Níže uvedené úlohy představují přehled otázek, které se vyskytly v tomto nebo v minulých semestrech ve cvičení nebo v minulých semestrech u zkoušky. Mezi otázkami semestrovými a zkouškovými není žádný rozdíl, předpokládáme, že připravený posluchač dokáže zdárně zodpovědět většinu z nich.

Tento dokument je k dispozici ve variantě převážně s řešením a bez řešení.

Je to pracovní dokument a nebyl soustavně redigován, tým ALG neručí za překlepy a jazykové prohřešky, většina odpovědí a řešení je ale pravděpodobně správně :-).

---------- SORT STABILITY ----------

1.
Stabilní řazení je charakterizováno následujícím tvrzením

a) řazení nemá asymptotickou složitost větší než $\Theta(n \cdot \log(n))$
b) řazení nemá asymptotickou složitost menší než $\Theta(n \cdot n)$
c) řazení nemění pořadí prvků s různou hodnotou
d) řazení nemění pořadí prvků se stejnou hodnotou
e) kód řazení nemusí kontrolovat překročení mezí polí

Řešení Stabilita řazení nesouvisí s rychlostí řazení, varianty a) a b) odpadají. Kdyby algoritmus neměnil pořadí prvků s různou hodnotou, nic by neseřadil, varianta c) nemá smysl. Kontrola mezí pole je záležitostí pouze implementační, s formulací algoritmu samotného nemá nic společného, varianta e) odpadá rovněž. Zbývá tak jen varianta d), jež je ostatně víceméně definicí stabilty.

2.
What sequence appears as a result of a stable sorting algorithm sorting the sequence
$B_1\ C_2\ C_1\ A_2\ B_2\ A_1$ where $A_1 = A_2 < B_1 = B_2 < C_1 = C_2$?

a) $A_1\ B_1\ C_1\ A_2\ B_2\ C_2$
b) $A_1\ A_2\ B_1\ B_2\ C_1\ C_2$
c) $B_1\ C_1\ A_1\ C_2\ A_2\ B_2$
d) $A_2\ A_1\ B_1\ B_2\ C_2\ C_1$
e) $C_1\ C_2\ A_1\ A_2\ B_1\ B_2$

3.
What sequence appears as a result of a stable sorting algorithm sorting the sequence
$C_1\ A_2\ B_2\ C_2\ A_1\ B_1$, where: $A_1 = A_2 < B_1 = B_2 < C_1 = C_2$?

a) $A_1\ B_1\ C_1\ A_2\ B_2\ C_2$
b) $A_1\ A_2\ B_1\ B_2\ C_1\ C_2$
c) $C_1\ A_1\ B_1\ C_2\ A_2\ B_2$
d) $A_2\ A_1\ B_2\ B_1\ C_1\ C_2$
e) $C_1\ C_2\ A_1\ A_2\ B_1\ B_2$

4.
Jaká posloupnost vznikne, když stabilní řadící algoritmus seřadí posloupnost
$B_1\ C_2\ A_2\ B_2\ C_1\ A_1$, v níž platí $A_1 = A_2 < B_1 = B_2 < C_1 = C_2$?

f) $A_1\ B_1\ C_1\ A_2\ B_2\ C_2$
g) $A_1\ A_2\ B_1\ B_2\ C_1\ C_2$
h) $B_1\ C_1\ A_1\ C_2\ A_2\ B_2$
i) $A_2\ A_1\ B_1\ B_2\ C_2\ C_1$
j) $A_1\ A_2\ B_1\ B_2\ C_1\ C_2$

5.

