;
  while(i<n && pole[i]!=x) i++;
  return i!=n ? i : -1;
}
```

- někdy se implementovalo sekvenční vyhledávání se zarážkou (tzv. sentinel)
  - pole má o jeden prvek navíc (poslední), do kterého vložím hledaný prvek
  - došlo ke zjednodušení podmínky v cyklu

```c
int hledej2(int x, int *pole, int n)
{
  int i=0;
  pole[n]=x;
  while(pole[i]!=x) i++;
  return i!=n ? i: -1;
}
```

Vyhledávání binárním půlením

- používá se při vyhledávání v poli, kde jsou prvky seřazeny
- princip:
  - porovnám hledaný prvek `x` s prvkem uprostřed pole `pole[i]`
  - dojde-li ke shodě, prvek je nalezen; je-li x<pole[i], pokračuji v hledání v levé polovině pole bin. půlením, je-li x>pole[i], pokračuji v hledání v pravé polovině pole bin. půlením
  - vyhledávání končí neúspěchem, je-li prohledávaná část pole prázdná

```
int bin_pul(int x, int *pole, int n)
{
  int i,l,p;
  l=0; p=n-1;
  do
  {
    i = (l+p)/2;
    if (x<pole[i]) p=i-1; else l=i+1;
  }
  while(pole[i]!=x && l<=p);
  return pole[i]==x ? i: -1;
}
```

- trochu jinak:

```
int bin_pul(int x, int *pole, int n)
{
  int i,l,p;
  l=0; p=n-1;
  do
  {
    i = (l+p)/2;
    if (x==pole[i]) return i;
    if (x<pole[i]) p=i-1; else l=i+1;
  }
  while(l<=p);
  return -1;
}
```

- binární půlení se někdy vylepšuje jiným výpočtem indexu i (lineární interpolace podle hodnoty hledaného prvku vzhledem k nejmenšímu a největšímu prvku v poli)

