

IAL – 8. přednáška



Řazení II.

12. a 13. listopadu 2024

Obsah přednášky

- Řazení polí na principu vkládání
 - Bubble-insert sort
 - Binary-insert sort
- Řazení na principu rozdělávání
 - Quick sort
 - Shell sort
- Řazení na principu slučování
 - Merge sort
 - Sequence-merge sort
 - List-merge sort
 - Tim sort
- Radix sort

Řazení na principu vkládání

- ❑ **Řazení vkládáním** (*insert*) se podobá mechanismu, kterým hráč karet bere postupně karty ze stolu a vkládá je do uspořádaného vějíře v ruce.
- ❑ Pole dělíme na dvě části:
 - Levou – seřazenou a pravou – neseřazenou.
 - Levou část tvoří na začátku první prvek.
- ❑ Algoritmus řazení má následující strukturu:

for $i \leftarrow (1, \text{MAX}-1)$:

```
// najdi v levé části index k, kam se má zařadit prvek A[i]  
// posuň část pole od k do i-1 o jednu pozici doprava  
// vlož na A[k] hodnotu zařazovaného prvku
```

Bubble-insert sort

- Metoda bublinového vkládání
- Kombinuje vyhledání místa pro vkládání i posun segmentu pole do jednoho cyklu:
 - Postupným porovnáváním a výměnou dvojic prvků.
- *Pozn.:* Tato metoda řazení odpovídá dříve zmíněné metodě *Shuttle sort*, která byla odvozena od *Bubble sortu* (tedy od metody založené na principu výběru). Postup řazení ale odpovídá spíše principu vkládání.

Bubble-insert sort

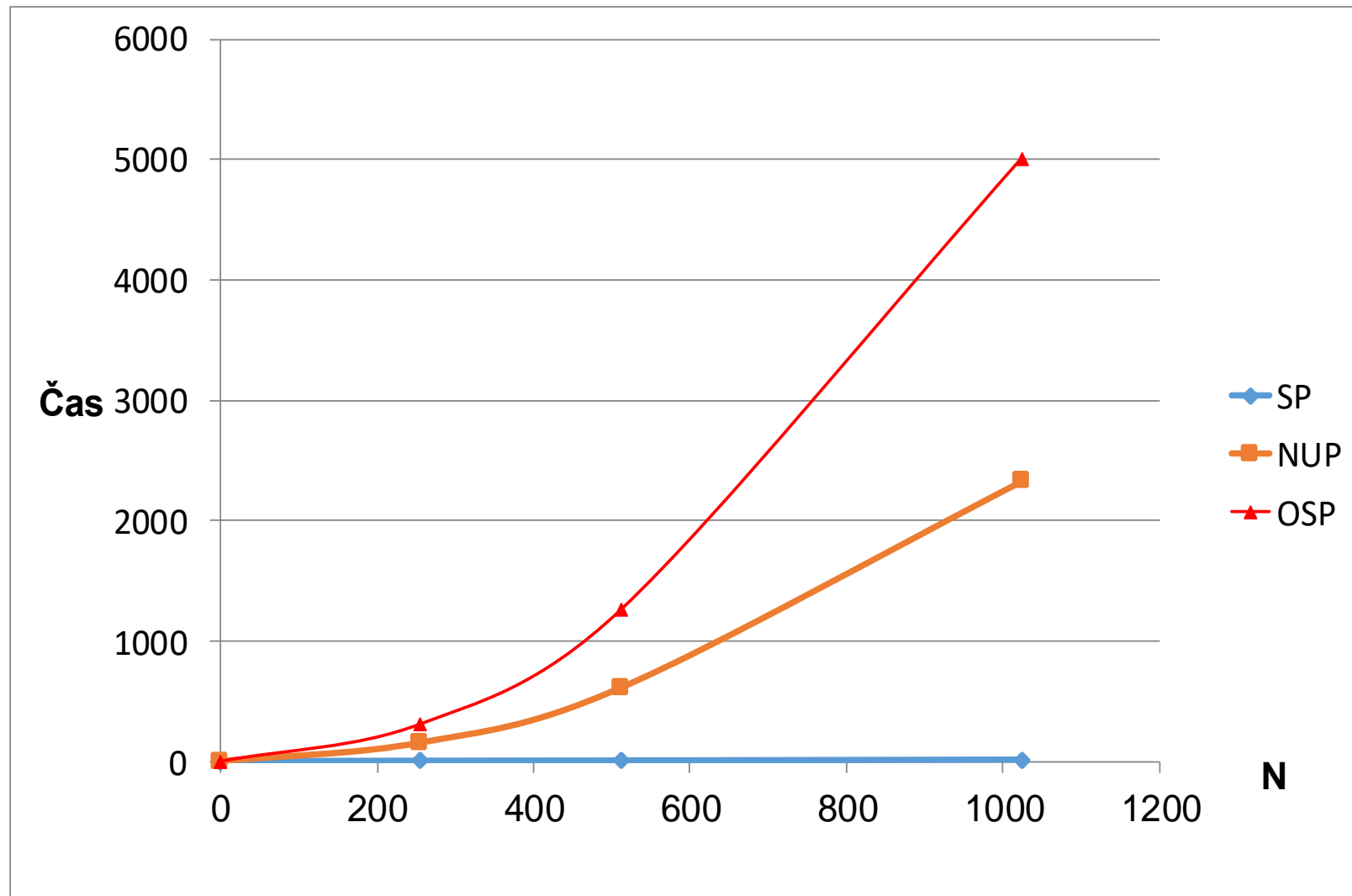
```
procedure BubbleInsertSort (TArray A)
  for i ← (1, MAX-1):
    tmp ← A[i]
    j ← i-1
    while j ≥ 0 and tmp < A[j]: // zkrat. vyhodnocování!
                                // najdi a posuň prvek
      A[j+1] ← A[j]
      j ← j-1
    A[j+1] ← tmp               // konečné vložení na místo
```

Bubble-insert sort – zhodnocení

- ❑ Metoda je **stabilní** – je vhodná pro vícenásobné řazení podle více klíčů.
- ❑ Chová se **přirozeně** a pracuje **in situ**.
- ❑ Má **kvadratickou** časovou složitost.
- ❑ Experimentálně byly naměřeny tyto hodnoty:

N	256	512	1 024
SP	4	6	14
NUP	156	614	2 330
OSP	312	1 262	5 008

Bubble-insert sort – naměřené výsledky graficky



Binary-insert sort

- ❑ Vkládání s binárním vyhledáváním.
- ❑ Pro vložení prvku vyhledáváme místo v **seřazené posloupnosti** – lze využít **binární vyhledávání**.
- ❑ V případě shodných klíčů musí metoda nalézt místo **za nejpravějším ze shodných klíčů** – varianta Dijkstrový metody binárního vyhledávání.