Jaká posloupnost vznikne, když stabilní řadící algoritmus seřadí posloupnost
A_2 C_2 B_2 B_1 C_1 A_1, v níž platí $A_1 = A_2 < B_1 = B_2 < C_1 = C_2$?

a) A_1 B_1 C_1 A_2 B_2 C_2
b) A_2 A_1 B_2 B_1 C_2 C_1
c) B_1 C_1 A_1 C_2 A_2 B_2
d) A_2 A_1 B_1 B_2 C_2 C_1
e) A_1 A_2 B_1 B_2 C_1 C_2

6.
The input a stable sorting algorithm is the sequence B_1 C_2 A_2 B_2 C_1 A_1. The following holds:
$A_1 = A_2 < B_1 = B_2 < C_1 = C_2$. What sequence represents the output of the algorithm?

a) A_1 B_1 C_1 A_2 B_2 C_2
b) A_1 A_2 B_1 B_2 C_1 C_2
c) B_1 C_1 A_1 C_2 A_2 B_2
d) A_2 A_1 B_1 B_2 C_2 C_1
e) A_1 A_2 B_1 B_2 C_1 C_2

7.
Stabilní algoritmus řadí do neklesajícího pořadí číselnou posloupnost
$A_2, B_2, A_3, B_3, A_1, B_1$, v níž platí $A_1 = A_2 = A_3 < B_1 = B_2 = B_3$.
Po seřazení vznikne posloupnost

f) $A_1, B_1, A_2, B_2, A_3, B_3$
g) $A_1, A_2, A_3, B_1, B_2, B_3$
h) $B_1, B_2, B_3, A_1, A_2, A_3$
i) $A_2, A_3, A_1, B_2, B_3, B_1$
j) $A_3, A_2, A_1, B_3, B_2, B_1$

8.
Stabilní algoritmus řadí do neklesajícího pořadí číselnou posloupnost
$B_2, A_2, B_1, A_1, B_3, A_3$, v níž platí $A_1 = A_2 = A_3 < B_1 = B_2 = B_3$.
Po seřazení vznikne posloupnost

a) $B_1, A_1, B_2, A_2, B_3, A_3$
b) $A_1, B_1, A_2, B_2, A_3, B_3$
c) $A_2, A_1, A_3, B_2, B_1, B_3$
d) $A_1, A_2, A_3, B_1, B_2, B_3$
e) $A_3, A_2, A_1, B_3, B_2, B_1$

9.
A stable algorithm sorts a sequence. The result is sorted in non-decreasing order. The original sequence is: $B_2, A_2, B_1, A_1, B_3, A_3$, and it also holds: $A_1 = A_2 = A_3 < B_1 = B_2 = B_3$. The sorted result is:

k) $A_1, B_1, A_2, B_2, A_3, B_3$
l) $A_1, A_2, A_3, B_1, B_2, B_3$
m) $B_1, B_2, B_3, A_1, A_2, A_3$
n) $A_2, A_3, A_1, B_2, B_3, B_1$
o) $A_3, A_2, A_1, B_3, B_2, B_1$

---------- SELECTION SORT ----------

1.

V určitém problému je velikost zpracovávaného pole s daty rovna rovna $2n^3 \cdot \log(n)$ kde n charakterizuje velikost problému. Pole se řadí pomocí Selection sort-u. Asymptotická složitost tohoto algoritmu nad uvedeným polem je tedy

a) $\Theta(n^2)$
b) $\Theta(n^6 \cdot \log^2(n))$
c) $\Theta(n^3 \cdot \log(n))$
d) $\Theta(n^3 \cdot \log(n) + n^2)$
e) $\Theta(n^5 \cdot \log(n))$

Řešení Máme-li velikost pole rovnu k, Selection sort jej zpracuje v čase $\Theta(k^2)$. Velikost našeho pole je $k = 2n^3 \cdot \log(n)$, takže bude zpracováno v čase $\Theta(k^2) = \Theta((2n^3 \cdot \log(n))^2) =$
$= \Theta(4n^6 \cdot \log^2(n)) = \Theta(n^6 \cdot \log^2(n))$.
Platí tedy možnost b).