```
i = (p-l)*(x-pole[l])/(pole[p]-pole[l])
```

Hodnocení algoritmů

- rychlost (kvalitu) algoritmů měříme tzv. *složitostí (complexity)* :
  - operační složitostí O(n) – doba trvání (počet kroků) algoritmu v závislosti na rozměru problému,
    - např. na počtu řazených čísel, velikost prohledávacího prostoru
  - paměťovou složitostí M(n) – velikost požadované paměti v závislosti na rozměru problému

Hodnocení algoritmů

- rychlost běhu programu závisí na mnoha faktorech
  - rychlost procesoru, velikost cache, progr. jazyku, překladači, stylu programování, …
- snažíme se určit rychlost algoritmu bez ohledu na tyto faktory; zajímá nás většinou chování algoritmu pro velké množství vstupních dat

Hodnocení algoritmů

- rychlost (kvalitu) algoritmů měříme tzv. asymptotickou *složitostí (complexity)*
  - jak závisí počet kroků algoritmu na počtu vstupních dat n limitně pro $n \to \infty$
  - snažíme se eliminovat konstanty
  - složitosti vyjadřujeme tzv. *řádem růstu funkce*

Hodnocení algoritmů

Asymptotická horní mez: O-notace

- jsou-li dány funkce $f(n)$ a $g(n)$, pak řekneme, že $f(n)$ je nejvýše řádu $g(n)$, $f(n) = O(g(n))$, jestliže platí

$$\exists c \in R^{+}\ \exists n_0 \in N^{+} :\ \forall n \geq n_0 : f(n) \leq c.g(n)$$

  - říkáme také, že $f(n)$ roste maximálně tak rychle jako $g(n)$
  - používáme ji pro vyjádření horní meze růstu až na multiplikativní konstantu

Hodnocení algoritmů

Asymptotická dolní mez: Ω-notace

- jsou-li dány funkce $f(n)$ a $g(n)$, pak řekneme, že $f(n)$ je nejméně řádu $g(n)$, $f(n) = \Omega(g(n))$, jestliže platí

$$\exists c \in R^{+}\ \exists n_0 \in N^{+} :\ \forall n \geq n_0 : f(n) \geq c.g(n)$$

  - říkáme také, že $f(n)$ roste minimálně tak rychle jako $g(n)$
  - používáme ji pro vyjádření dolní meze růstu až na multiplikativní konstantu

Hodnocení algoritmů

Asymptotická těsná mez: Θ-notace

- jsou-li dány funkce $f(n)$ a $g(n)$, pak řekneme, že $f(n)$ je téhož řádu jako $g(n)$, $f(n) = \Theta(g(n))$, jestliže platí

$$\exists c_1, c_2 \in R^+ \ \exists n_0 \in N^+ : \ \forall n \geq n_0 : c_1 g(n) \leq f(n) \leq c_2 g(n)$$

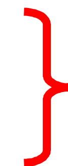
  - říkáme také, že $f(n)$ roste stejně tak rychle jako $g(n)$ až na multiplikativní konstantu

Hodnocení algoritmů

- platí

$$f(n) = \Theta(g(n)) \Leftrightarrow f(n) = O(g(n)) \wedge f(n) = \Omega(g(n))$$

- např.
  - $3n^2 + 2n + 5$ je řádu $O(n^2)$
  - $5n^4 + 3n^2 - 3$ je řádu $O(n^4)$



polynomiální

  - $2^n + 3$ je řádu $O(2^n)$ **exponenciální**
  - $n! + 3n + 4$ je řádu $O(n!)$ **faktoriální**

Odhadněte operační složitost:

- sekvenčního vyhledávání

$O(n)$

- násobení čtvercových matic o rozměru *n*

$O(n^3)$

- hledání binárním půlením

$O(\log_2 n)$