Binary-insert sort

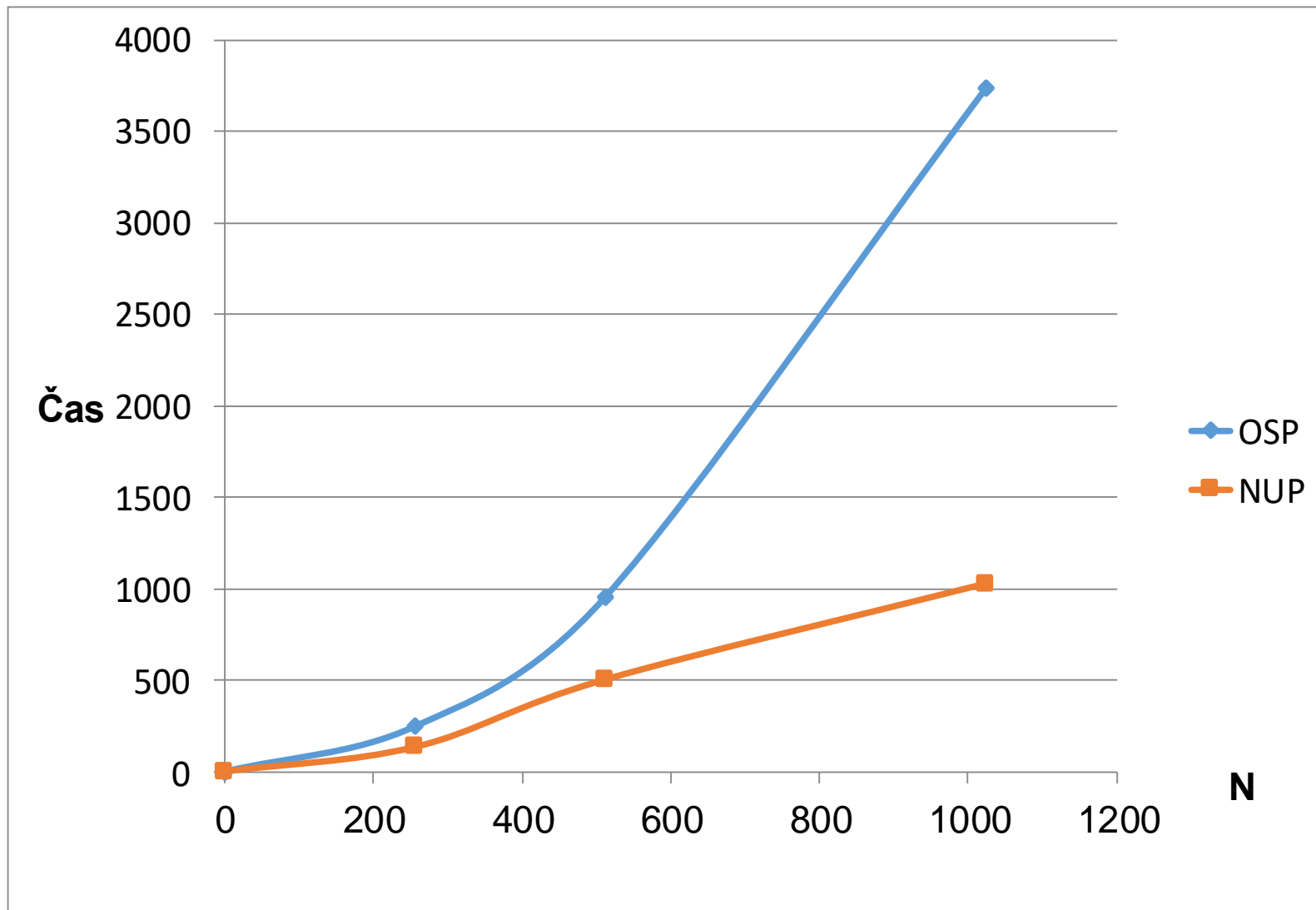
```
procedure BinaryInsertSort (TArray A)
  for i  $\leftarrow$  (1, MAX-1):
    tmp  $\leftarrow$  A[i]
    left  $\leftarrow$  0                // hranice již seřazené části
    right  $\leftarrow$  i-1
    while left  $\leq$  right:        // binární vyhledání
      m  $\leftarrow$  (left+right) div 2
      if tmp < A[m]:
        right  $\leftarrow$  m-1
      else:
        left  $\leftarrow$  m+1        // ale skončíme těsně za
    for j  $\leftarrow$  (i-1, left)-1:
      A[j+1]  $\leftarrow$  A[j]        // posun segmentu pole doprava
    A[left]  $\leftarrow$  tmp          // prvek z pozice i na své místo
```

Binary-insert sort zhodnocení

- ❑ Metoda je **stabilní**.
- ❑ Chová se **přirozeně** a pracuje **in situ**.
- ❑ Má **kvadratickou** časovou složitost.
- ❑ Binární vyhledávání vkládací princip výrazně nevylepšilo:

N	256	512	1 024
NUP	134	502	1 024
OSP	248	956	3 736

Binary-insert sort – naměřené výsledky graficky



Řazení rozdělčováním – Quick sort

□ **Princip:** Představme si algoritmus, který umí (rychle) rozdělit množinu položek na **dvě podmnožiny**:

- jedna by obsahovala všechny **prvky s klíčem menším** (nebo rovným) **jisté hodnotě**
- druhá by obsahovala všechny **prvky s klíčem větším** (nebo rovným) **téže hodnotě**

13	12	6	10	2	5	15	9
klíče 2 – 9				klíče 10 – 15			
2 – 5		6 – 9		10 – 12		13 – 15	
2	5	6	9	10	12	13	15

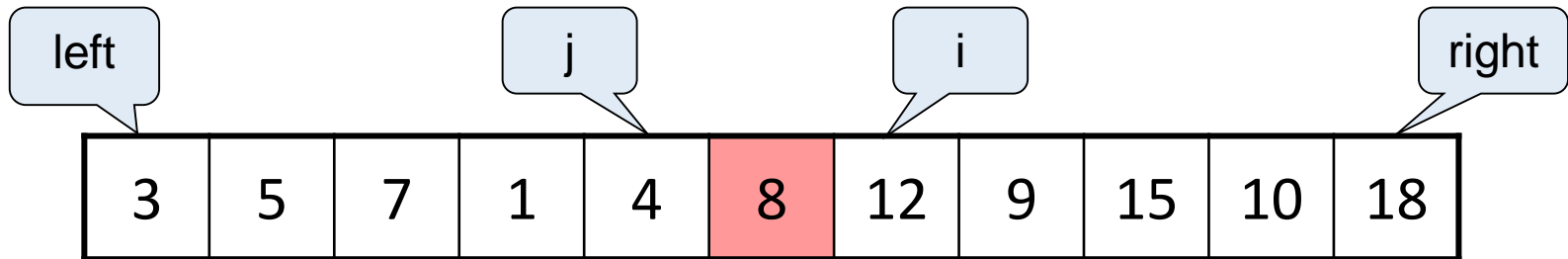
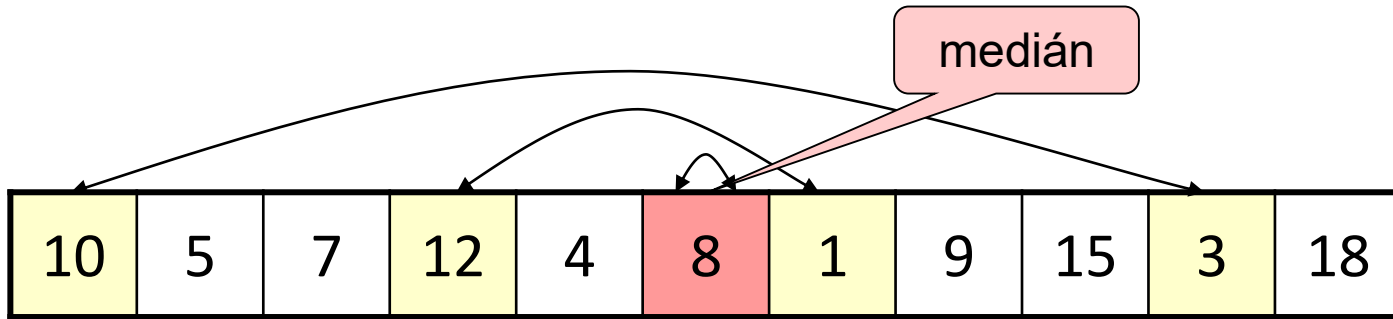
- Postupnou **aplikací** tohoto algoritmu **na každou získanou podmnožinu**, která obsahuje více než 1 prvek, lze získat **seřazenou množinu**.
- Mechanismu rozdělení říkáme **partition**.

Mechanismus rozdělení – partition

- **Medián** – prvek z množiny hodnot, pro který platí:
 - Polovina prvků je menší než medián.
 - Polovina prvků je větší než medián.
- Při znalosti mediánu je snadné implementovat proceduru **partition**, která rozdělí pole na dvě části:
 - Procházíme pole současně zleva (index **i**) a zprava (index **j**).
 - Zleva hledáme prvek větší nebo roven mediánu, zprava prvek menší nebo roven mediánu.
 - Nalezené prvky vyměníme a hledáme další prvky pro výměnu.
 - Proces ukončíme až se dvojice indexů překříží.
 - **i** a **j** jsou návratové hodnoty funkce, definující intervaly **left..j** (prvky menší nebo rovny mediánu) a **i..right** (prvky větší nebo rovny mediánu).
- **Problém:** nalezení mediánu.

Mechanismus rozdělení – partition

`(int, int) function partition (TArray A, int left, int right)`



Quick sort – rekurzivní zápis

```
procedure QuickSort (TArray A, int left, int right)
// Při volání má left hodnotu 0 a right hodnotu MAX-1

i, j ← partition(A, left, right)
if left < j:
    QuickSort(A, left, j)           // Rekurze doleva
if i < right:
    QuickSort(A, i, right)         // Rekurze doprava
```

Mechanismus partition I.

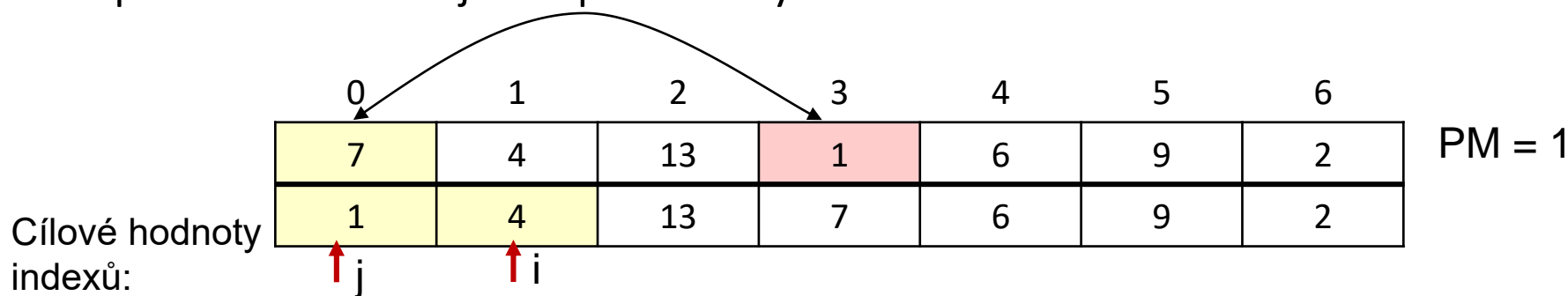
- Protože hledání mediánu je časově náročné, použijeme tzv. **pseudomedián**:
 - Libovolná hodnota z daného souboru čísel
 - Vhodnou hodnotou je číslo ze středu intervalu: **$(\text{left} + \text{right}) \div 2$**
 - Experimentálně je prokázáno, že toto číslo splní svou roli velmi podobně jako medián.
- Abychom nemuseli při hledání hodnot pro výměnu kontrolovat hranice pole, používáme **pseudomedián jako zářezku**:
 - Hledáme hodnoty menší/větší **nebo rovny** pseudomediánu.
 - Takto nemusíme kontrolovat hranice pole např. v případě, že pole bude naplněno stejnými čísly.
- **Pozn.:** Autorem uvedených postupů je C. A. R. Hoare, významná osobnost v oboru teorie a tvorby programů. Uvedené postupy jsou klíčem k rychlosti Quick sortu.