2.
Pole n různých prvků je uspořádáno od druhého prvku sestupně, první prvek má nejmenší hodnotu ze všech prvků v poli.
Zatrhněte všechny možnosti, které pravdivě charakterizují asymptotickou složitost Selection Sortu (třídění výběrem) pracujícího nad tímto konkrétním polem.
$O(n)$ $\Omega(n)$ $\Theta(n)$ $O(n^2)$ $\Omega(n^2)$ $\Theta(n^2)$

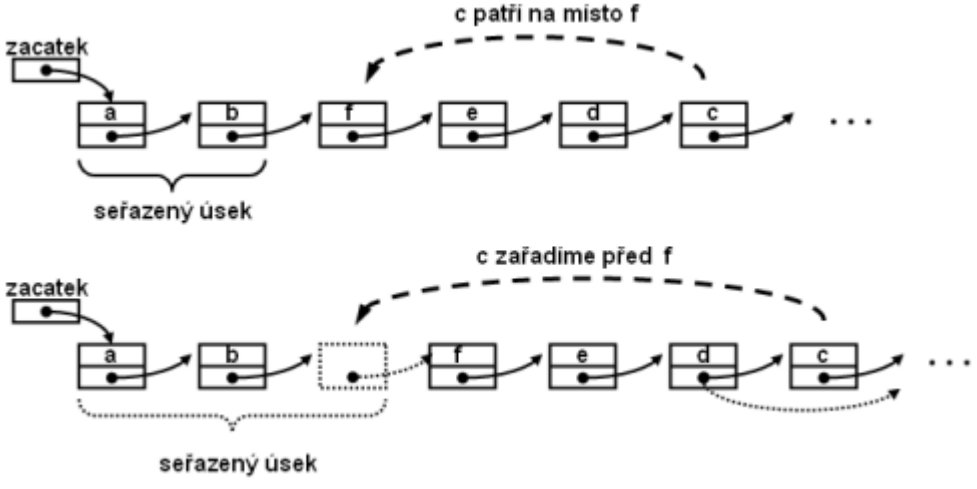
Řešení Pro výběr aktuálního minima musí Select Sort pokaždé zkontrolovat všechny prvky od aktuálního až do konce pole, takže jeho složitost je je vždy právě $\Theta(n^2)$ nezávisle na datech v poli.
V zadání je tedy nutno zatrhnout možnosti $\Omega(n)$, $O(n^2)$, $\Omega(n^2)$, $\Theta(n^2)$

3.
Implementujte Selection sort pro jednosměrně zřetězený spojový seznam.

Řešení (Pro přehlednost připomínáme nejprve kód pro řazení v poli)

```
for (i = 0; i < n-1; i++) {
 // select min
  jmin = i;
  for (j = i+1; j < n; j++) {
    if (a[j] < a[jmin]) {
      jmin = j;
    }
  }
 // make place for min
  min = a[jmin];
  for (j = jmin; j > i; j--) {
    a[j] = a[j-1];
 // put min
  a[i] = min;
}
```

Vnější cyklus algoritmu obsahuje tří fáze: Nalezení minima, uvolnění prostoru pro přesouvané minimum a vložení minima na správné místo.
V zřetězeném seznamu je myšlenka jednodušší: Nalezení minima, vyjmutí minima ze seznamu a jeho



vložení na příslušné místo. Ostatní prvky seznamu se automaticky posunou. Pracujeme totiž s celými prvky seznamu, nikoli jen s jejich hodnotami, jak to – celkem zbytečně – činil nejeden respondent této otázky. Obrázek snad naznačuje myšlenku:

Protože máme k dispozici jen jednosměrný seznam, budeme se potýkat s určitými potížemi.
Zaprvé musíme na daný prvek ukazovat nikoli přímo ale pomocí jeho předchůdce v seznamu, abychom mohli korektně prvek vyjímat a vkládat z/do seznamu. Díky neexistenci předchůdce prvního prvku budeme navíc muset nalezení nejmenšiho prvku v seznamu obsloužit zvlášť.
Vystačíme si jen se čtyřmi dalšími ukazateli: **`uk_i`** pro vnější cyklus, **`uk_j`** pro vnitřní cyklus a pomocnými ukazateli **`presouvany`**, resp. **`predPresouvany`**, pomocí kterých budeme manipulovat s přesouvaným prvkem, resp ukazovat na předchůdce prvku, který má být přesunut. Doplnění chybějících deklarací v uvedeném pseudokódu necháváme na čtenáři.

```
prvekSeznamu selectSort (prvekSeznamu zacatek) {

if (zacatek == null) return;
if (zacatek.next == null) return;

// celkove minimum:
// najdeme jej v seznamu pocinaje druhym prvkem,
// pamatujme, ze na nic nemuzeme ukazovat primo diky jednosmernemu zretezeni
uk_j = zacatek;
predPresouvany = zacatek;
while (uk_j.next != null)
   if (uk_j.next.value < predPresouvany.next.value)
      predPresouvany = uk_j;
    // minimum v useku od druheho prvku nalezeno
    if (zacatek.value <= predPresouvany.next.value) { // minimum je na zacatku,
    }                                                 // nebudeme delat nic
    else { // presun minima  na zacatek
     // nejprve jej vyjmeme ...
     presouvany = predPresouvany.next;
     predPresouvany.next = presouvany.next;
     // ... a pak zaradime
     presouvany.next = zacatek;
     zacatek = presouvany;
}

// nyni jiz muzeme nasadit obecny cyklus
uk_i = zacatek;
while (uk_i.next.next != null) {
  // najdeme minimum v neserazene casti seznamu
  // uk_i ukazuje na posledni prvrk v serazene casti
  predPresouvany = uk_i;
  uk_j = predPresouvany.next;
  while(uk_j.next != null) {
    if (uk_j.next.value < predPresouvany.next.value)
       predPresouvany = uk_j;

  // predPresouvany ted ukazuje na prvek pred minimem, nastane presun minima
  if (predPresouvany == uk_i)  {} // neni co delat,
  else {
     // vyjmeme presouvany ze seznamu...
     presouvany = predPresouvany.next;
     predPresouvany.next = presouvany.next
     // ... a zaradime ZA uk_i
     presouvany.next = uk_i.next;
     uk_i.next = presouvany;
  } // konec presunu
```

```
  uk_i = uk_i.next;     // posuneme uk_i pro dalsi beh vnejsiho cyklu
} // konec vnejsiho cyklu
return zacatek; // ktery se mohl v prubehu razeni zmenit
}
```

---------- **INSERT SORT** ----------

1.
Insert sort řadí pole obsahující prvky 1 2 4 3 (v tomto pořadí). Celkový počet vzájemných porovnání prvků pole při tomto řazení bude

a) 3
b) 4
c) 5
d) 7
e) 8

2.
Insert sort sorts the array containing elements 1 2 4 3 (in this order). The total number of comparisons of elements in this particular case will be:

f) 3
g) 4
h) 5
i) 7
j) 8

3.
Insert Sort sorts a sequence 4 3 1 2. Number of element comparisons during the sort is

a) 3
b) 4
c) 5
d) 6
e) 8

4.
Insert Sort sorts a sequence 2 1 4 3. Number of comparisons it performs during the search is

a) 1
b) 3
c) 4
d) 5
e) 7

5.
Nahradíme-li Selection sort Insert sort-em, pak asymptotická složitost řazení
a) nutně klesne pro každá data

b) nutně vzroste pro každá data
c) může klesnout pro některá data
d) může vzrůst pro některá data
e) zůstane beze změny pro libovolná data

Řešení Tady je patrně nutné si pamatovat, že asymptotická složitost Selection sortu je $\Theta(n^2)$, zatímco asymptotická složitost Insert sortu je $O(n^2)$. To znamená, že existují případy, kdy Insert sort svá data seřadí v čase asymptoticky kratším než n^2. Na těchto datech však asymptotická složitost Selection sortu zůstane stejná, protože ta je rovna $\Theta(n^2)$ vždy. To znamená, že při přechodu od Selection sortu k Insert sortu nad týmiž daty může asymptotická složitost někdy klesnout, což odpovídá variantě c).