- někdy se stanovuje složitost pro různé případy uspořádání vstupních dat:
  - v nejlepším případě
  - v průměrném případě
    - často je složité statisticky určit tento případ
  - v nejhorším případě

- příklad: lineární hledání:
  - hledaný prvek je na prvním místě v poli
  - hledaný prvek je uprostřed
  - hledaný prvek je na konci pole nebo není přítomen

Řazení

Řazení:

- vytvoření posloupnosti prvků x_i, takové, že

$$x_{j1} \leq x_{j2} \leq, \ldots, \leq x_{jn}$$

$$\text{resp. } x_{j1} \geq x_{j2} \geq, \ldots, \geq x_{jn}$$

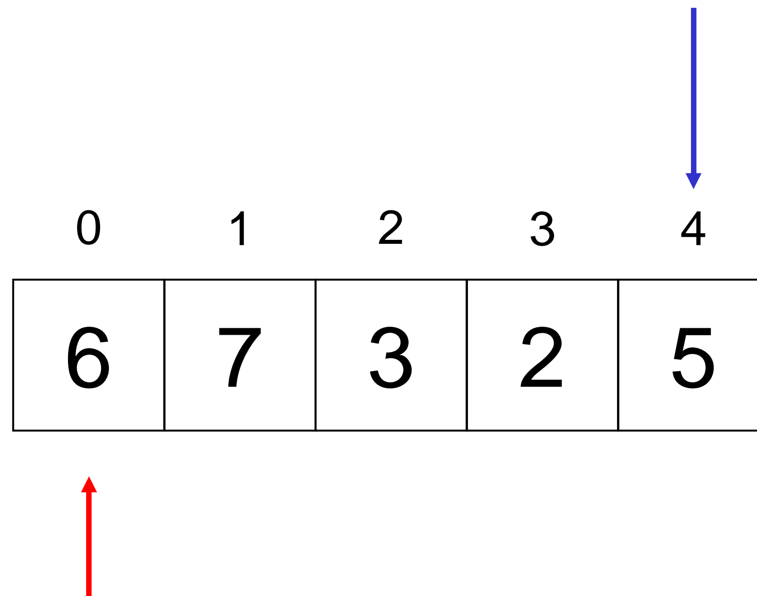
- univerzální algoritmy řazení:
  - zatřiďováním
  - výběrem maximálního (minimálního) prvku
  - záměnou (Bubble Sort)
  - Quick Sort

Řazení zatřiďováním

- do již seřazené posloupnosti vkládáme každý nový prvek rovnou na správné místo
- vhodné např. pro spojový seznam, nikoliv pro pole
- nutné spojit s operací hledání

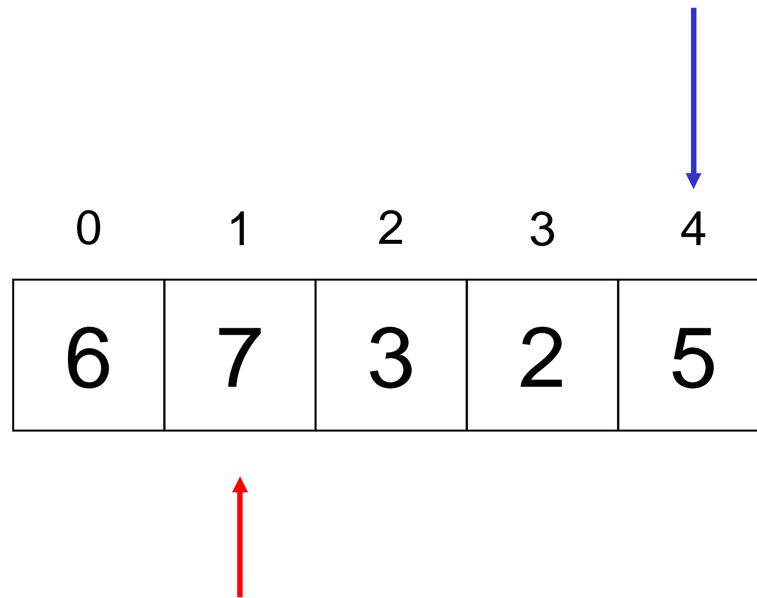
Řazení výběrem maximálního (minimálního) prvku

- v poli o `n` prvcích nalezneme maximální (minimální) prvek a vyměníme jej s posledním (prvním) prvkem
- krok hledání maximálního (minimálního) prvku opakujeme na poli o délce n-1, n-2,…,1



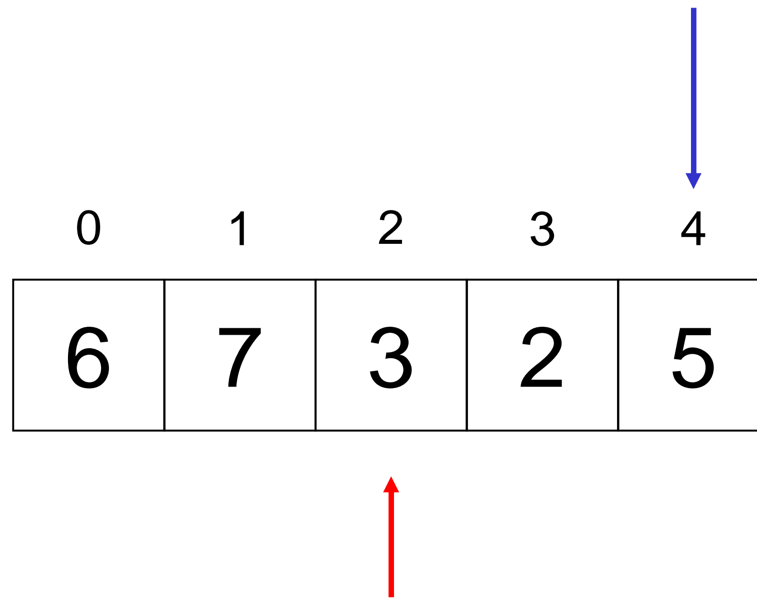
imax: 0

max: 6



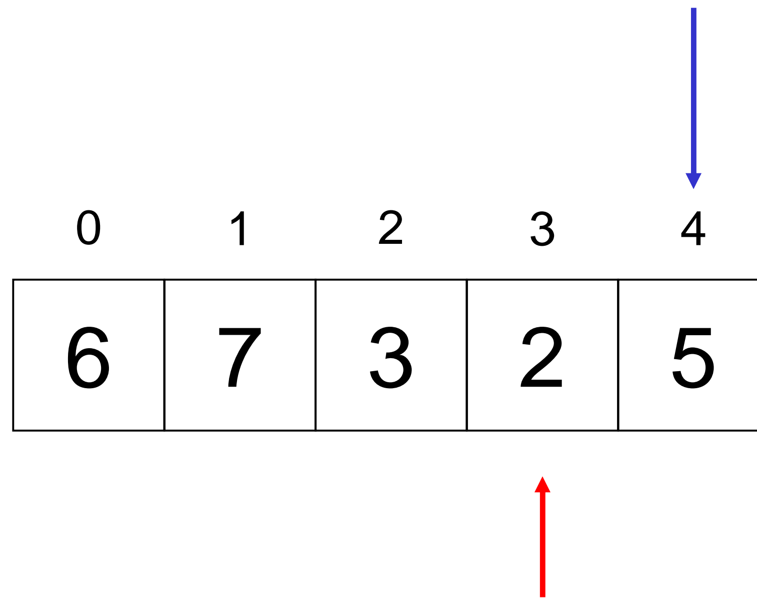
imax: 0

max: 6



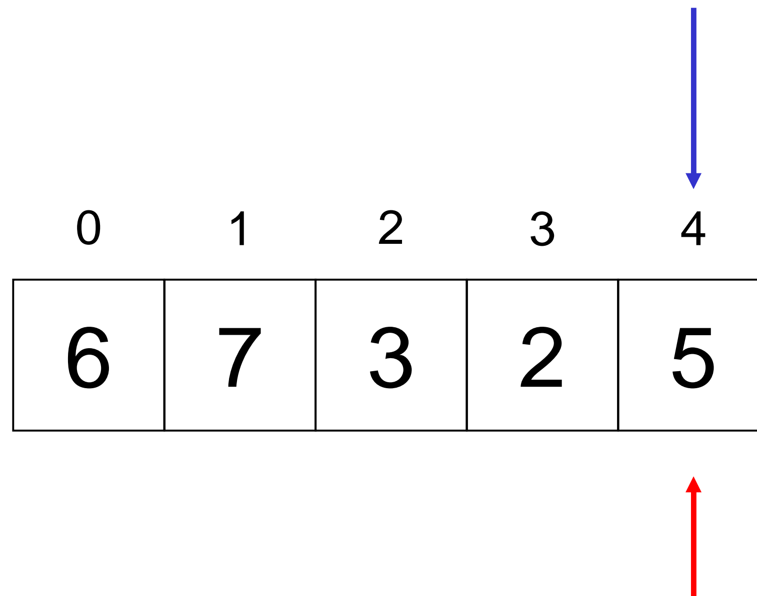
imax: 1

max: 7



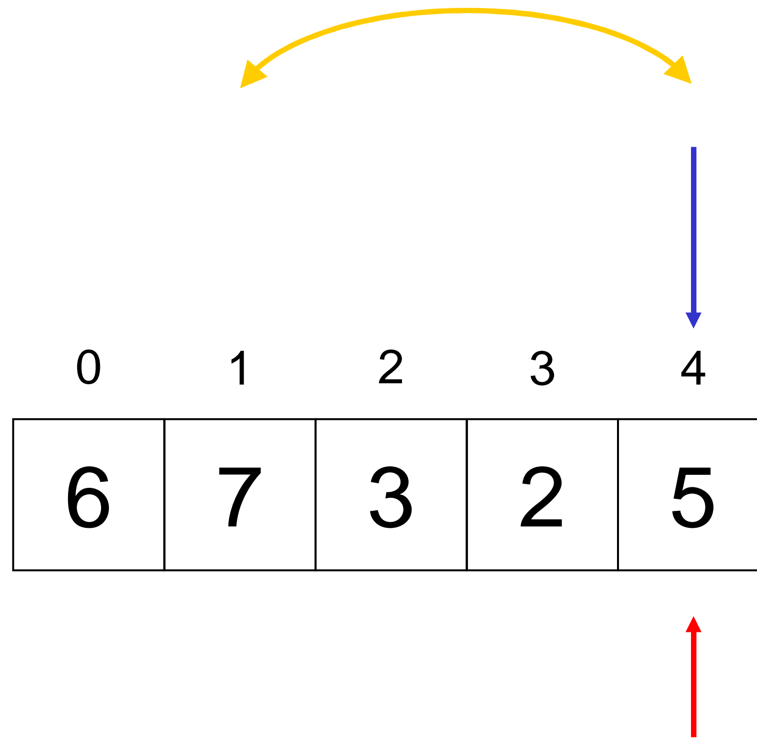
imax: 1

max: 7



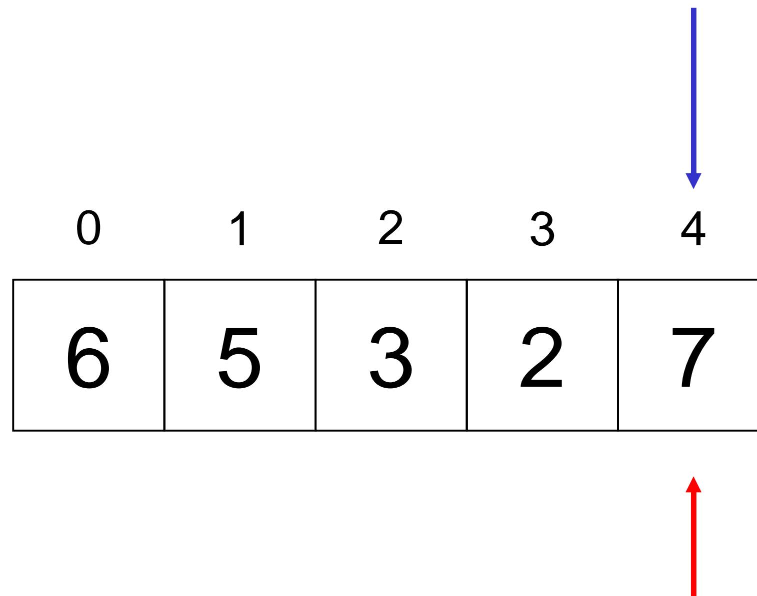
imax: 1

max: 7



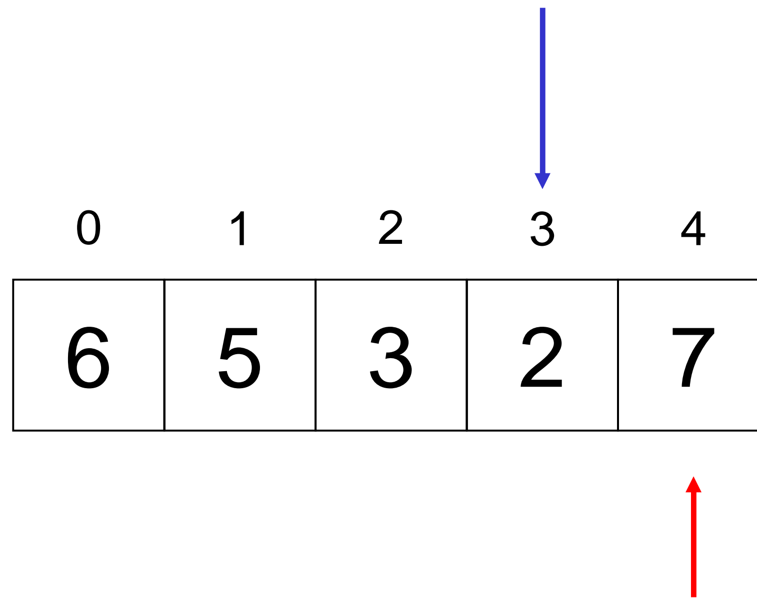
imax: 1

max: 7



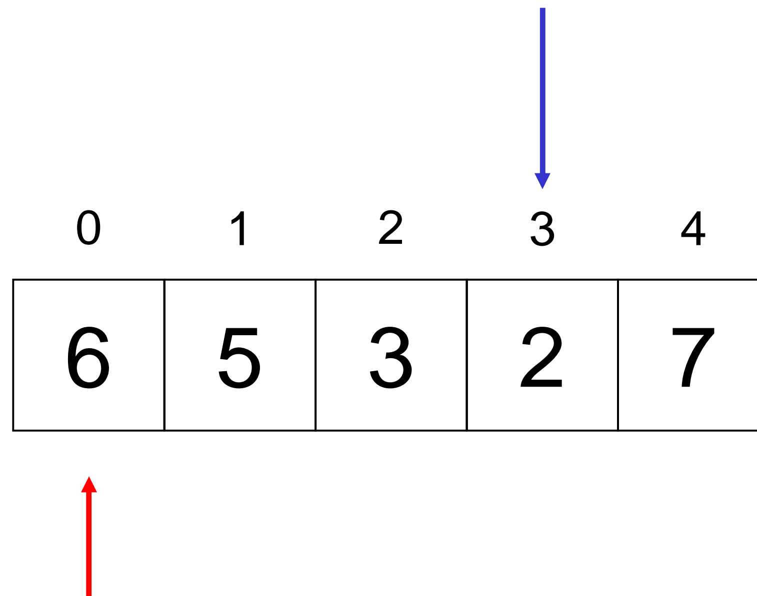
imax: 1

max: 7



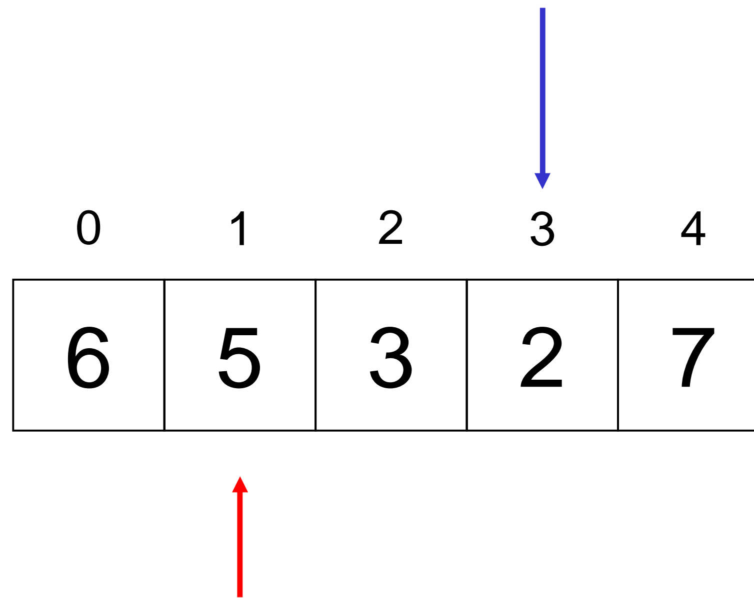
imax: 1

max: 7



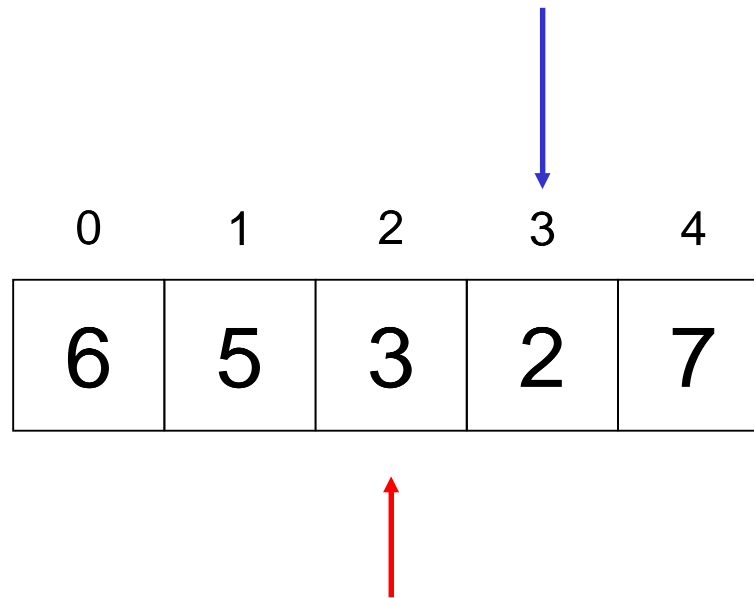
imax: 0

max: 6



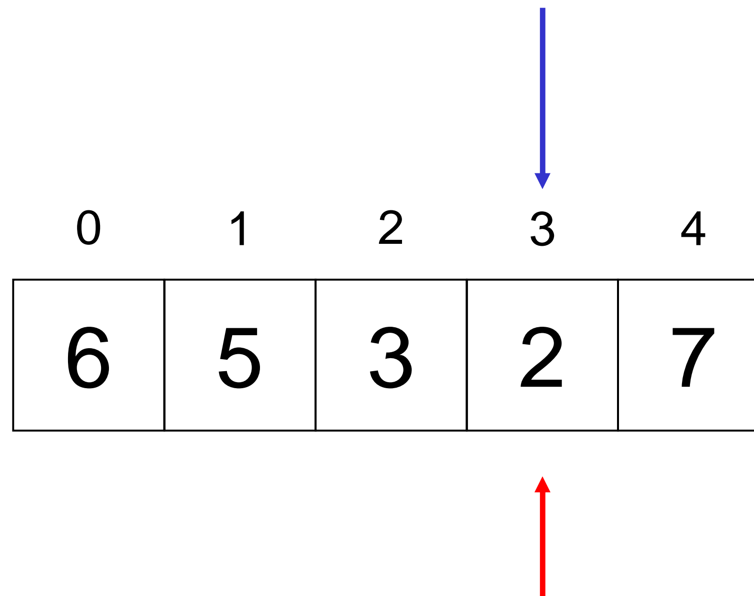
imax: 0

max: 6



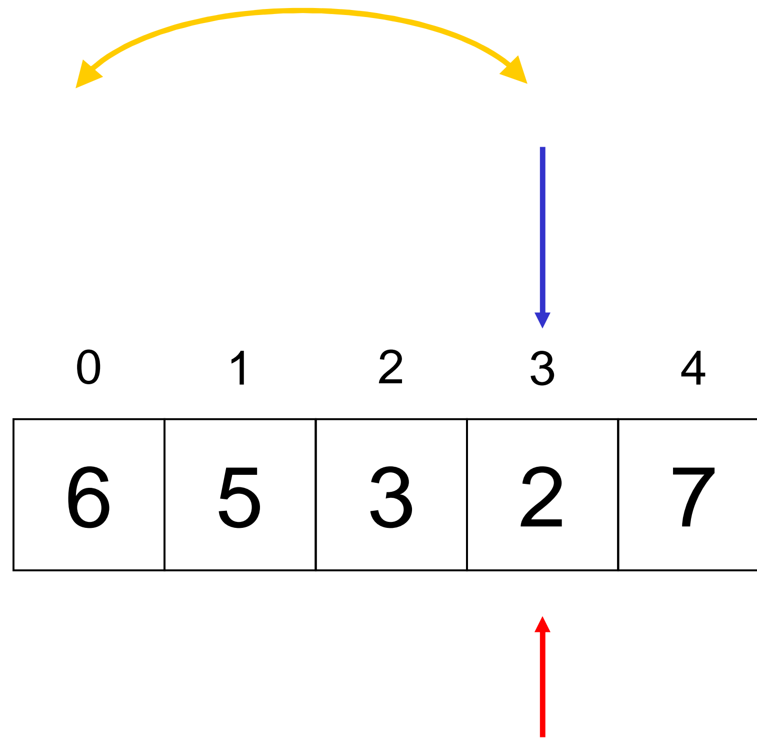
imax: 0

max: 6



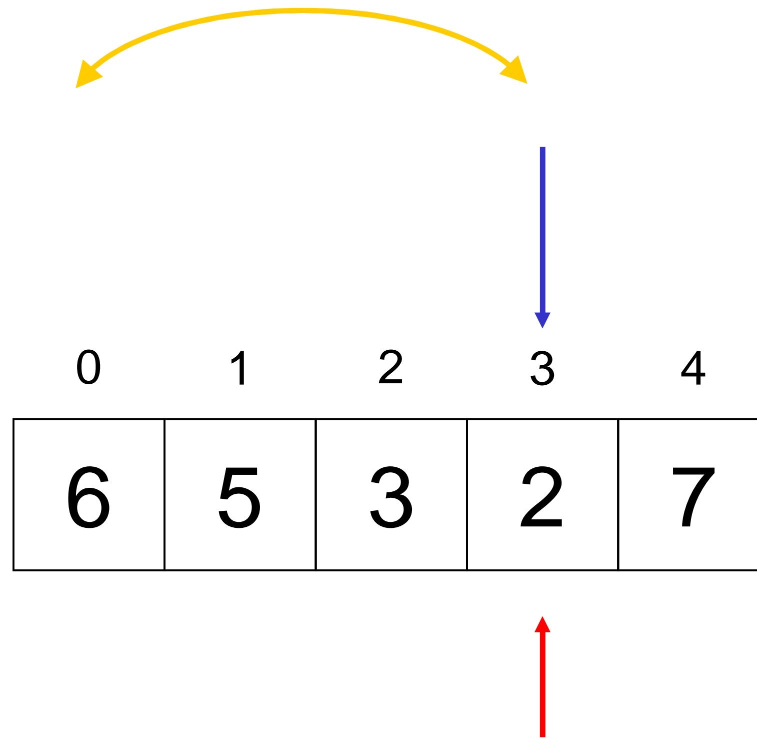
imax: 0

max: 6



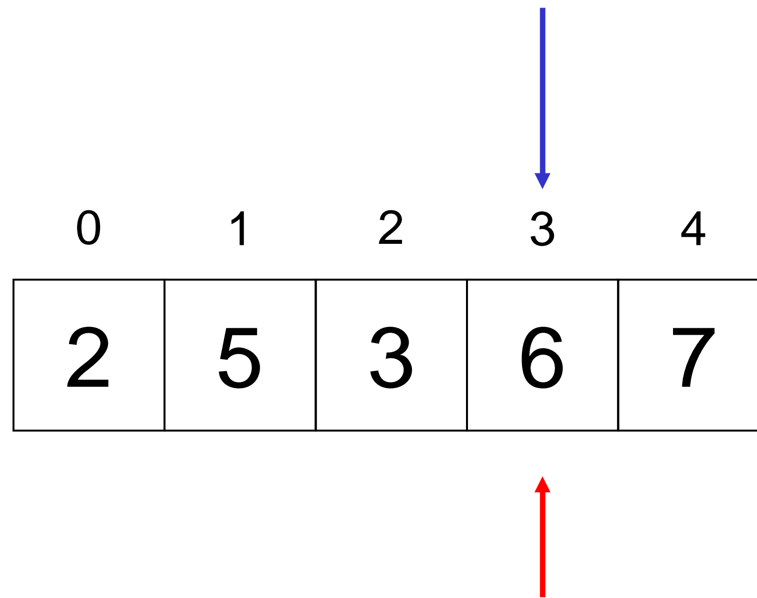
imax: 0

max: 6



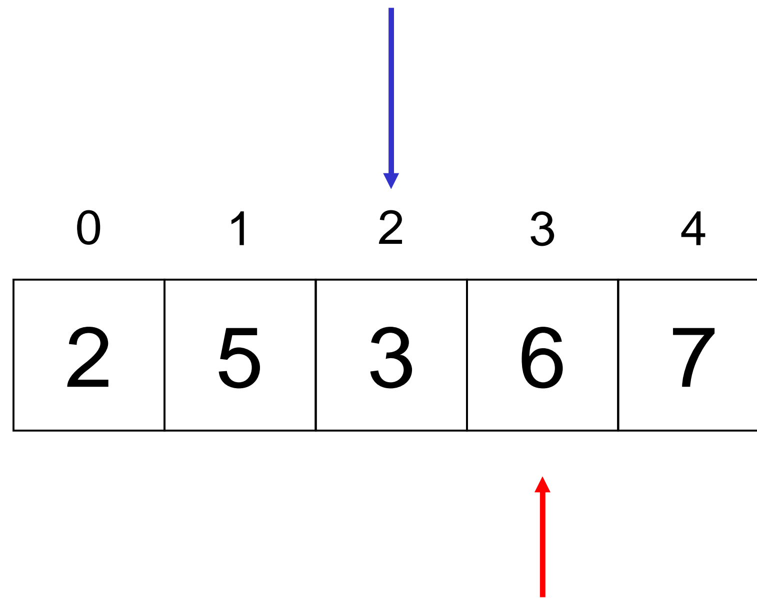
imax: 0

max: 6



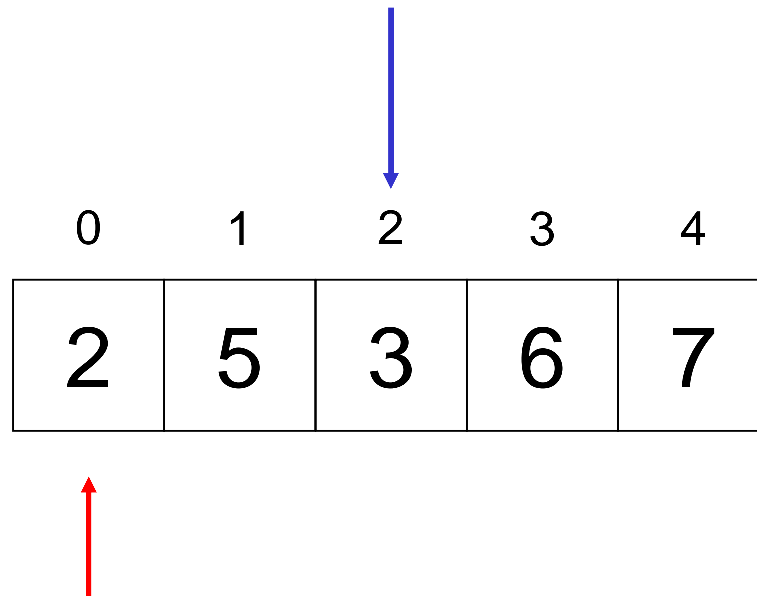
imax: 0

max: 6



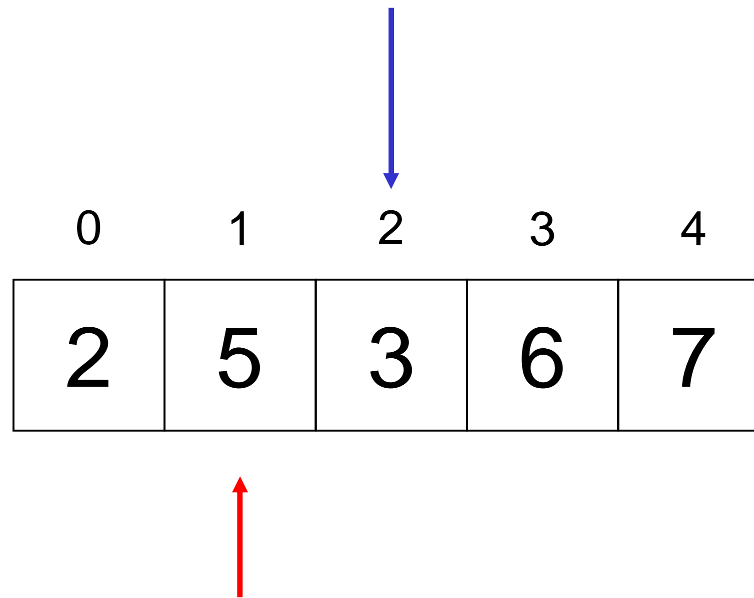
imax: 0

max: 6



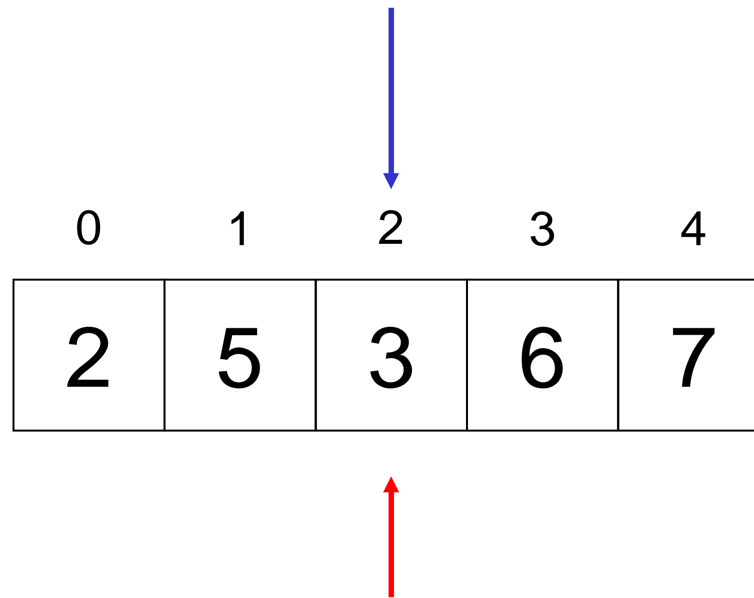
imax: 0

max: 2



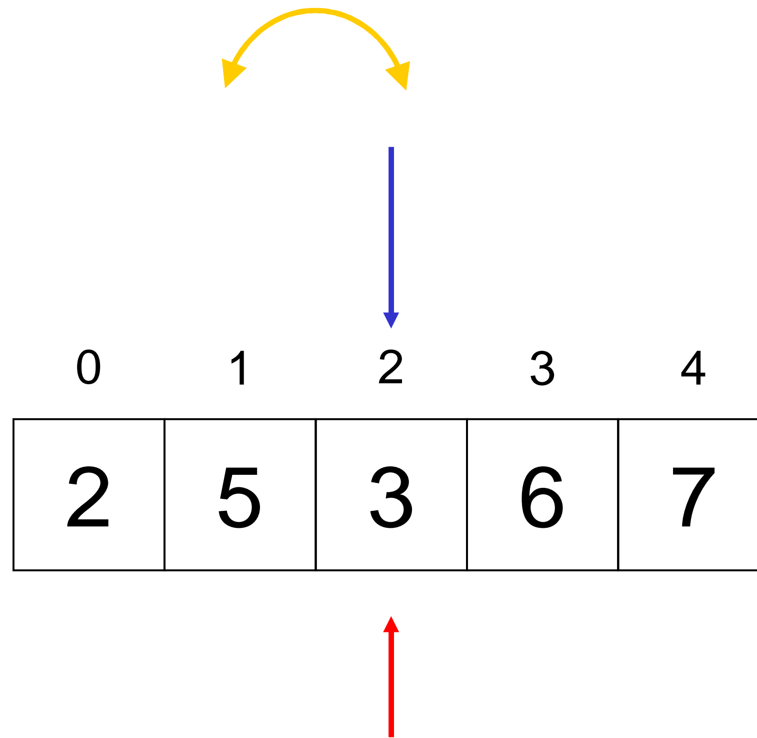
imax: 0

max: 2



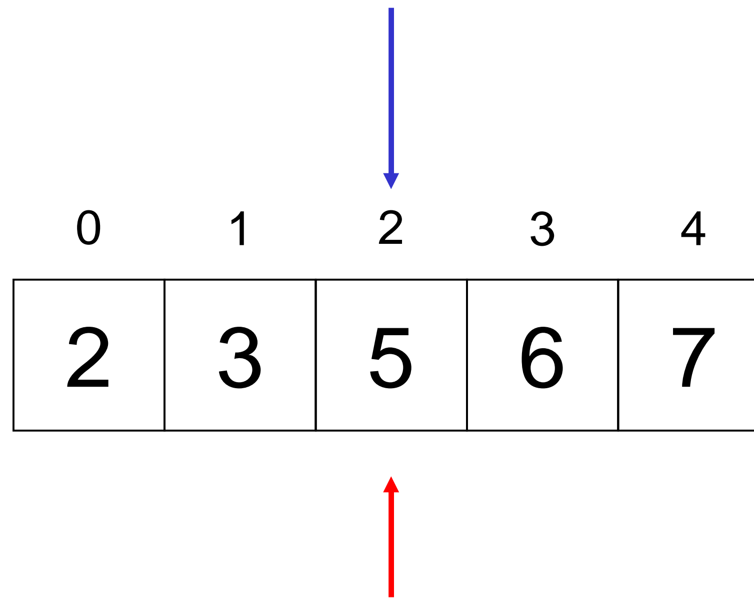
imax: 1

max: 5



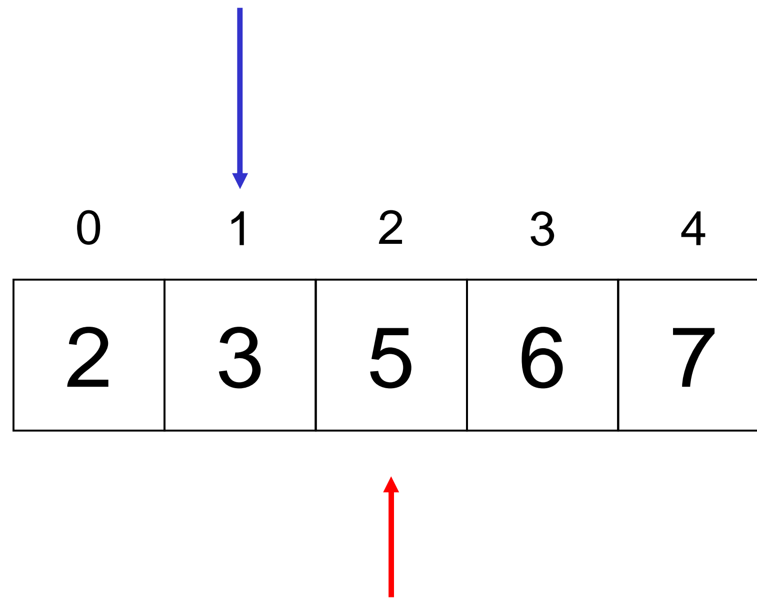
imax: 1

max: 5



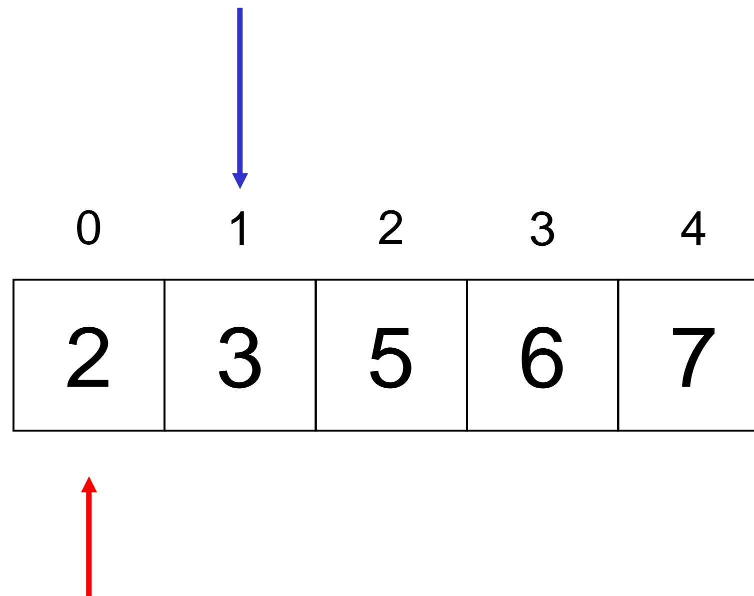
imax: 1

max: 5



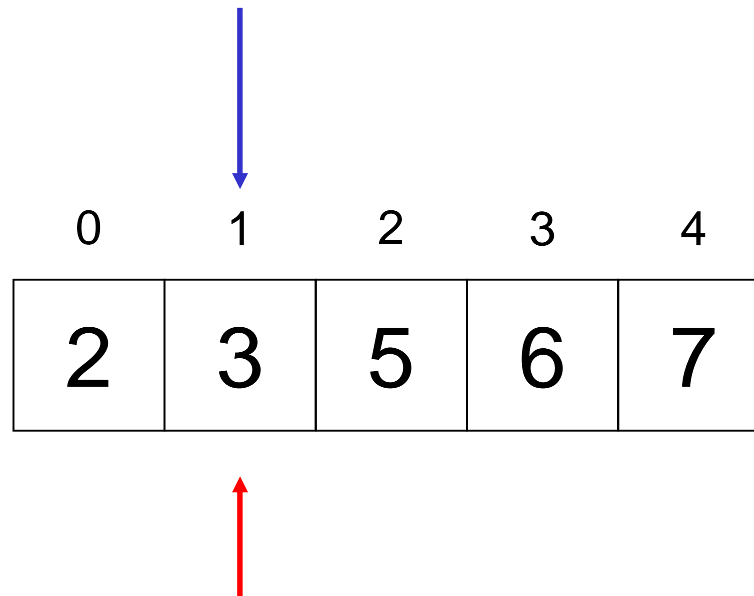
imax: 1

max: 5



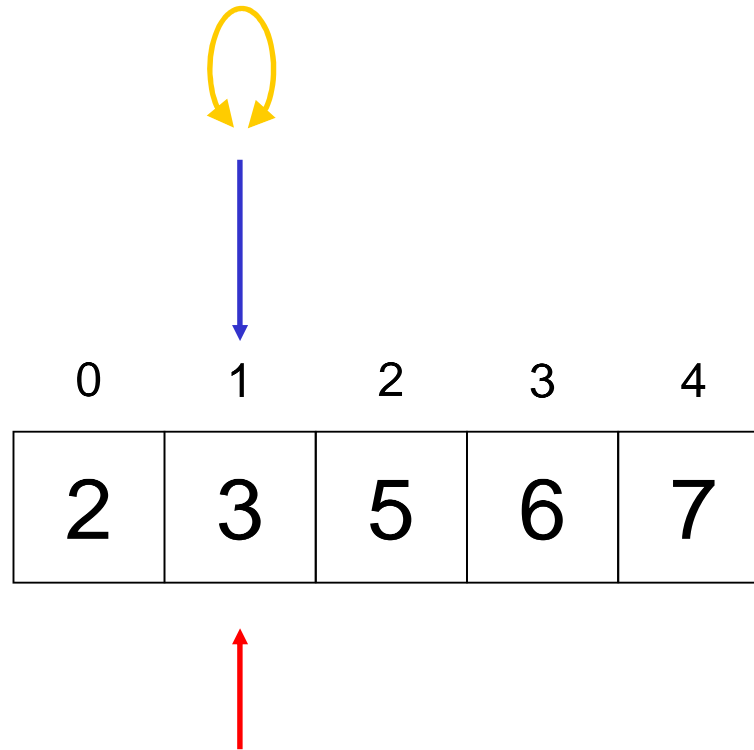
imax: 0

max: 2



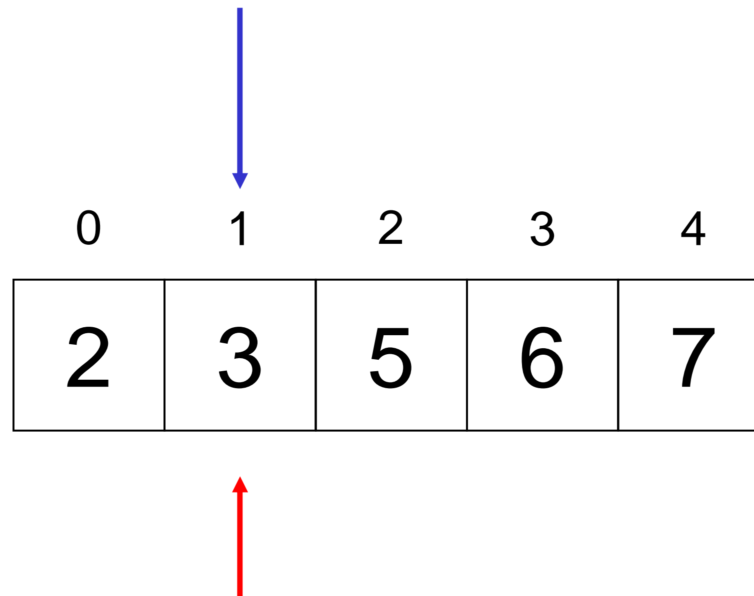
imax: 0

max: 2



imax: 1

max: 3



imax: 1

max: 3

0	1	2	3	4
2	3	5	6	7

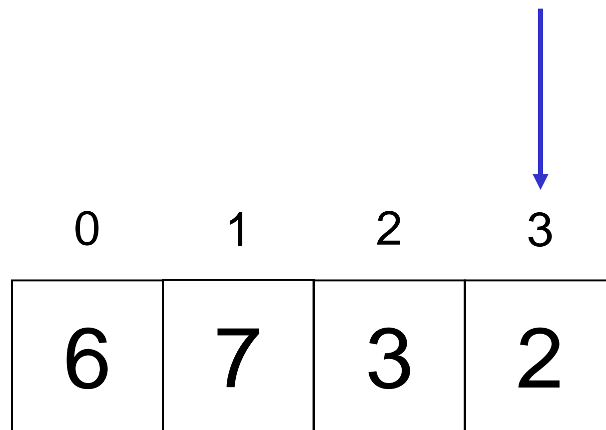
imax: 1

max: 3