Mechanismus partition I.

```
(int, int) function partition (TArray A, int left, int right)
    i ← left                                // inicializace i
    j ← right                                // inicializace j
    PM ← A[(i+j) div 2]                     // ustavení pseudomediánu
    do
        while A[i] < PM:
            i ← i+1                          // první i zleva, pro A[i] ≥ PM
        while A[j] > PM:
            j ← j-1                          // první j zprava pro A[j] ≤ PM
        if i ≤ j:
            A[i] ↔ A[j]                      // výměna nalezených prvků
            i ← i+1
            j ← j-1
    while i ≤ j                             // konec, když se indexy i a j překříží
    return (i, j)
```

Mechanismus partition I.

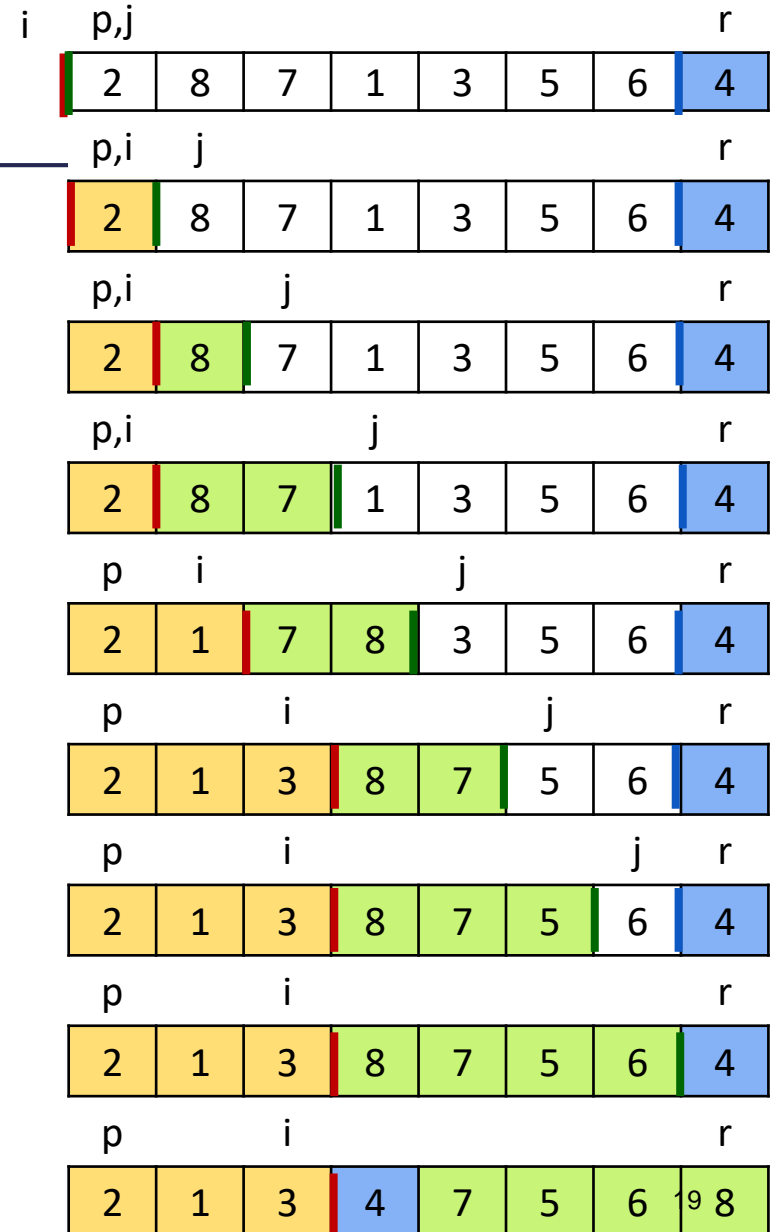
- **Pozn. 1:** Na konci mechanismu `partition` je vždy $i > j$. Obvykle jsou to dvě sousední hodnoty, ale někdy mohou mít hodnotu ob jeden index.
- **Pozn. 2:** Bude-li jako pseudomedián vybráno minimum nebo maximum, pole se rozdělí na jeden prvek a zbytek.



- **Pozn. 3:** Existují různé varianty uvedeného mechanismu `partition`. Liší se především v následujících aspektech:
 - Volbou pseudomediánu – první /poslední /prostřední / libovolný prvek.
 - Počtem hodnot, které mechanismus vrací – 1 nebo 2 hodnoty.
 - Využitím pseudomediánu při výměnách prvků - pseudomedián může být vyměňován s vhodnými prvky průběžně, nebo může být vyměněn s vhodným prvkem až na konci daného rozdělení prvků.

Mechanismus partition II.

- Jako pseudomedian (pivot) je volen nejpravější prvek
- Pole procházíme postupně zleva doprava (index j) a ve zpracované části udržujeme vlevo prvky menší nebo rovny pivotu (do indexu i) a vpravo prvky větší než pivot.
- Vždy když narazíme na prvek menší než pivot, vyměníme ho s prvním prvkem, který je větší než pivot
- Nakonec je pivot vyměněn s prvním prvkem části pole s prvky většími než pivot.
- Partition vrací index nové pozice pivotu, rekurzivní volání pokračují vlevo a vpravo od tohoto prvku
- Méně efektivní než předchozí mechanismus.



Mechanismus partition II.

```
int function partitionII (TArray A, int left, int right)
    i ← left - 1
    PM ← A[right]                // ustavení pseudomediánu
    for j ← (left, right-1):      // projdi pole zleva
        if A[j] ≤ PM:             // menší musí do levé části
            i ← i+1               // za poslední prvek
            A[i] ↔ A[j]           // výměna nalezených prvků
    A[i+1] ↔ A[right]
    return i+1
```

```
procedure QuickSortII (TArray A, int left, int right)
    if left < right:
        q ← partitionII (A, left, right)
        QuickSortII(A, left, q-1)    // rekurze doleva
        QuickSortII(A, q+1, right)   // rekurze doprava
```

Quick sort – nerekurzivní zápis

- **Nerekurzivně** můžeme Quick sort implementovat takto:
 - Využijeme zásobník.
 - Dva segmenty určené mechanismem *partition* zpracujeme tak, že jeden podrobíme dalšímu dělení a hranice druhého uložíme do zásobníku.
- Algoritmus sestává **ze dvou cyklů**:
 - Vnitřní cyklus provádí opakované dělení segmentu pole a uchovávání hraničních bodů druhého segmentu v zásobníku. Cyklus se ukončí, až *není co dělit*, a opakuje se vnější cyklus.
 - Vnější cyklus vyzvedne ze zásobníku hraniční body dalšího segmentu a vstoupí do vnitřního cyklu. Vnější cyklus se ukončí, když je zásobník prázdný a není žádný další segment k dělení.

Quick sort – nerekurzivní zápis

```
procedure NonRecQuicksort (TArray A, int left, int right)
  InitStack(s)
  Push(s, left)           // uložení hranic celého pole
  Push(s, right)
  while not IsEmpty(s):   // vnější cyklus
    right ← Top(s)
    Pop(s)                // čtení v opačném pořadí
    left ← Top(s)
    Pop(s)
    while left < right:   // dokud je co dělit
      i, j ← Partition(A, left, right)
      Push(s, i)           // interval pravé části do zás.
      Push(s, right)
      right ← j            // pravý index pro další cyklus
```

Nerekurzivní Quick sort – velikost zásobníku

- **Kapacita zásobníku** musí být dostatečná pro nejhorší případ:
 - Pokud se **vždy dělí levý segment** a pravý se uchovává, nejhorší je případ, kdy se vždy segment rozdělí na jeden prvek a zbytek. Pak se musí uchovat **$n-1$ dvojic indexů**.
 - *Alternativa:* algoritmus bude dělit **vždy menší segment** a hranice většího uchová v zásobníku. Nejhorší případ nastane, když se interval vždy rozdělí na dva stejně velké segmenty. V tom případě je třeba zásobník dimenzovat na kapacitu **$\log_2 n$ dvojic indexů**.
 - *Příklad:* pokud by se řadilo pole obsahující 1000 prvků, pak v případě prvního algoritmu je zapotřebí zásobník o kapacitě 999 dvojic. Ve druhém případě, kdy se menší segment dělí a větší uchovává, stačí zásobník o kapacitě $\log_2 1000$, tj. 10 dvojic.

Quick sort – zhodnocení

- ❑ Quick sort patří **mezi nejrychlejší** algoritmy pro řazení polí.
- ❑ Quick sort je **nestabilní** a **nepracuje přirozeně**.
- ❑ Časová složitost je **linearitmická** pro vhodně zvolený pseudomedián.
- ❑ V nejhorším případě – při špatné volbě pseudomediánu (vždy minimum nebo maximum), je časová složitost **kvadratická**.
 - zlepšení volby pseudomediánu – **výběr mediánu ze tří náhodně vybraných hodnot**

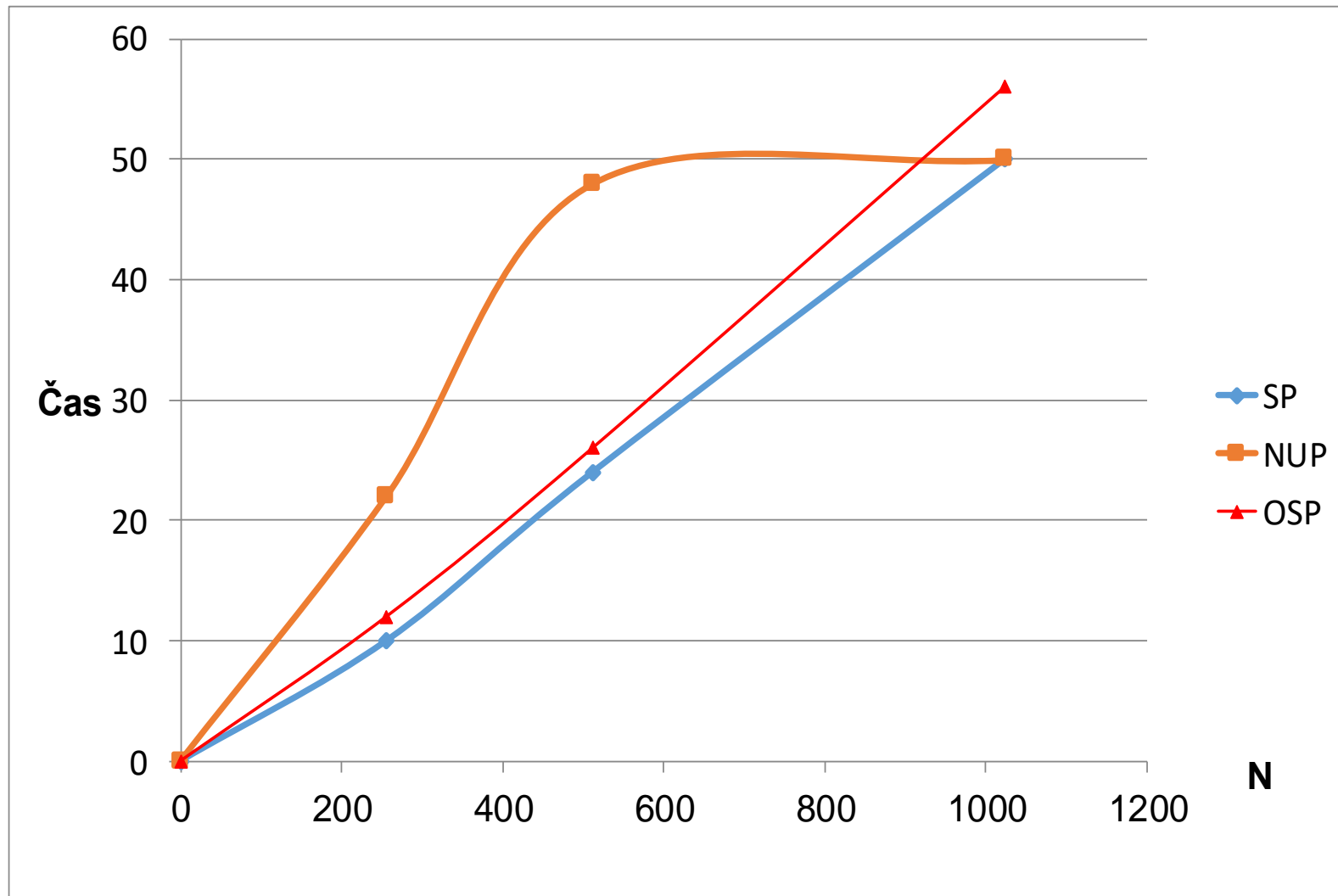
Quick sort – naměřené výsledky

- Tabulka experimentálně naměřených hodnot pro Quick sort:

N	256	512	1 024
SP	10	24	50
OSP	22	48	50
ISP	12	26	56

- *K procvičení:* Upravte algoritmus nerekurzivního zápisu Quick sortu pro variantu menšího zásobníku.

Quick Sort – naměřené výsledky graficky



Shell sort

- Metoda řazení **se snižujícím se přírůstkem**
- Metoda používá **opakované průchody polem**, ve kterých řadí vždy jen určitou **podposloupnost** původní sekvence:
 - Původní sekvence je rozdělena na několik podposloupností, do kterých jsou vybrány **prvky vzdálené od sebe o určitý krok**.
 - Prvky v každé podsekvenci jsou uspořádány jedním bublinovým průchodem (*Bubble-insert sort*).
 - Po seřazení všech podposloupností se **krok zmenší** a **opakuje se řazení** pro nové podsekvence.
 - **V poslední etapě** řazení je krok roven jedné, všechny prvky jsou v jedné podposloupnosti, a řazení je dokončeno posledním bublinovým průchodem.
- **Pozn.:** Rychlé řadicí algoritmy se vyznačují tím, že se prvky blíží ke svému správnému místu většími kroky.

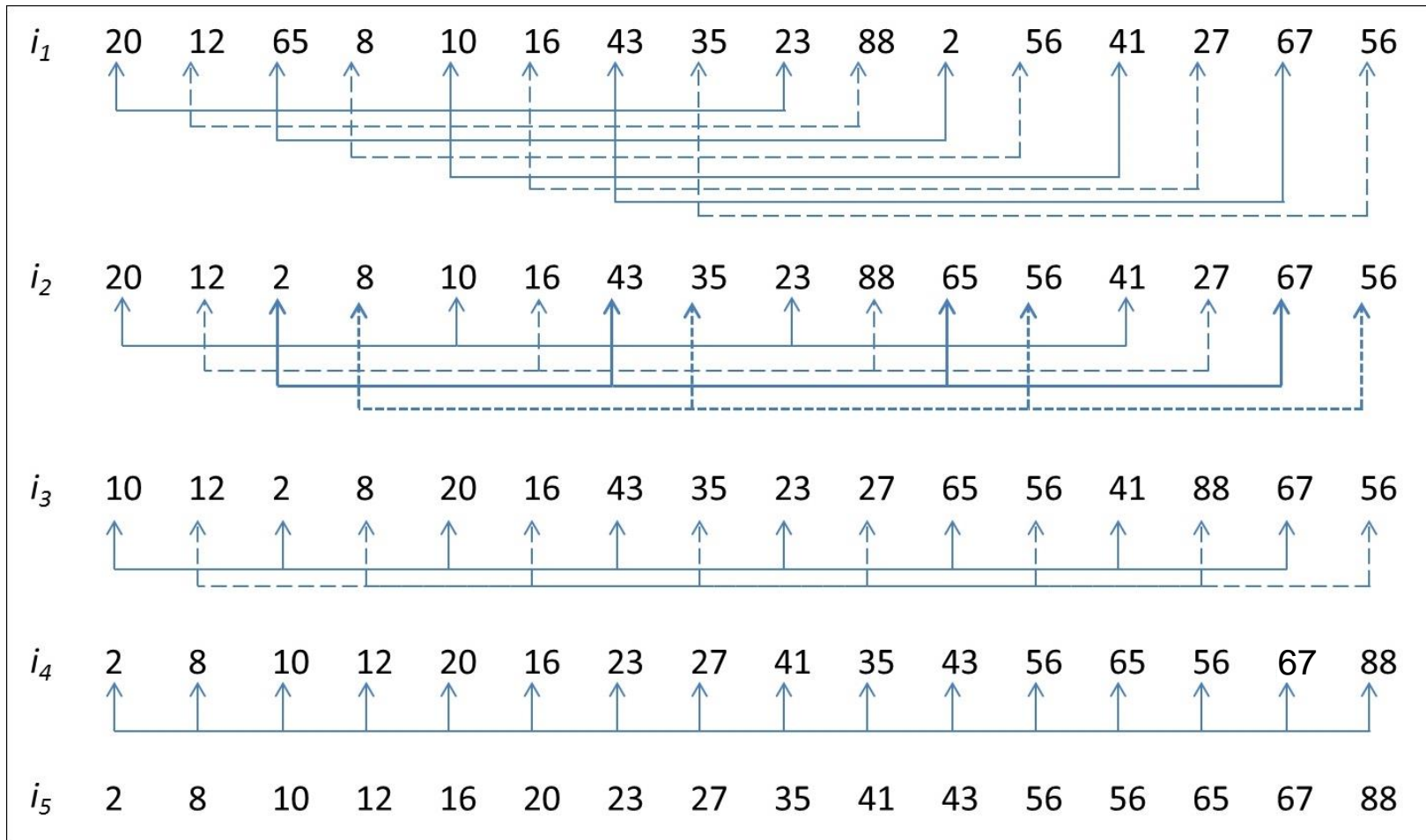
Shell sort

- Následující sekvence (reprezentovaná indexy) může být rozčleněna na čtyři podsekvence čísel vzdálených o krok 4:

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
1				5				9				13				17			
	2				6				10				14				18		
		3				7				11				15				19	
			4				8				12				16				20

- Jednotlivé podsekvence jsou seřazeny bublinovým průchodem.
- Ve druhé etapě se krok sníží na dvě a obě sekvence se zpracují jedním bublinovým průchodem.
- V poslední etapě se na celou sekvenci aplikuje bublinový průchod s krokem jedna. Tím je řazení ukončeno.