```
void razeni_max_prvek(int *pole, int n)
{
  int i,j,index_max,d;
  for(i=n-1;i>=1;i--)
  {
    index_max = 0;
    for(j=1;j<=i;j++)
      if(pole[j]>pole[index_max]) index_max=j;
    d=pole[index_max]; pole[index_max]=pole[i];
  pole[i]=d;
  }
}
```

Řazení záměnou (bublinkové řazení) (Bubble Sort)

- porovnáváme postupně v celém poli dva sousední prvky; pokud nejsou ve správném pořadí, zaměníme je
- po tomto kroku je na posledním místě pole největší prvek („probublá” na konec)
- krok algoritmu probublávání aplikujeme na postupně na pole o délce n-1, n-2,…,1


0
1
2
3
6
7
3
2

porovnám

0	1	2	3
6	7	3	2

porovnám

0	1	2	3
6	7	3	2

prohodím

0	1	2	3
6	7	3	2

prohodím

0	1	2	3
6	3	7	2

porovnám

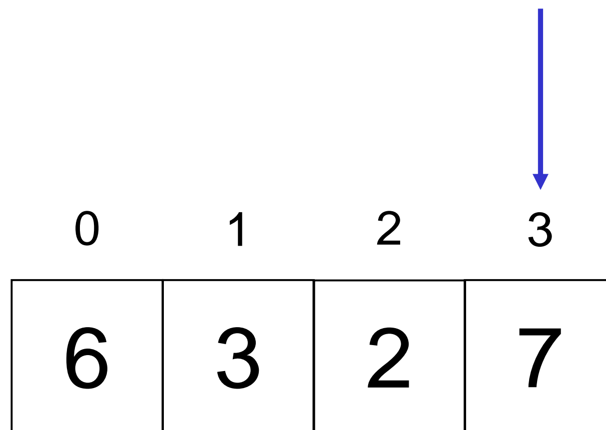
0	1	2	3
6	3	7	2

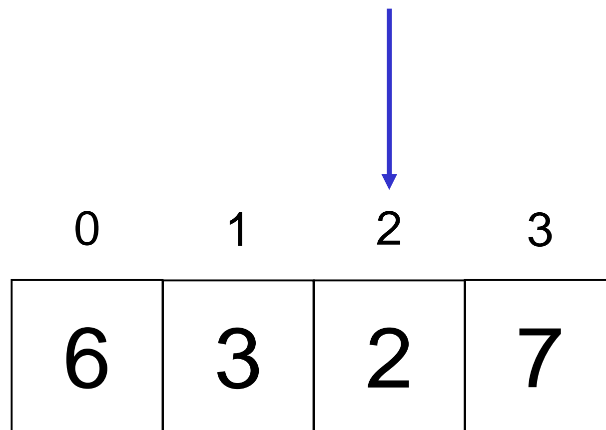
prohodím

0	1	2	3
6	3	7	2

prohodím

0	1	2	3
6	3	2	7


0
1
2
3
6
3
2
7


0
1
2
3
6
3
2
7

porovnám

0	1	2	3
6	3	2	7

prohodím

0	1	2	3
6	3	2	7

prohodím

0	1	2	3
3	6	2	7

porovnám

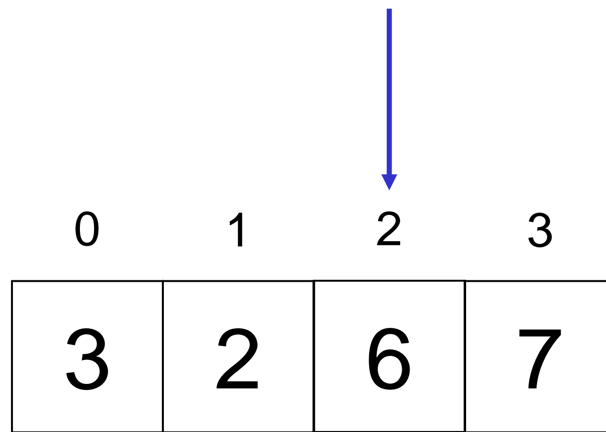
0	1	2	3
3	6	2	7

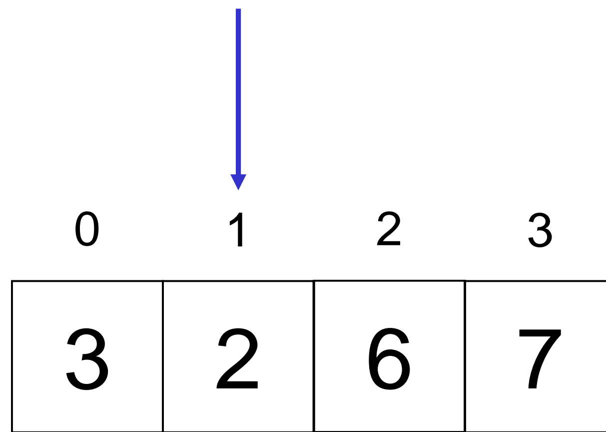
prohodím

0	1	2	3
3	6	2	7

prohodím

0	1	2	3
3	2	6	7


0
1
2
3
3
2
6
7


0
1
2
3
3
2
6
7

porovnám

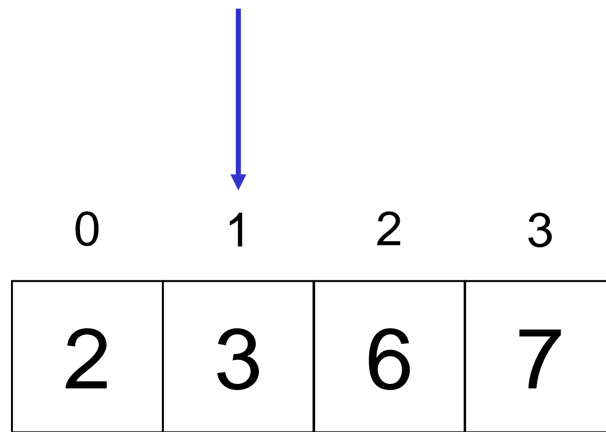
0	1	2	3
3	2	6	7

prohodím

0	1	2	3
3	2	6	7

prohodím

0	1	2	3
2	3	6	7



0
1
2
3
2
3
6
7

```
void bublinka(int *pole, int n)
{
  int i,j,d;
  for(i=n-1;i>=1;i--)
  {
    for(j=0;j<=i-1;j++)
      if(pole[j]>pole[j+1])
      {
       d=pole[j];
       pole[j]=pole[j+1];
       pole[j+1]=d;
      }
  }
}
```

- bublinkové řazení se dá vylepšit:
  - pokud jsme při průchodu polem neprovedli ani jedenkrát záměnu, pole je seřazené a cyklus předčasně ukončíme
  - bublání provádíme oběma směry (i k nižším indexům)

Domácí úkol: vylepšete bublinkové řazení

```c
void bublinka2(int *pole, int n)
{
  int i,j,d;
  int prohozeno=1;
  for(i=n-1;i>=1 && prohozeno;i--)
  {
    prohozeno = 0;
    for(j=0;j<=i-1;j++)
      if(pole[j]>pole[j+1])
      {
       prohozeno = 1;
       d=pole[j];
       pole[j]=pole[j+1];
       pole[j+1]=d;
      }
  }
}
```

- odhadněte operační složitost bublinkového řazení a řazení výběrem maximálního prvku

$O(n^2)$

- problémy, které lze řešit (vyzkoušením všech možností) v polynomiálním čase, se nazývají **P-problémy**
- mnoho problémů v polynomiálním čase řešit nelze (mají nepolynomiální složitost)
  – u některých existuje ale *nedeterministický algoritmus, který je řeší v polynomiálním čase (NP – nedeterministicky polynomiální)*
- není znám deterministický algoritmus pro nalezení řešení: **NP- úplné problémy (NP- complete)**

Bohužel, většina reálných a zajímavých problémů (i dopravních) je NP-úplných

- řeší se přibližnými metodami (heuristikami), metodami umělé inteligence, genetickými algoritmy aj.

Příklad NP problémů

- problém obchodního cestujícího
  - odvozený: souřadnicová vrtačka
- testování obvodů - úplný test
- splnitelnost booleovských formulí
- hledání podmnožiny dep
- hledání všech cest mezi dvěma body
- Hanojské věže

Quick Sort

- nejrychlejší algoritmus řazení
- založen na technice řešení problémů *rozděl a panuj (divide and conquer)*
  - oblast řešení se rozdělí na dvě (stejně) velké oblasti, vyřeší se na každé polovině zvlášť a spojí se do výsledného řešení

Aplikace rozděl a panuj na Quick-Sort

1. Vyber pivot – prvek ve středu pole
2. Přeskup pole – prvky menší než pivot přesuň nalevo od něj, prvky větší než pivot přesuň napravo od něj
3. Seřaď část pole nalevo a napravo od pivota

- řazení levé a pravé části pole se provádí stejnou technikou (výběr pivota,…) až k části pole o délce 1
- nelze naprogramovat jinak než ***rekurzí***

```
void Quick_sort(int l, int r, int *pole)
{
  int pivot,d,i,j;
  if (l < r)
  {
     i = l; j = r;
     pivot = pole[(l+r)/2];
     do
     {
       while (pole[i] < pivot) i++;
       while (pole[j] > pivot) j--;
       if (i < j) { d = pole[i];
                    pole[i] = pole[j];  pole[j] = d; }
     }
     while (i < j);
     Quick_sort(l,j-1,pole);
     Quick_sort(i+1,r,pole);
  }
}
```

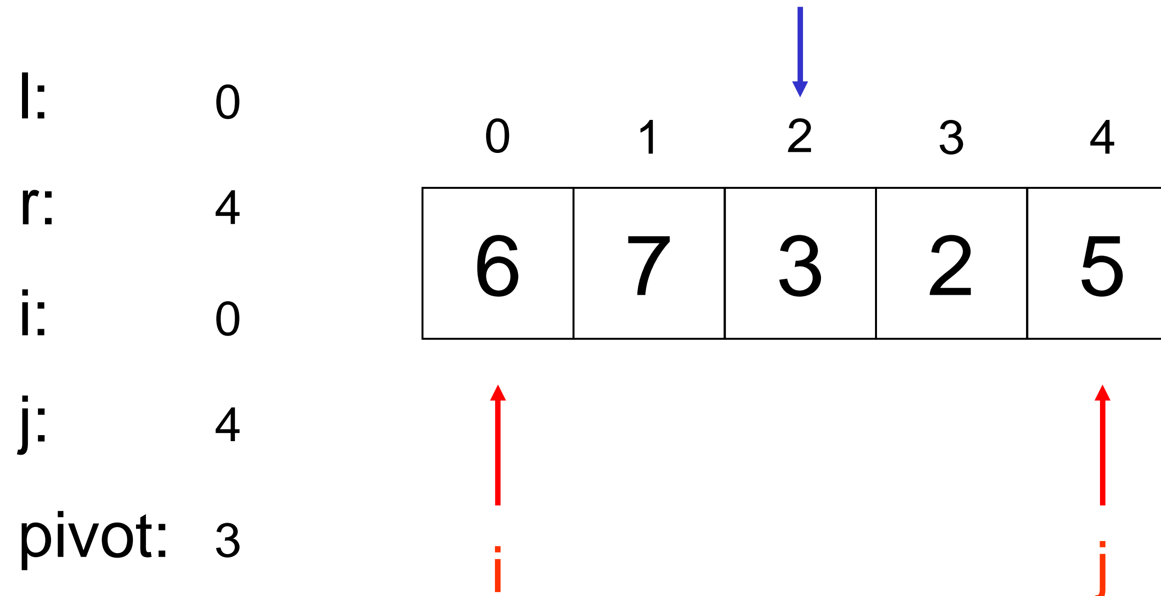
- volání funkce