Shell sort – příklad



Shell sort – velikost kroku

- Teoretické analýzy **nenašly nejvhodnější řadu** snižujících se kroků.
- **Základem** je verze algoritmu, v níž je první krok $(n \div 2)$ a v první etapě dochází k výměně $(n \div 2)$ dvojic, pokud nejsou uspořádány v žádoucím směru.
- V další etapě se krok vždy pólí a n -tice zpracovávané bublinovým průchodem se zdvojnásobují.
- Poslední etapou je průchod celým polem s krokem jedna.

Shell sort

```
procedure ShellSort (TArray A)
  step ← MAX div 2           // první krok - polovina délky pole
  while step > 0:
    for i ← (step, MAX-1):    // cykly pro paralelní n-tice
      j ← i-step
      while (j ≥ 0) and (A[j] > A[j+step]): //bubl.ins.
        A[j] ↔ A[j+step]
        j ← j-step           // snížení indexu o krok
      step ← step div 2      // půlení kroku
```

Pozn.: Bublinový průchod provádí zavedení prvku z indexu $[j+step]$ na jeho místo v dané podposloupnosti (čili porovnává jeho hodnotu s hodnotou prvků na indexech snížených o $step$). Se zaváděním končí, je-li hodnota prvku na sníženém indexu menší než hodnota zaváděného prvku.

Shell sort – zhodnocení

- ❑ Shell sort je **nestabilní** metoda.
- ❑ Pracuje **in situ**.
- ❑ Časová složitost závisí na zvolené řadě snižujících se kroků:
 - Pro uvedenou verzi ($n/2, n/4, \dots, 1$) je v nejhorším případě časová složitost **n^2** .
 - Existují řady, pro které je časová složitost **$n^{3/2}$** nebo **$n \cdot \log^2 n$** .
- ❑ V uvedené modifikaci pracuje rychleji než Heap sort ale pomaleji než Quick sort.
- ❑ Nepotřebuje ani rekurzi ani zásobník.

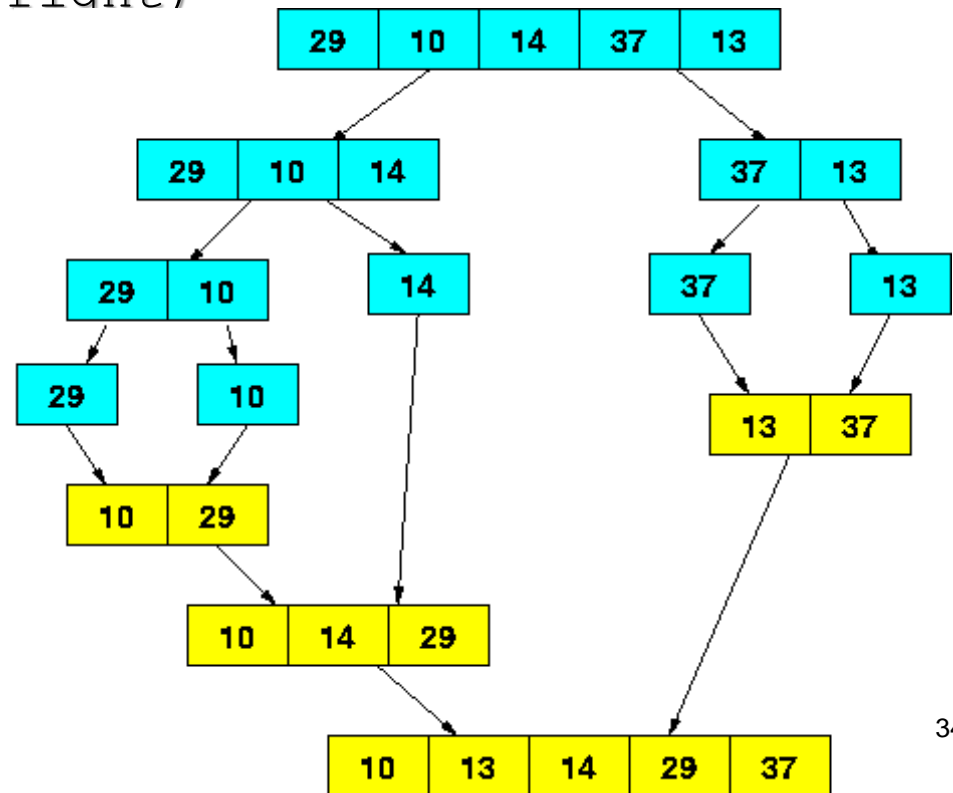
Řazení setřídováním – Merge sort

- Merge sort je založen na **principu slučování** (setřídování), tedy na principu komplementárním k rozdělování. **Princip:**
 - Pole rozdělujeme do tzv. běhů – souvislých **úseků, které už jsou setříděny (seřazeny)**.
 - Na začátku budou všechny běhy jednoprvkové.
 - Poté budeme dohromady **slévat vždy dva sousední** běhy do jediného setříděného běhu o délce dané součtem počtu prvků slévaných běhů, který bude ležet na místě obou vstupních běhů.
 - Po poslední iteraci bude posloupnost sestávat z jediného běhu, a bude tudíž setříděná (seřazená).
- Metoda **vyžaduje pomocné (nebo pomocná) pole**, pro uložení setříděné posloupnosti (nebo uložení vstupních posloupností).
- **Rekurzivní varianta** – metoda postupně volá sebe sama pro levou a pravou polovinu zadané části pole a při návratu z rekurze slévá již setříděné posloupnosti.

```

procedure MergeSort (TArray A, int left, int right)
// Při volání má left hodnotu 0 a right hodnotu MAX-1
if (left < right):
    q ← (left + right) div 2
    MergeSort(A, left, q)
    MergeSort(A, q+1, right)
    Merge(A, left, q, right)

```



Operace Merge

- Operace Merge provádí slévání – použije dvě pomocná pole (typu `TArray`), do kterých přesune vstupní posloupnosti. Výslednou setříděnou posloupnost vkládá zpět do pole A.

```

procedure Merge (TArray A, int left, int mid, int right)
    left_count ← mid - left + 1 // počet prvků levé posloupnosti
    right_count ← right - mid    // počet prvků pravé posloupnosti
    for i ← (0, left_count-1):    // levá posloupnost do pom. pole
        L[i] ← A[left+i]
    for j ← (0, right_count-1):  // přesun pravé posloupnosti
        R[j] ← A[mid+1+j]        // začíná na indexu mid+1
    L[left_count] ← MaxInt       // ustavení zářátek
    R[right_count] ← MaxInt
    i ← 0
    j ← 0
    for k ← (left, right):       // slévání a vkládání do pole A
        if L[i] ≤ R[j]:          // vyber menšího
            A[k] ← L[i]          // menší byl v levé posloupnosti
            i ← i+1
        else:
            A[k] ← R[j]          // menší byl vpravo
            j ← j+1