```
Quick_sort(0,n-1,pole);
```

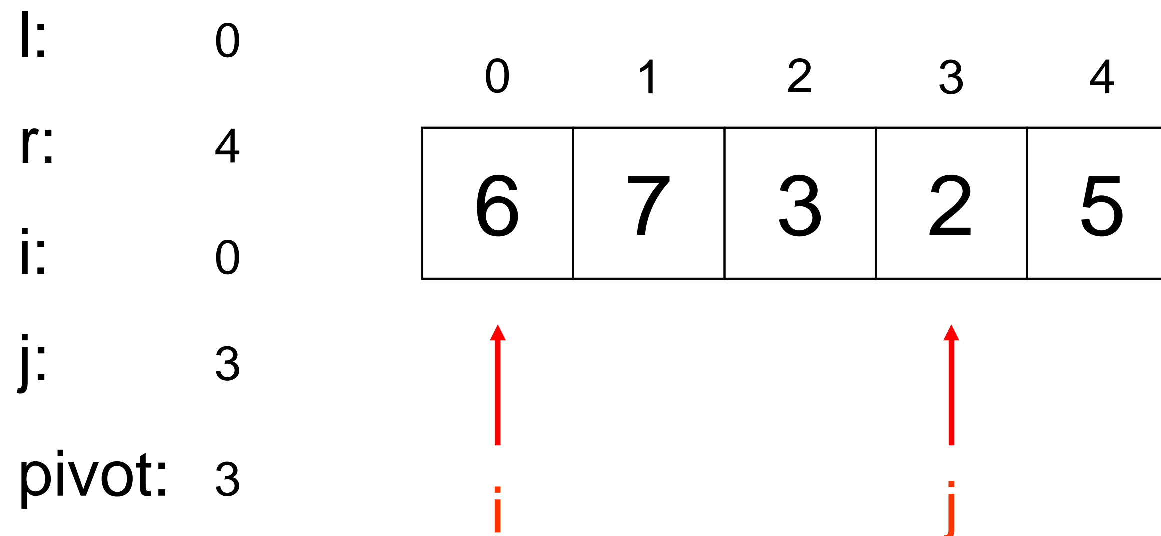
- nejrychlejší algoritmus řazení, složitost **$O(n\log_2 n)$**

- poznámka: často se při řazení nepřehazují prvky, ale vytváří se *pole indexů*

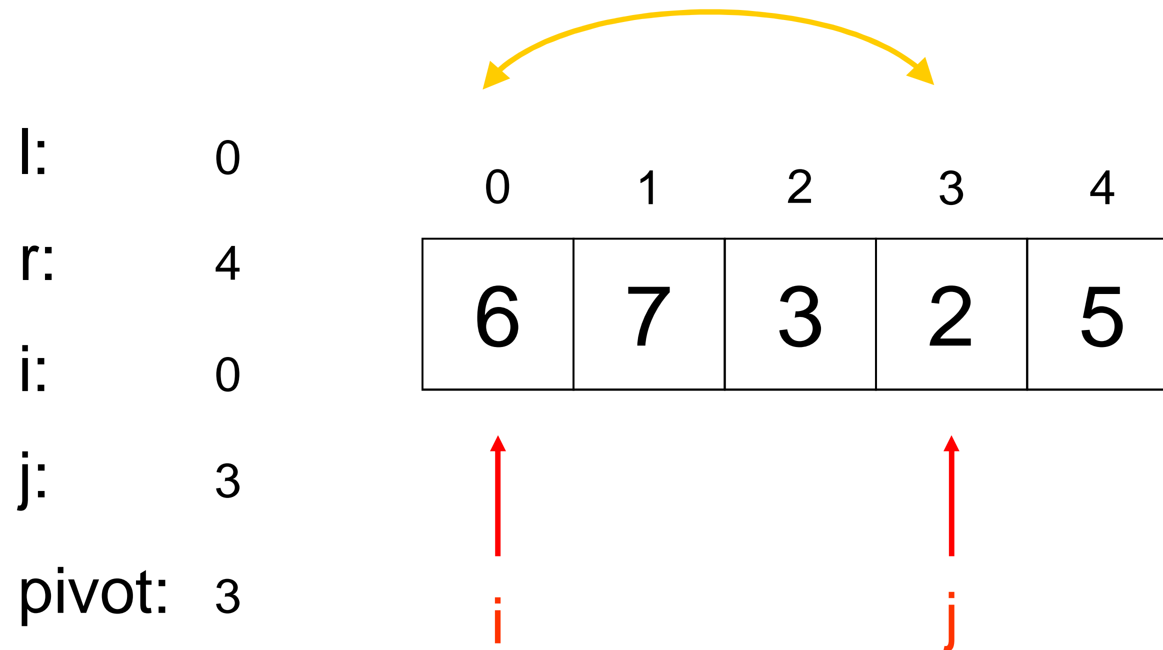
```
do
{
  while (pole[i] < pivot) i++;
  while (pole[j] > pivot) j--;
  if (i < j) { d = pole[i];
                pole[i] = pole[j];
                pole[j] = d;
              }
} while (i < j);
```



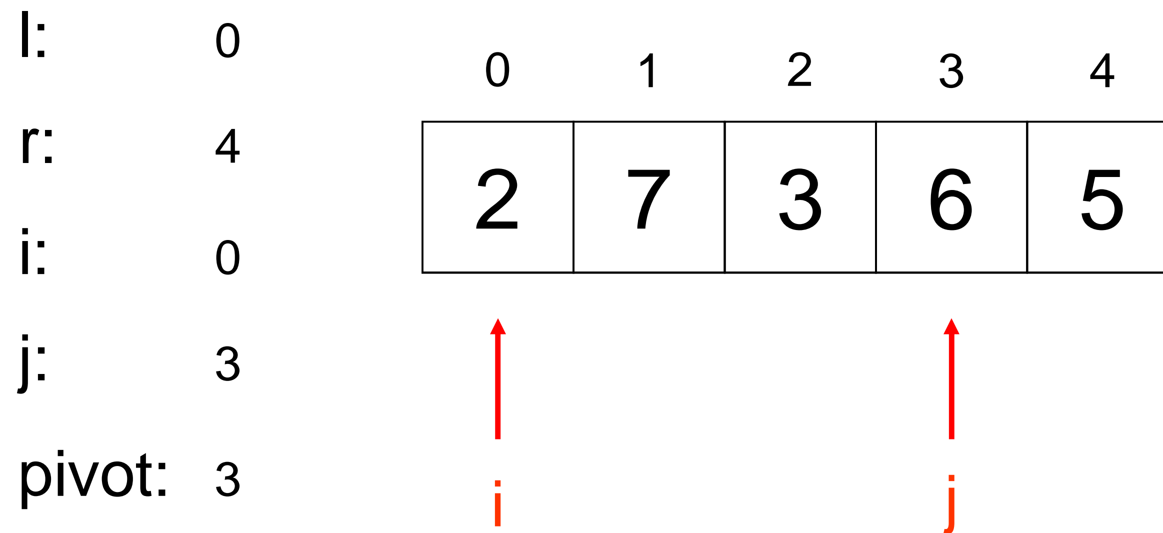
```
do
{
  while (pole[i] < pivot) i++;
  while (pole[j] > pivot) j--;
  if (i < j) { d = pole[i];
               pole[i] = pole[j];
               pole[j] = d;
             }
} while (i < j);
```



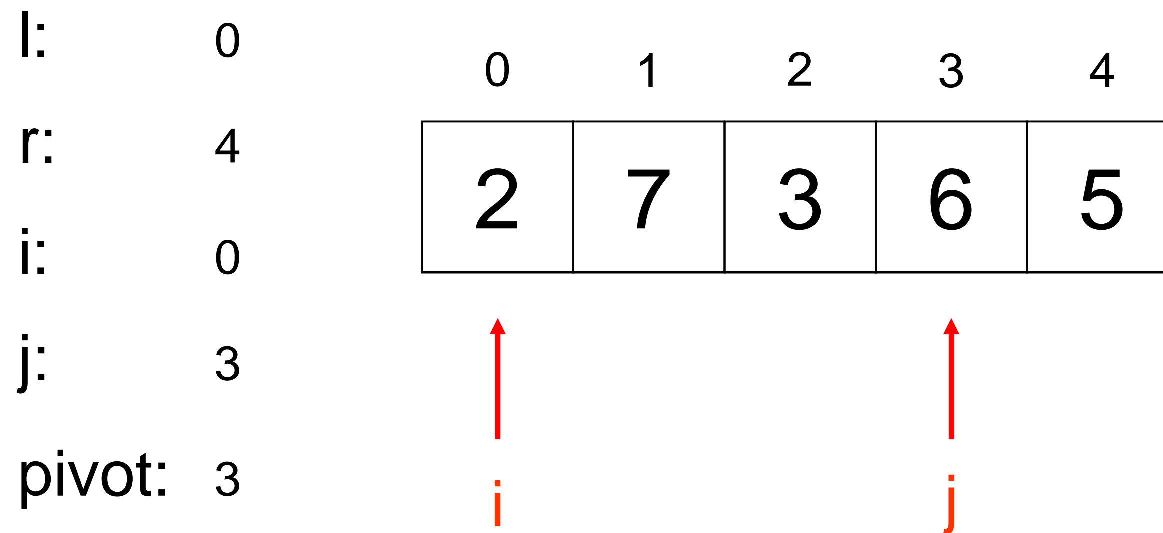
```
do
{
  while (pole[i] < pivot) i++;
  while (pole[j] > pivot) j--;
  if (i < j) { d = pole[i];
               pole[i] = pole[j];
               pole[j] = d;
             }
} while (i < j);
```



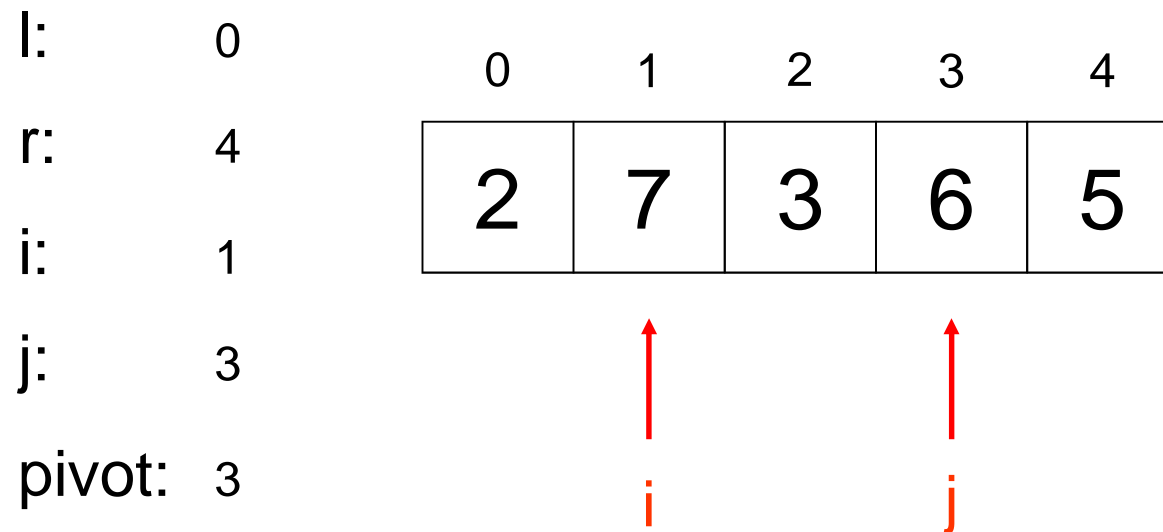
```
do
{
  while (pole[i] < pivot) i++;
  while (pole[j] > pivot) j--;
  if (i < j) { d = pole[i];
               pole[i] = pole[j];
               pole[j] = d;
             }
} while (i < j);
```



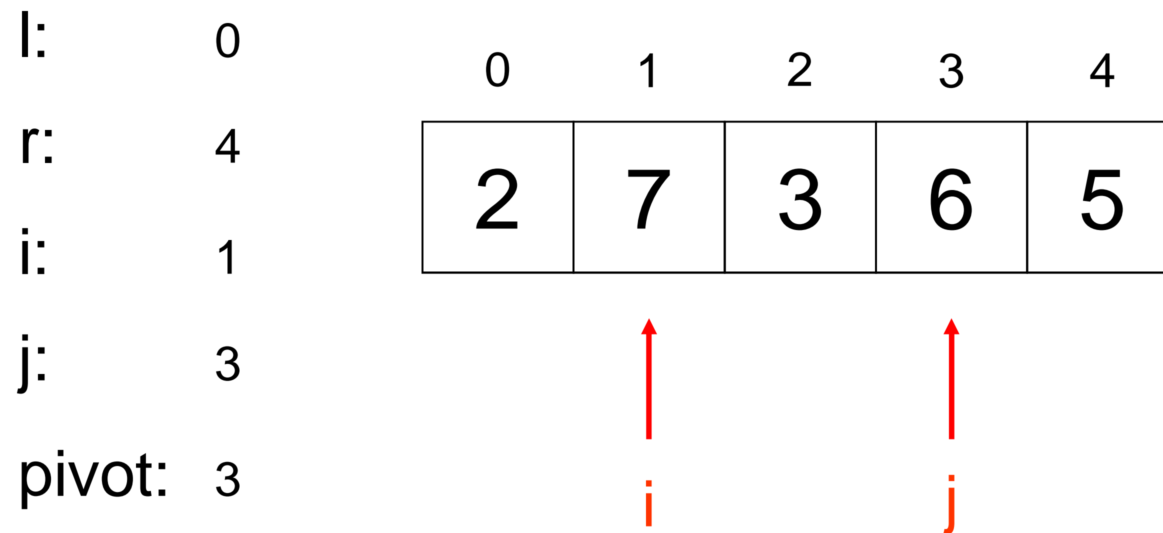
```
do
{
  while (pole[i] < pivot) i++;
  while (pole[j] > pivot) j--;
  if (i < j) { d = pole[i];
               pole[i] = pole[j];
               pole[j] = d;
             }
} while (i < j);
```



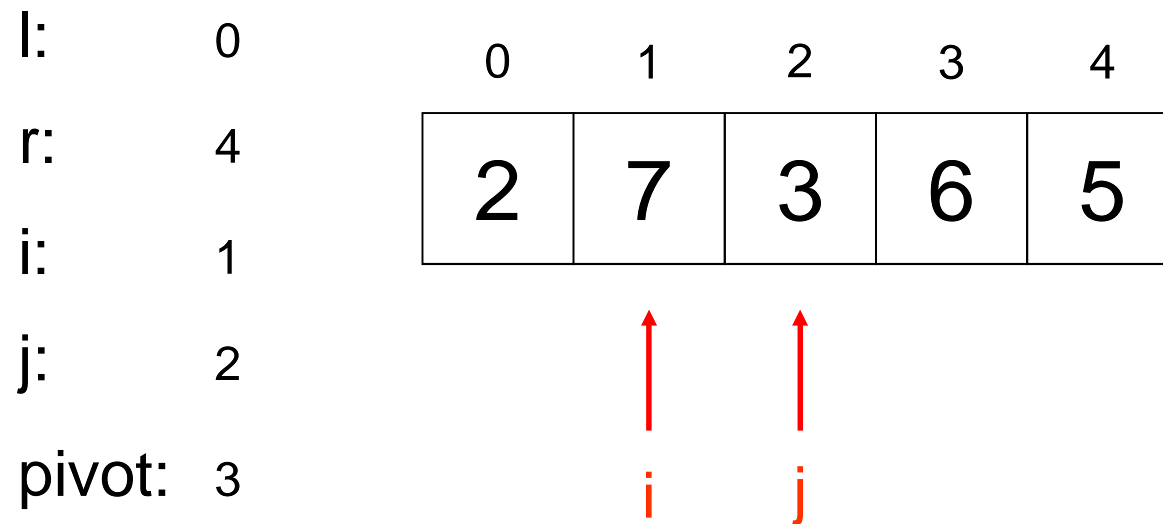
```
do
{
  while (pole[i] < pivot) i++;
  while (pole[j] > pivot) j--;
  if (i < j) { d = pole[i];
                pole[i] = pole[j];
                pole[j] = d;
              }
} while (i < j);
```



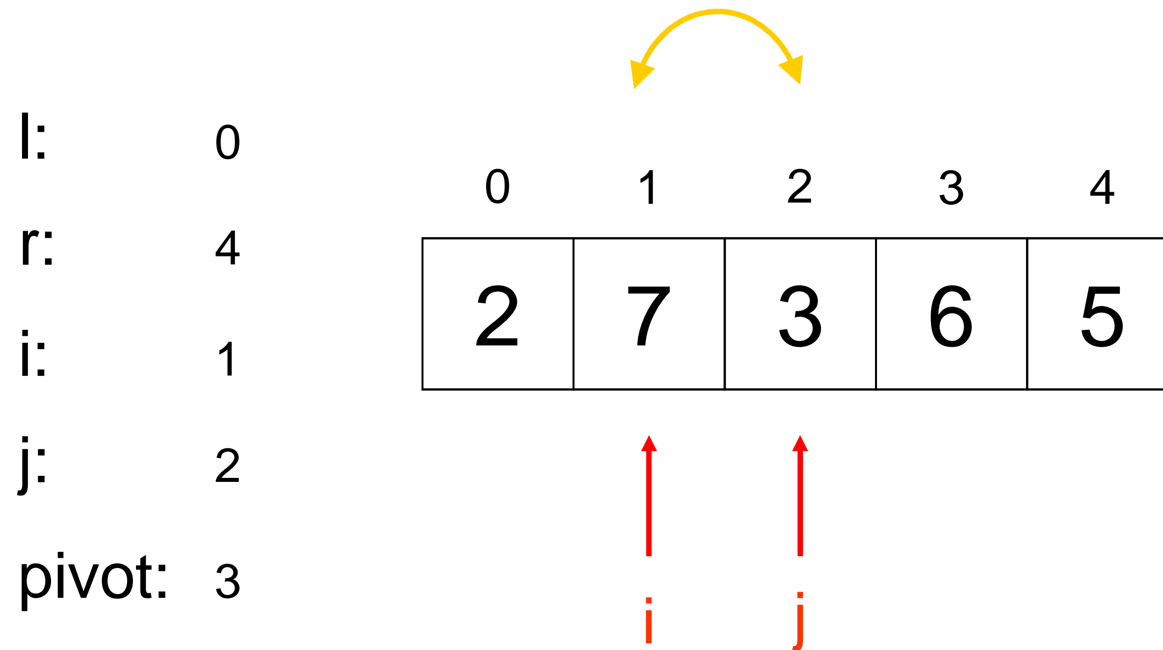
```
do
{
  while (pole[i] < pivot) i++;
  while (pole[j] > pivot) j--;
  if (i < j) { d = pole[i];
               pole[i] = pole[j];
               pole[j] = d;
             }
} while (i < j);
```



```
do
{
  while (pole[i] < pivot) i++;
  while (pole[j] > pivot) j--;
  if (i < j) { d = pole[i];
               pole[i] = pole[j];
               pole[j] = d;
             }
} while (i < j);
```



```
do
{
   while (pole[i] < pivot) i++;
   while (pole[j] > pivot) j--;
   if (i < j) { d = pole[i];
                pole[i] = pole[j];
                pole[j] = d;
             }
} while (i < j);
```



```
do
{
  while (pole[i] < pivot) i++;
  while (pole[j] > pivot) j--;
  if (i < j) { d = pole[i];
               pole[i] = pole[j];
               pole[j] = d;
             }
} while (i < j);
```

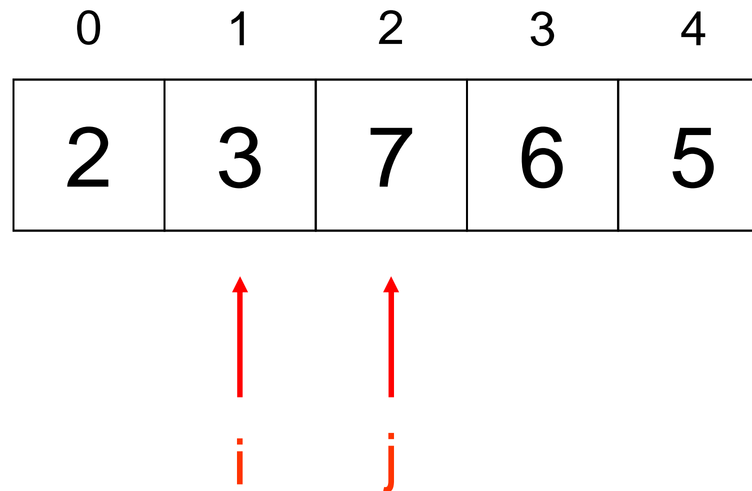
l: 0

r: 4

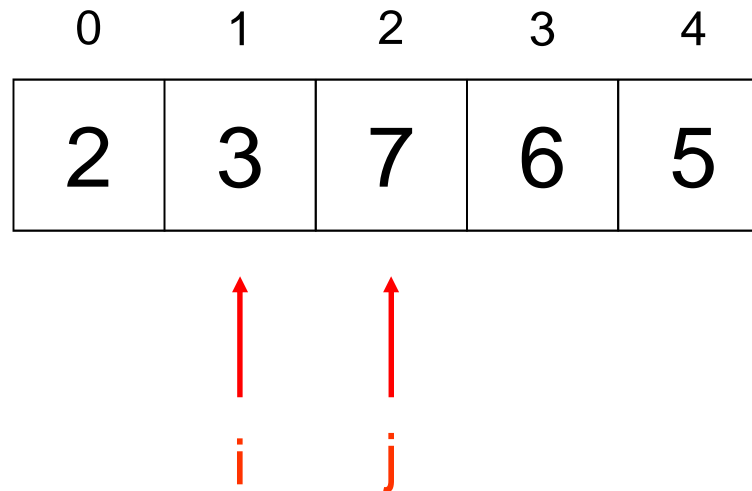
i: 1

j: 2

pivot: 3



```
do
{
  while (pole[i] < pivot) i++;
  while (pole[j] > pivot) j--;
  if (i < j) { d = pole[i];
               pole[i] = pole[j];
               pole[j] = d;
             }
} while (i < j);
```