```

Merge sort – zhodnocení

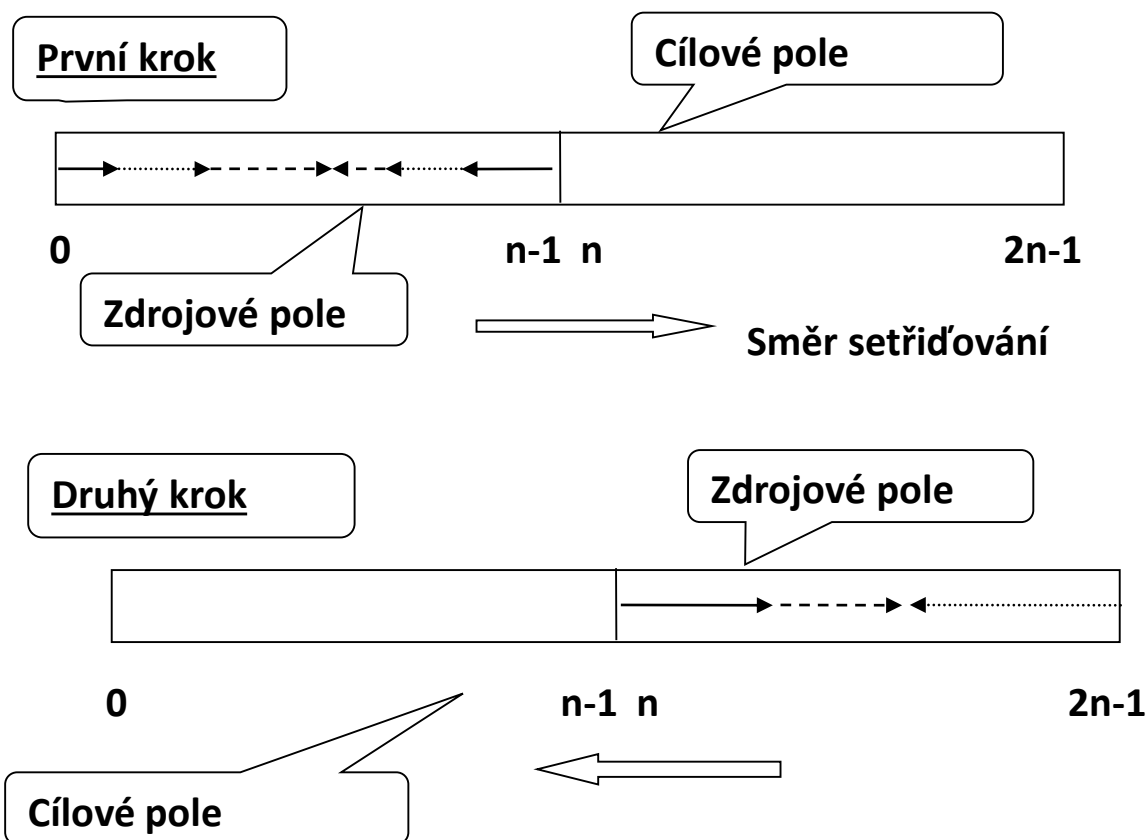
- ❑ Jedná se o **stabilní** metodu.
- ❑ Potřebuje pomocné pole o stejné velikosti jako je zdrojové pole – tzn. **nepracuje in situ**.
- ❑ Časová složitost je **linearitymická**.

Sequence-merge sort

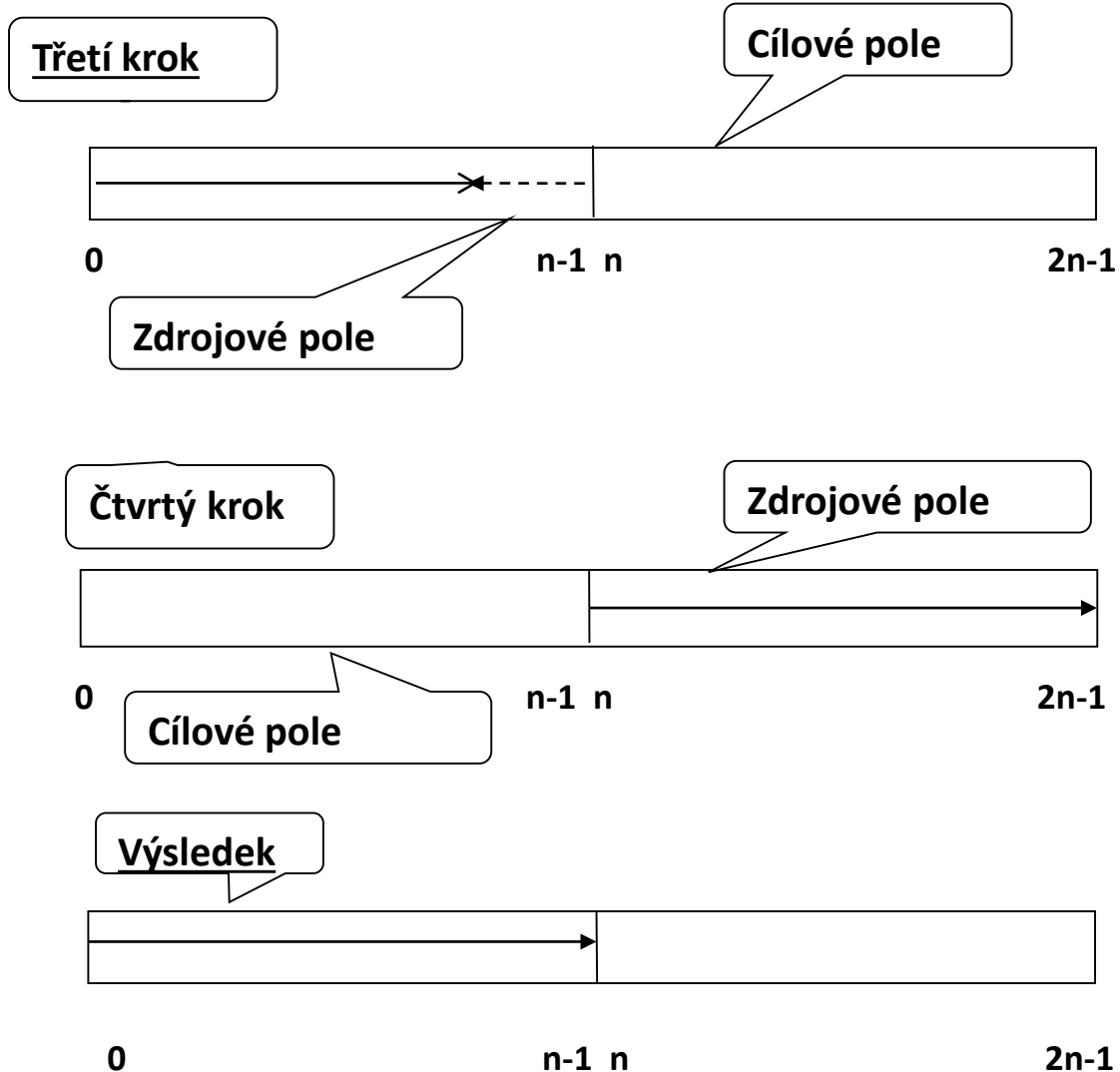
- ❑ Řazení **setřídováním posloupností** – sekvenční metoda využívající přímý přístup k prvkům pole.
- ❑ Postupuje polem zleva a současně zprava a setřídí dvě **proti sobě** postupující neklesající posloupnosti. Výsledek se ukládá do cílového pole.
- ❑ Počet vzniklých posloupností se počítá v počítadle.
- ❑ **Algoritmus končí, vznikne-li jen jedna cílová posloupnost.**

Sequence-merge sort – schéma

- Šipky představují neklesající posloupnosti



Sequence-merge sort – schéma



Sequence-merge sort – zhodnocení

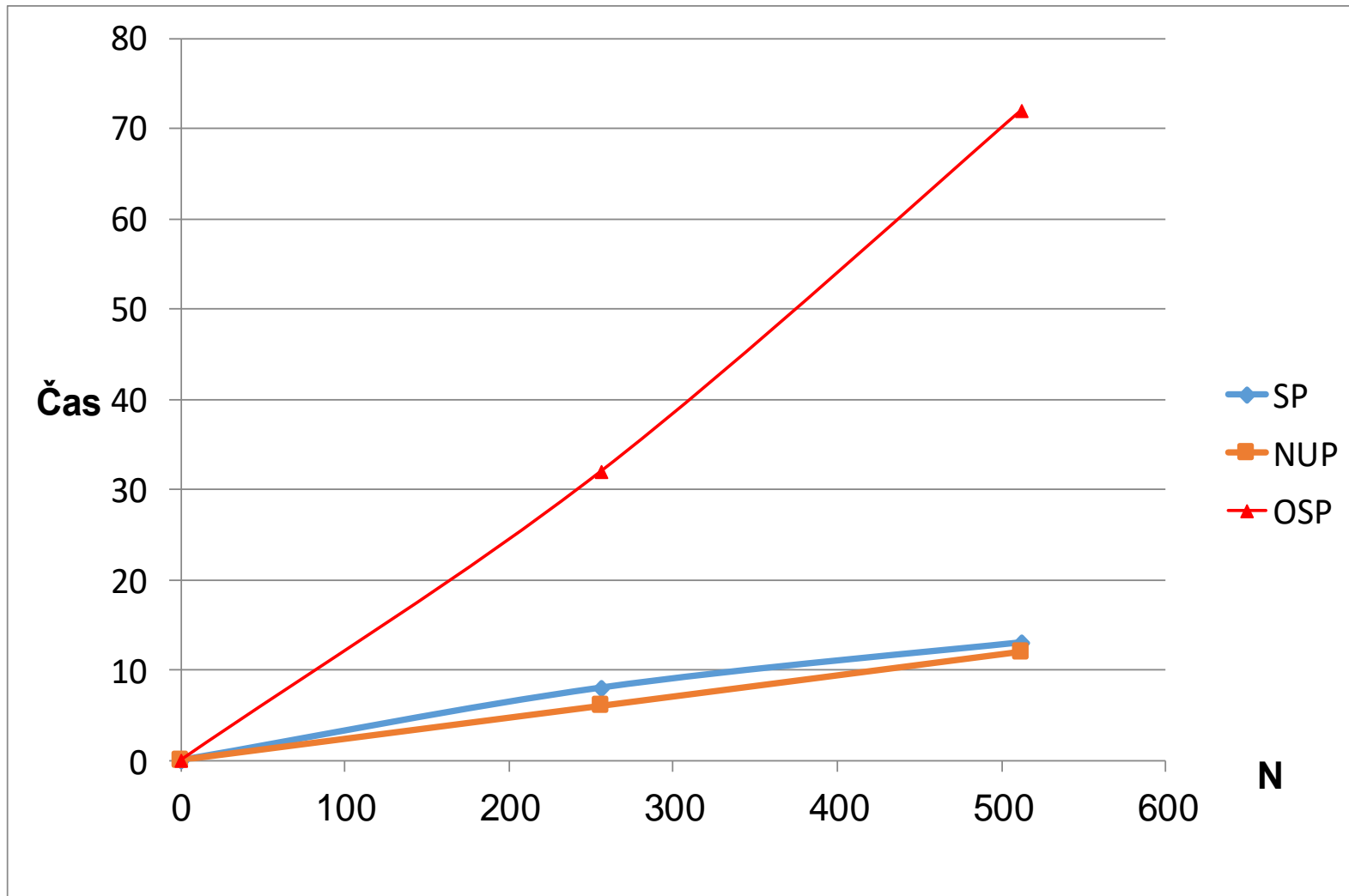
- Významným rysem algoritmu je jeho **houpačkový mechanismus**:
 - automaticky střídá pozici zdrojového a cílového pole i krok postupující proti sobě orientovanými slučovanými neklesajícími posloupnostmi.
- Metoda Sequence-merge sort je **nestabilní**.
- **Nechová se přirozeně** a nepracuje **in situ**.