```
do
{
  while (pole[i] < pivot) i++;
  while (pole[j] > pivot) j--;
  if (i < j) { d = pole[i];
               pole[i] = pole[j];
               pole[j] = d;
             }
} while (i < j);
```

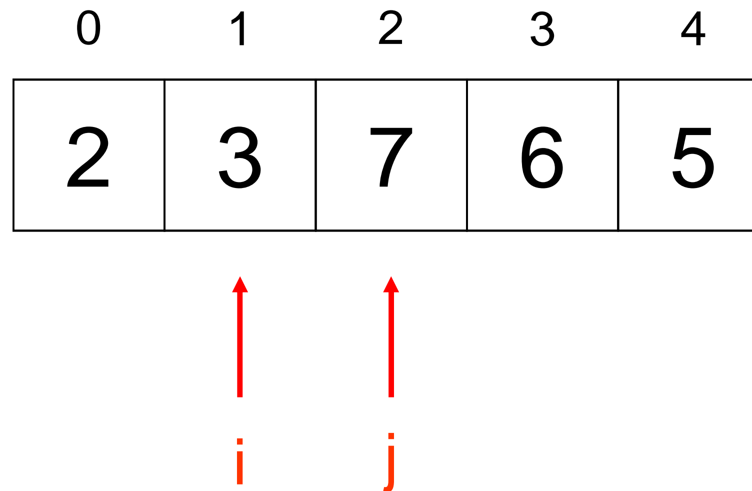
l: 0

r: 4

i: 1

j: 2

pivot: 3



```
do
{
  while (pole[i] < pivot) i++;
  while (pole[j] > pivot) j--;
  if (i < j) { d = pole[i];
               pole[i] = pole[j];
               pole[j] = d;
             }
} while (i < j);
```

l: 0

r: 4

i: 1

j: 1

pivot: 3

0	1	2	3	4
2	3	7	6	5

i j

```
do
{
  while (pole[i] < pivot) i++;
  while (pole[j] > pivot) j--;
  if (i < j) { d = pole[i];
               pole[i] = pole[j];
               pole[j] = d;
             }
} while (i < j);
QuickSort(l,j-1,pole);
QuickSort(j+1,r,pole);
```

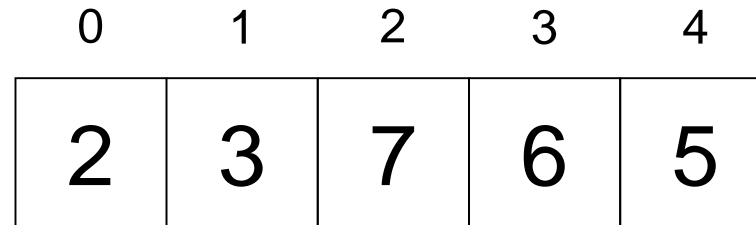
l: 0

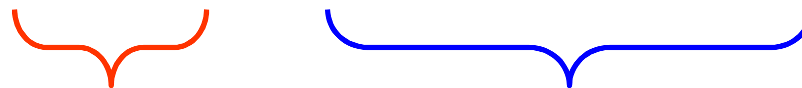
r: 4

i: 1

j: 1

pivot:





QuickSort(0,0,pole) **QuickSort(2,4,pole)**

Mergesort
(řazení slučováním)

1. Rozdělíme řazené pole na poloviny
2. Seřadíme každou polovinu
3. Obě seřazené poloviny sloučíme

Mergesort

```
void Merge_sort(int l, int r, int *pole)
// l - levý index, r - pravý index včetně
{
  int i,j,k,q;
  int *s;

  if (l>=r) return;
  q = (l+r)/2;
  Merge_sort(l,q,pole);
  Merge_sort(q+1,r,pole);
```

Mergesort

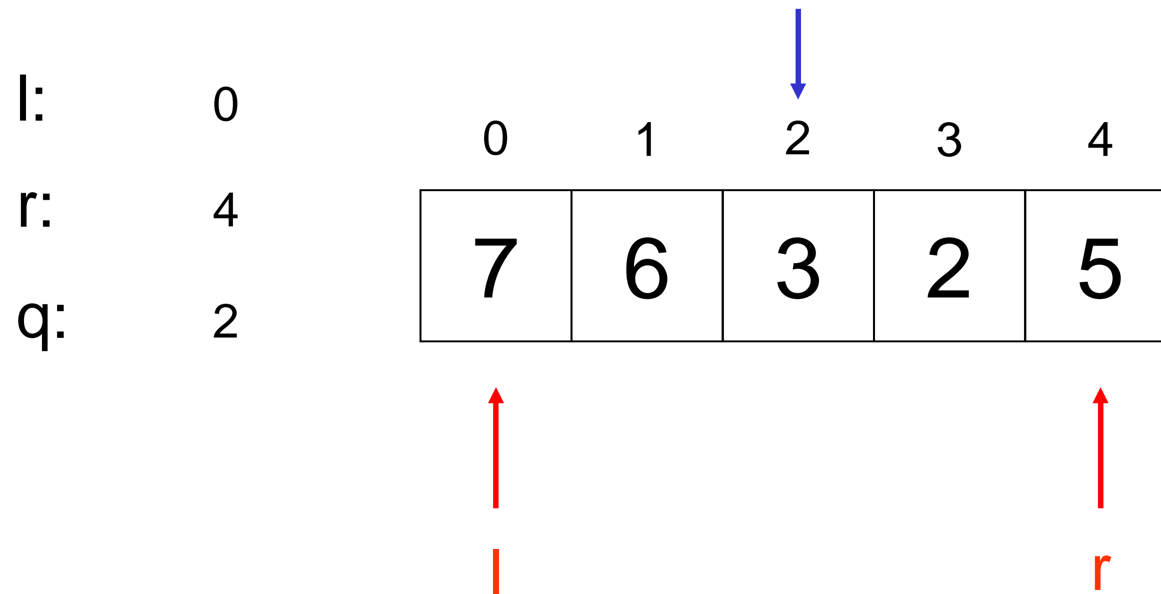
```
  // slučuji, s je pomocne pole
  s = (int*)malloc(sizeof(int)*(r-l+1));
  i = l; j = q+1;
  k = 0;
  while(i<=q && j <= r)
  {
      if (pole[i]<pole[j]) s[k++] = pole[i++];
      else s[k++] = pole[j++];
  }
  // kopirovani zbytku poli - probehne jen jeden z cyklu
  while (i<=q) s[k++] = pole[i++];
  while (j<=r) s[k++] = pole[j++];
  // kopirovani pole s zpet do casti pole od indexu l
  for(i=0;i<k;i++) pole[l++] = s[i];
  free(s);
}
```

- volání funkce

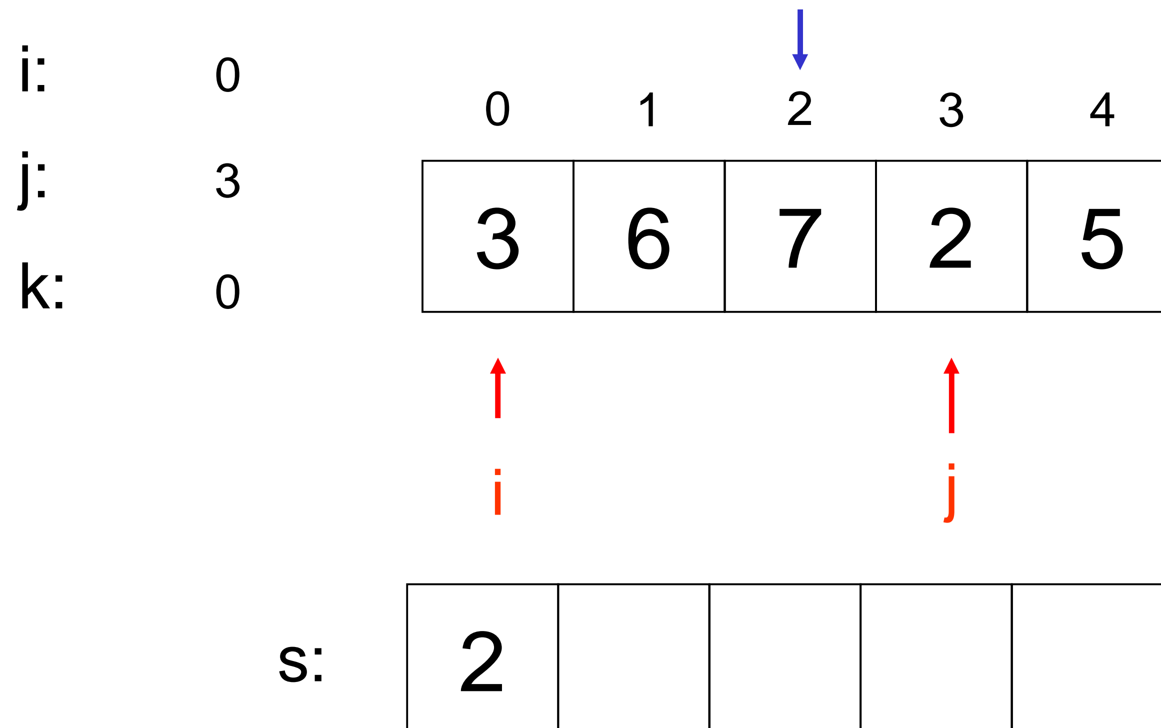
```
void razeni_merge(int n, int pole[])
// n - velikost pole
{
  Merge_sort(0,n-1,pole);
}
```

- složitost algoritmu řazení **$O(n\log_2 n)$**
- nevýhoda
  - nutnost existence pomocného pole (větší paměťová složitost)

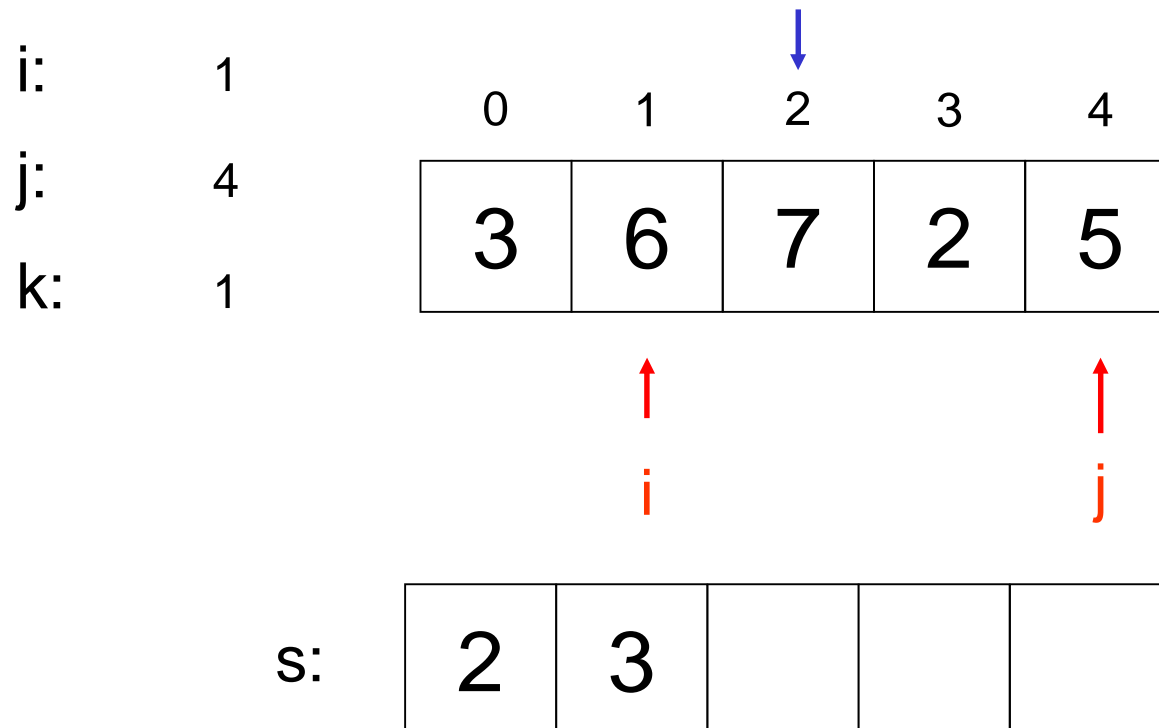
```
void Merge_sort(int l, int r, int *pole)
// l - levý index, r - pravý index včetně
{
  int i,j,k,q;
  int *s;

  if (l>=r) return;
  q = (l+r)/2;
  Merge_sort(l,q,pole);
  Merge_sort(q+1,r,pole);
```



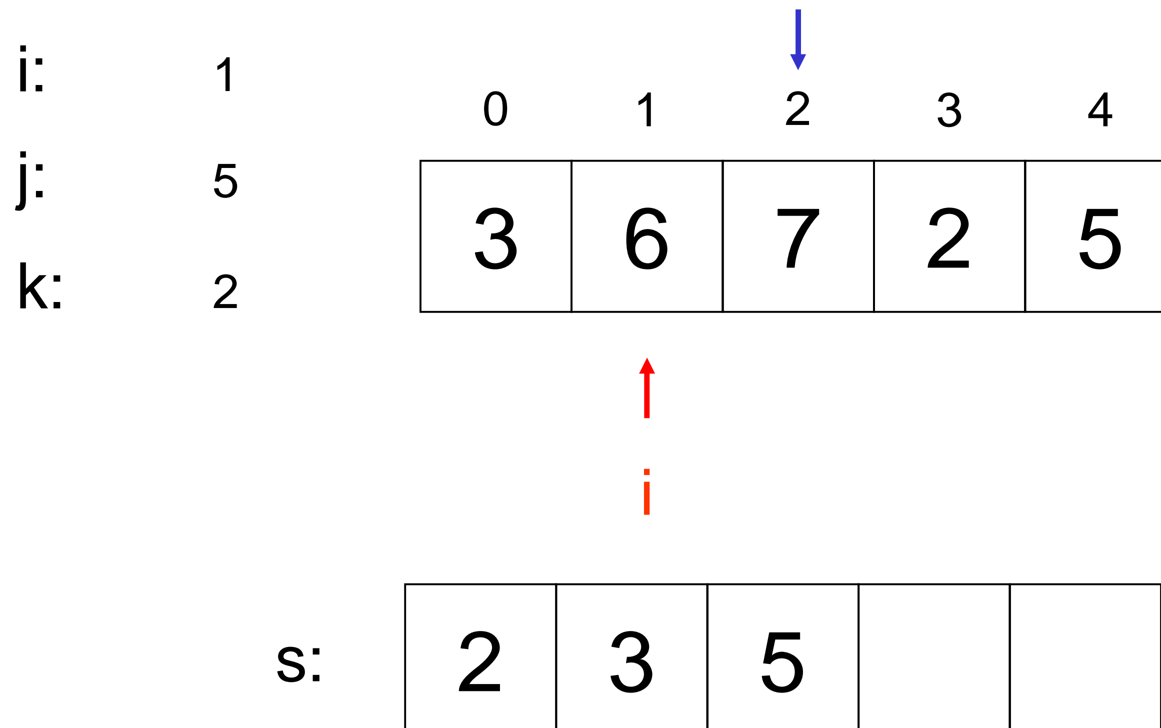
```
// slučuji, s je pomocne pole
  i = l; j = q+1;
  k = 0;
  while(i<=q && j <= r)
  {
      if (pole[i]<pole[j]) s[k++] = pole[i++];
      else s[k++] = pole[j++];
  }
  // kopirovani zbytku poli - probehne jen jeden z cyklu
  while (i<=q) s[k++] = pole[i++];
  while (j<=r) s[k++] = pole[j++];
```



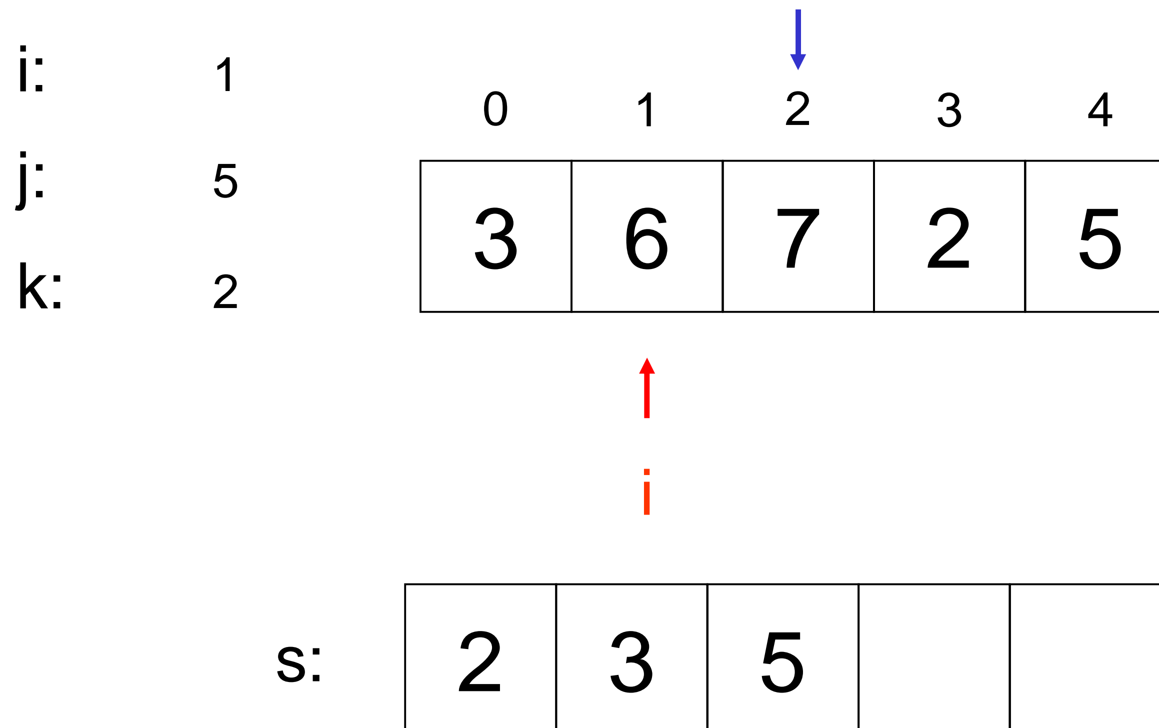
```
// slučuji, s je pomocne pole
  i = l; j = q+1;
  k = 0;
  while(i<=q && j <= r)
  {
      if (pole[i]<pole[j]) s[k++] = pole[i++];
      else s[k++] = pole[j++];
  }
  // kopirovani zbytku poli - probehne jen jeden z cyklu
  while (i<=q) s[k++] = pole[i++];
  while (j<=r) s[k++] = pole[j++];
```



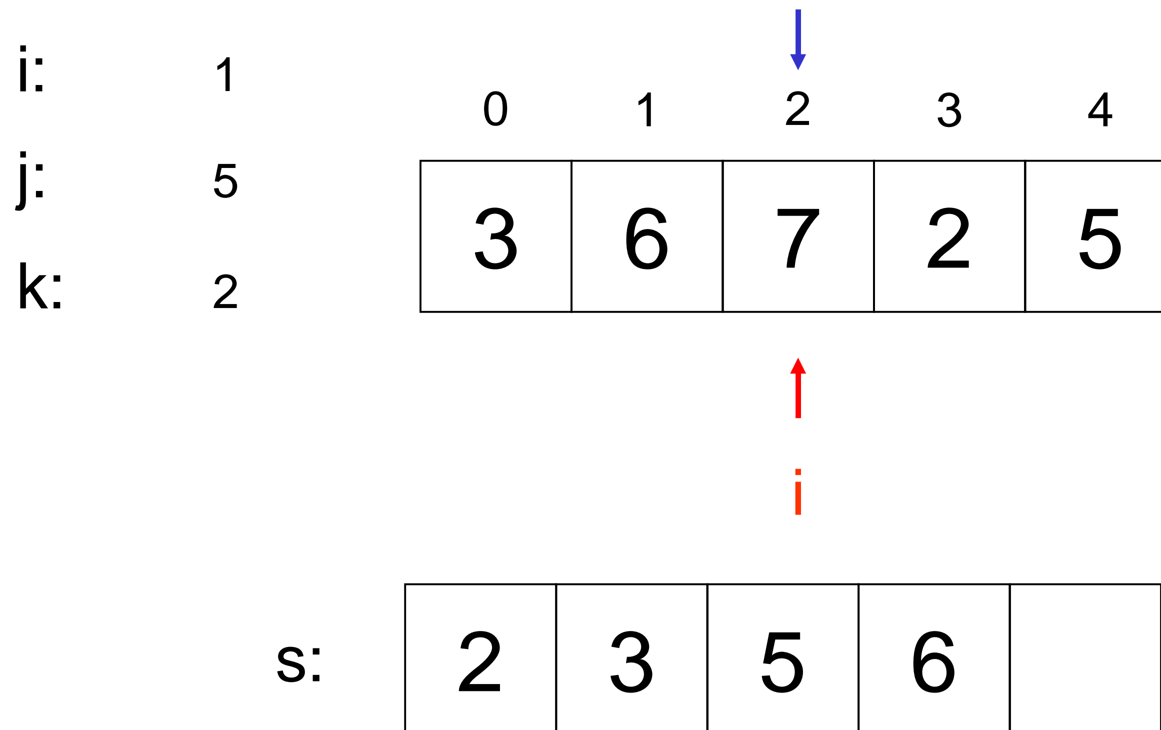
```
// slučuji, s je pomocne pole
  i = l; j = q+1;
  k = 0;
  while(i<=q && j <= r)
  {
      if (pole[i]<pole[j]) s[k++] = pole[i++];
      else s[k++] = pole[j++];
  }
  // kopirovani zbytku poli - probehne jen jeden z cyklu
  while (i<=q) s[k++] = pole[i++];
  while (j<=r) s[k++] = pole[j++];
```



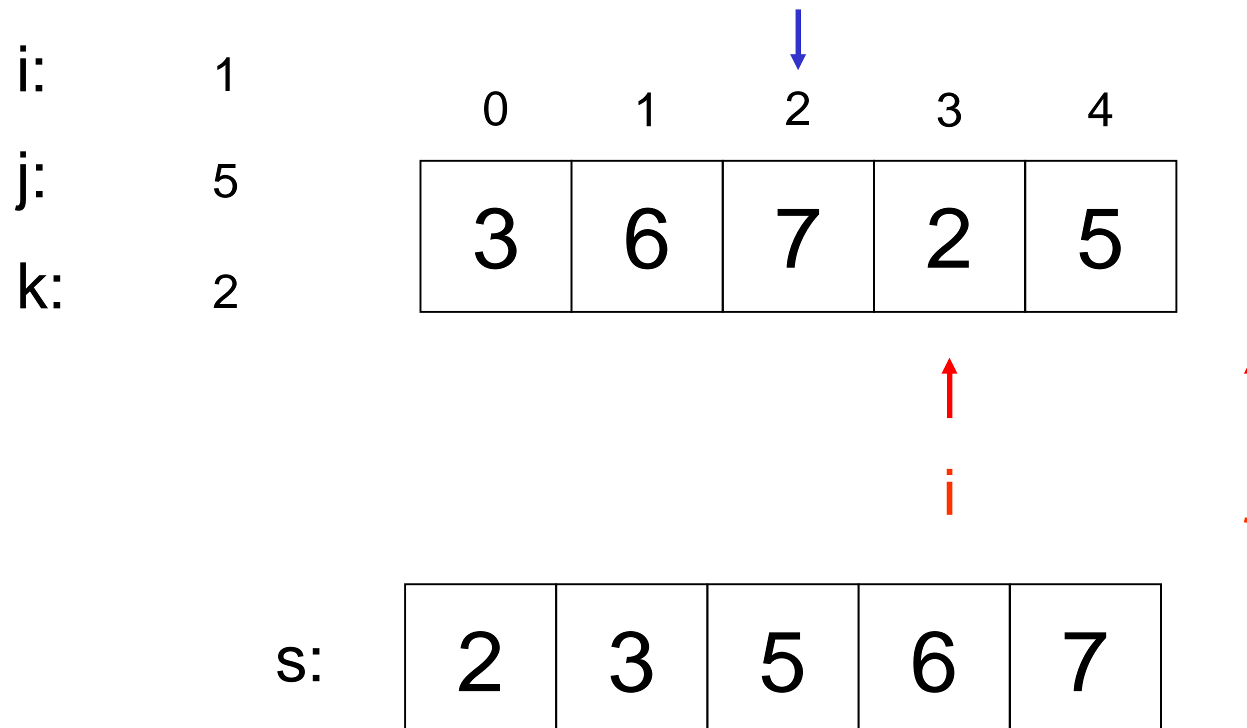
```
// slučuji, s je pomocne pole
  i = l; j = q+1;
  k = 0;
  while(i<=q && j <= r)
  {
      if (pole[i]<pole[j]) s[k++] = pole[i++];
      else s[k++] = pole[j++];
  }
  // kopirovani zbytku poli - probehne jen jeden z cyklu
  while (i<=q) s[k++] = pole[i++];
  while (j<=r) s[k++] = pole[j++];
```



```
// slučuji, s je pomocne pole
   i = l; j = q+1;
   k = 0;
   while(i<=q && j <= r)
   {
        if (pole[i]<pole[j]) s[k++] = pole[i++];
        else s[k++] = pole[j++];
   }
   // kopirovani zbytku poli - probehne jen jeden z cyklu
   while (i<=q) s[k++] = pole[i++];
   while (j<=r) s[k++] = pole[j++];
```



```
// slučuji, s je pomocne pole
  i = l; j = q+1;
  k = 0;
  while(i<=q && j <= r)
  {
      if (pole[i]<pole[j]) s[k++] = pole[i++];
      else s[k++] = pole[j++];
  }
  // kopirovani zbytku poli - probehne jen jeden z cyklu
  while (i<=q) s[k++] = pole[i++];
  while (j<=r) s[k++] = pole[j++];
```



Poznámky ke složitostem

- lineární
  - zdvojnásobení velikosti problému vede k dvojnásobnému času výpočtu
  - zrychlení počítače 2x urychlí řešení problému 2x
- kvadratická
  - zdvojnásobení velikosti problému vede k čtyřnásobnému času výpočtu
  - zrychlení počítače 2x urychlí řešení problému 1,414x

Poznámky ke složitostem

- předpokládejme, že 1 operace trvá 1us, počet operací je dán vztahem v 1. sloupci
- doba výpočtu:

n	10	20	40	80	500	1000
$\log n$	$3,3\ \mu s$	$4,3 \mu s$	$5 \mu s$	$5,8 \mu s$	$9 \mu s$	$10 \mu s$
n	$10 \mu s$	$20 \mu s$	$40 \mu s$	$60 \mu s$	$0,5 ms$	$1 ms$
$n \log n$	$33 \mu s$	$86 \mu s$	$0,2 ms$	$0,35 ms$	$4,5 ms$	$10 ms$
n^2	$0,1 ms$	$0,4 ms$	$1,6\ ms$	$3,6 ms$	$0,25 s$	$1 s$
n^3	$1 ms$	$8 ms$	$64 ms$	$0,2 s$	$125 s$	$17 min$
n^4	$10 ms$	$160 ms$	$2,56 s$	$13 s$	17 h	$11,6$ dní
2^n	$1 ms$	$1 s$	$12,7$dní	3600 let	10^{137} let	10^{287} let
$n!$	$3,6 s$	77100 let	10^{34} let	10^{105} let	10^{1110} let	10^{2554} let

- srovnej: počet atomů ve vesmíru se odhaduje na 10^{80} a stáří vesmíru na 14 10^9 let)
- zdroj: přednášky ZDT, FIT ČVUT, doc. Kolář

Poznámky ke složitostem

- pro malé množství dat, resp. určité uspořádání dat může být asymptoticky pomalejší algoritmus rychlejší
  - pokud aplikujeme bublinkové řazení s testem prohození na seřazenou posloupnost, vykoná se pouze n kroků, ale Quick-Sort vykoná $n.\log_2 n$ vždy
  - pro malý rozměr pole u algoritmu Quick-Sort převáží konstanty – režie související s rekurzivním voláním

Poznámky ke složitostem

- ale vždy existuje určité n_0, od kterého bude asymptoticky rychlejší algoritmus rychlejší bez ohledu na rychlost počítače, jazyk, překladač, …
  - výkon počítače, překladač, …, mění konstanty vztahu, které posouvají bod n_0