Sequence-merge sort – zhodnocení

- Asymptotická časová složitost je **linearitnická**.
- Algoritmus je velmi rychlý.
 - Z hlediska konstrukce programu nepatří k jednoduchým algoritmům.
 - Z hlediska programovacích technik patří k velmi zajímavým algoritmům.
- Experimentálně naměřené hodnoty jsou uvedeny v následující tabulce:


N	256	512
SP	8	13
NUP	6	12
ISP	32	72

Sequence-merge sort – naměřené výsledky graficky



List-merge sort


- ❑ Řazení polí **setřídováním seznamů** – pracuje na principu **slučování** metodou **bez přesunů položek**.
- ❑ K základnímu poli je nezbytné vytvořit stejně velké pomocné pole **Ptr** indexových ukazatelů, které **zřetězí neklesající posloupnosti**.
- ❑ Jádrem algoritmu je **setřídění dvou seznamů** zřetězených v pomocném poli indexovými ukazateli.



Ind	0	1	2	3	4	5	6	7	8	9	10	11	12	13
A	2	5	9	3	7	10	16	5	8	6	3	11	18	20
Ptr	1	2	-1	4	5	6	-1	8	-1	-1	11	12	13	-1


List-merge sort

□ Stav po inicializaci: Seznam začátků: 0→3→7→9→10



Ind	0	1	2	3	4	5	6	7	8	9	10	11	12	13
A	2	5	9	3	7	10	16	5	8	6	3	11	18	20
Ptr	1	2	-1	4	5	6	-1	8	-1	-1	11	12	13	-1

□ Stav po sloučení prvních dvou neklesajících posloupností:



Ind	0	1	2	3	4	5	6	7	8	9	10	11	12	13
A	2	5	9	3	7	10	16	5	8	6	3	11	18	20
Ptr	3	4	5	1	2	6	-1	8	-1	-1	11	12	13	-1

Seznam začátků: 7→9→10→0

List-merge sort – princip

□ První krok:

- zřetězení neklesajících posloupností do seznamu a vložení jejich začátků do dvojsměrného seznamu začátků.

□ Následující **cyklus**:

- V každé iteraci se vyzvednou ze seznamu začátky **dvou** zřetězených neklesajících posloupností.
- **Setříděním** těchto posloupností vznikne jedna zřetězená neklesající posloupnost, jejíž začátek se vloží na konec seznamu.
- Cyklus se **ukončí**, je-li v seznamu již jen začátek jedné neklesající zřetězené posloupnosti.

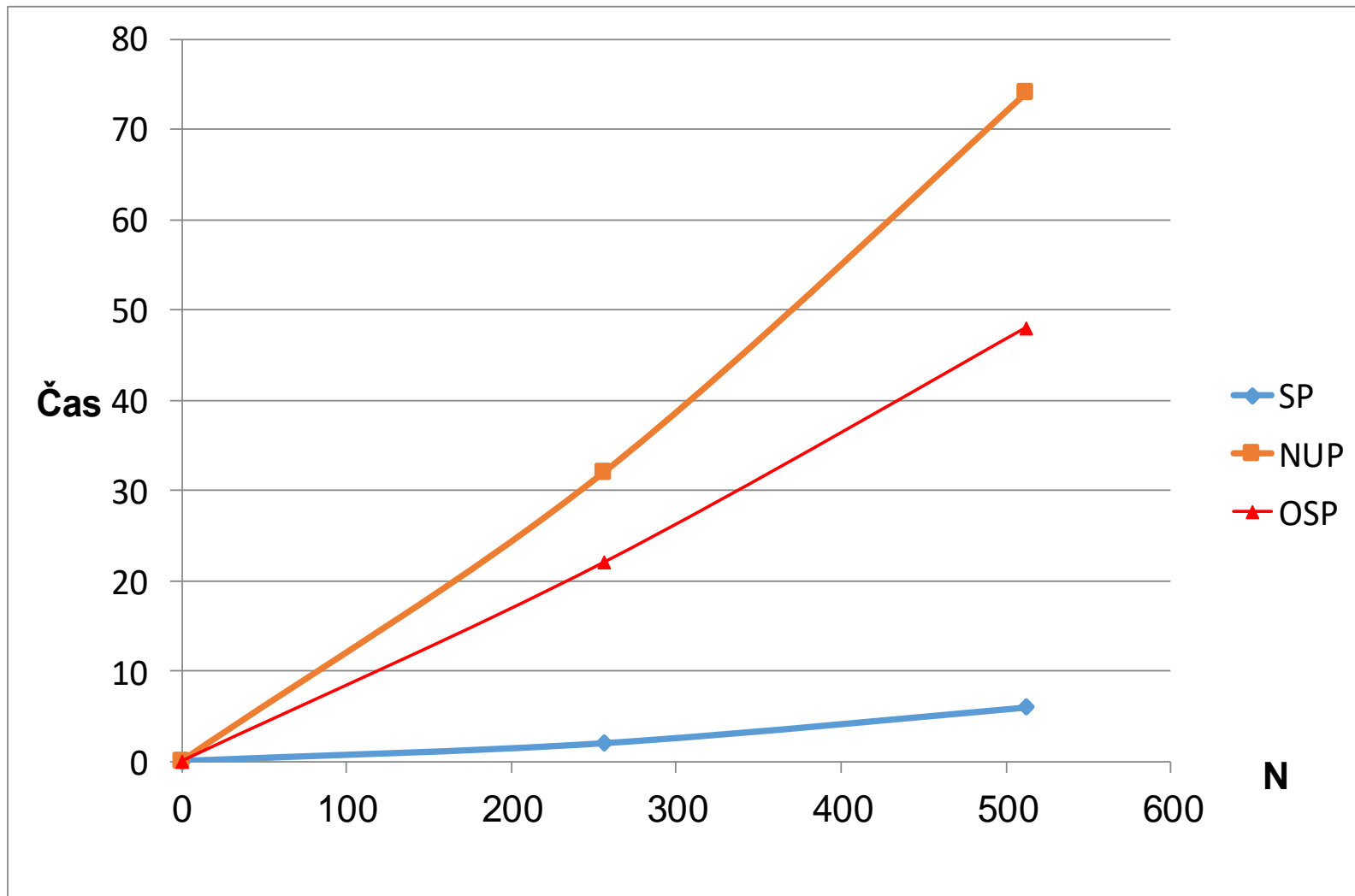
- **Pozn.:** výsledek se může do podoby výstupního seřazeného pole zpracovat vhodným průchodem skrz indexové ukazatele.

List-merge sort – zhodnocení

- List-merge sort je algoritmus pracující **bez přesunu položek**.
- Je potenciálně **stabilní**.
 - Stabilita se zajistí např. tím, že se začátky vkládají do dvojsměrného seznamu (na pozici vyjmutých začátků) a při setřídování se u shodných prvků musí do výstupní posloupnosti vložit prvek první posloupnosti.
- Experimentálně byly naměřeny hodnoty uvedené v následující tabulce.

N	256	512
SP	2	6
NUP	32	74
OSP	22	48

List-merge sort – naměřené výsledky graficky



Tim sort

- ❑ Kombinuje **Merge sort** a **Insert sort**.
- ❑ **Merge sort** je použit na **setřídování neklesajících posloupností**.
- ❑ Pokud jsou neklesající posloupnosti příliš **krátké**, jsou metodou **insert sort** prodlouženy.
- ❑ Jsou setřídovány vždy dvě sousední podposloupnosti – **stabilní** metoda.
- ❑ Nalezené/vytvořené podposloupnosti nemusí být setříděny hned, ale mohou být odloženy na **zásobník**. Díky tomu dochází k setřídění podobně dlouhých podposloupností.
- ❑ Použití dalších technik pro **zlepšení výkonnosti**: Binary-search (pro nalezení první/poslední pozice, které se dotkne vkládání), galloping mode (při vkládání více prvků za sebou ze stejné podposloupnosti), detekce klesajících posloupností, velikost běhů atd.
- ❑ Časová složitost je lineární, nepracuje in situ.

Tim sort

□ Spojení dvou posloupností:

1	2	3	6	10
---	---	---	---	----

4	5	7	9	12	14	17
---	---	---	---	----	----	----

- Binárním vyhledáváním je nalezena pozice na kterou bude vložen první prvek druhé posloupnosti a současně ve druhé posloupnosti je nalezena pozice, kam bude vložen poslední prvek první posloupnosti.
- Prvky před (a za) touto pozicí jsou již na svém místě, zbývá setřídit ostatní.
- Do pomocného pole je přesunuta ta menší, ze zbývajících posloupností. V našem případě prvky 6 a 10.

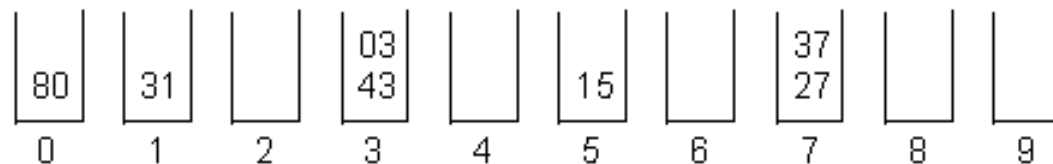
Řazení tříděním podle základu – Radix sort

- **Řazení tříděním podle základu** je počítačová verze procesu řazení na děroštitkových třídících strojích.
 - Na těchto strojích je základem desítková číslice na daném řádu v daném sloupci štitku. Ve většině počítačových aplikací je to desítková číslice čísla v kódu BCD (Binary-Coded-Decimal).
- Radix sort využívá **pomocné datové struktury**:
 - Seznamy (příp. fronty) prvků pro stejnou cifru.
 - Pole pro uchování začátků jednotlivých seznamů.
- Řazení tříděním lze implementovat tak, aby šlo o metodu pracující **bez přesunu položek**.
- Řazení tříděním podle základu je jednou z verzí tzv. *přihrádkového třídění* (bucket sort), které lze použít i na jiné než číselné klíče.

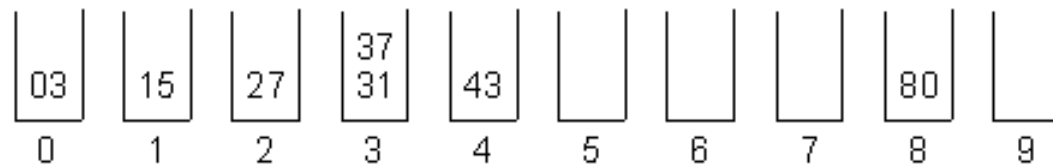
Radix sort

- Implementace s využitím *příhrádek* (např. s využitím jednosměrných seznamů).
- Po každém roztrídění prvků jsou prvky znovu spojeny do jedné posloupnosti.

43 27 31 15 37 80 03



80 31 43 03 15 27 37



03 15 27 31 37 43 80

Radix sort

Vstupní posloupnost vlož do seznamu S

for $j \leftarrow (1, \text{POCCIF})$ **do**

 // inicializace přihrádek

 // třídění prvků ze seznamu S do přihrádek dle j -té číslice

 // vytvoření prázdného seznamu S

 // postupné připojení všech přihrádek do seznamu S

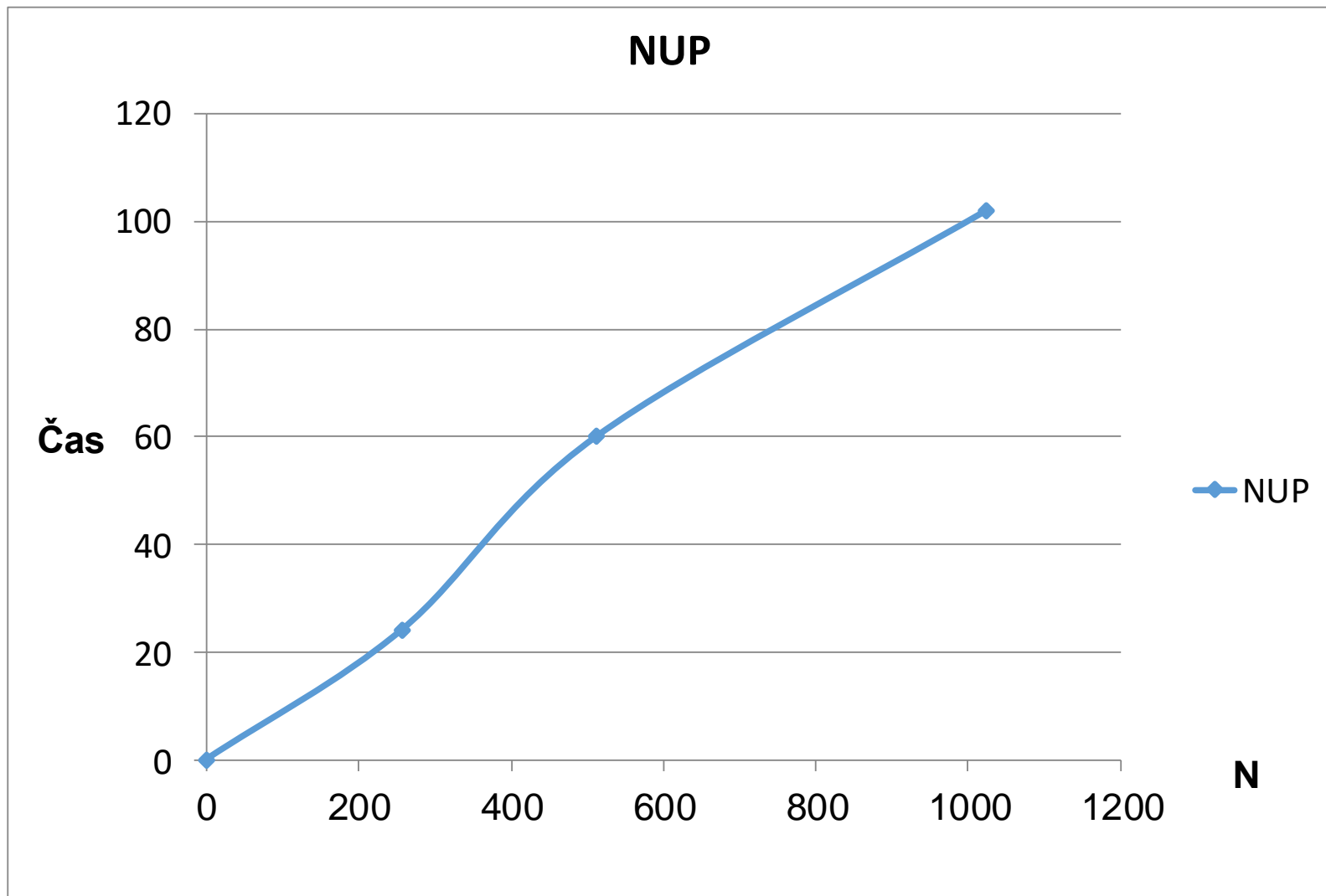
end for

Hodnocení Radix sortu

- ❑ Radix sort je **stabilní** metoda.
- ❑ Stav uspořádání nemá podstatný vliv na čas a proto se jeví, jako by se **nechoval přirozeně**.
- ❑ Metoda **nepracuje in situ**.
- ❑ Časová složitost je **lineární**.
- ❑ Experimentálně byly naměřeny tyto hodnoty pro náhodně uspořádaném pole:

N	256	512	1 024
NUP	24	60	102

RadixSort – naměřené výsledky graficky



Zhodnocení řadících metod

Algoritmus	Časová složitost	Pomocná paměť	Stabilita
Bubble sort	$O(n^2)$	-	ano
Heap sort	$O(n \log n)$	-	ne
Insert sort	$O(n^2)$	-	ano
Quick sort	$O(n \log n)$	$O(\log n)$	ne
Shell sort	$O(n^2)$	-	ne
Merge sort	$O(n \log n)$	$\Theta(n)$	ano
Radix sort	$O(n \log n)$	$\Theta(n)$	ano

Pozn.: Quick sort má uvedenou složitost pro vhodně zvolený medián.