6.

Čtyři prvky řadíme pomocí Insert Sortu. Celkový počet vzájemných porovnání prvků je

a) je alespoň 4
b) je nejvýše 4
c) je alespoň 3
d) může být až 10
e) je vždy roven $4*(4-1)/2 = 6$.

Řešení Nejmenší možný počet testů je:
při zařazení 2. prvku - 1,
při zařazení 3. prvku - 1,
při zařazení 4. prvku - 1.
Tedy celkem 3 porovnání, pokud je posloupnost prvků rostoucí již před seřazením.
Největší možný počet testů je
při zařazení 2. prvku - 1,
při zařazení 3. prvku - 2,
při zařazení 4. prvku - 3.
Tedy celkem 6 porovnání, pokud je posloupnost prvků klesající před seřazením. Pro 4 prvky tedy Insert sort vykoná 3 až 6 porovnání prvků, podle toho, jaká má data. S tímto zjištěním koresponduje pouze varianta c).

7.

V určitém problému je velikost zpracovávaného pole s daty rovna rovna $3n^2 \cdot \log(n^2)$ kde n charakterizuje velikost problému. Pole se řadí pomocí Insert sort-u.
Asymptotická složitost tohoto algoritmu nad uvedeným polem je tedy

a) $O(n^2 \cdot \log(n))$
b) $O(n^2 \cdot \log(n^2))$
c) $O(n^4 \cdot \log^2(n))$
d) $O(n^2 \cdot \log(n) + n^2)$
e) $O(n^2)$

Řešení Máme-li velikost pole rovnu k, Insert sort jej zpracuje v čase $O(k^2)$. Velikost našeho pole je $k = 3n^2 \cdot \log(n^2)$, takže bude zpracováno v čase $O(k^2) = O((3n^2 \cdot \log(n^2))^2) = = O(9n^4 \cdot \log^2(n^2)) = O(9n^4 \cdot 4 \cdot \log^2(n)) = O(n^4 \cdot \log^2(n))$. Platí tedy možnost c).

8.

Insert sort řadí (do neklesajícího pořadí) pole o n prvcích, kde jsou stejné všechny hodnoty kromě poslední, která je menší. Jediné nesprávné označení asymptotické složitosti výpočtu je

a) $O(n)$
b) $\Theta(n)$
c) $\Omega(n)$
d) $O(n^2)$
e) $\Omega(n^2)$

Řešení Zpracování prvních (n–1) prvků pole proběhne v čase úměrném (n–1), protože žádný prvek se nebude přesouvat a zjištění toho, že se nebude přesouvat, proběhne v konstantním čase. Poslední prvek se pak přesune na začátek, což opět proběhne v čase úměrném n. Celkem tedy veškeré řazení proběhne v čase úměrném n. Platí ovšem

$konst \cdot n \in O(n)$
$konst \cdot n \in \Theta(n)$
$konst \cdot n \in \Omega(n)$
$konst \cdot n \in O(n^2)$

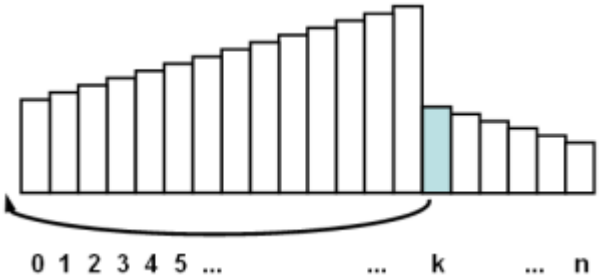
Jediná nepravdivá varianta je e).

9.

Insert sort řadí pole seřazené sestupně. Počet porovnání prvků za celý průběh řazení bude

a) n
b) n–1
c) n*(n–1)/2
d) n*n
e) $\log_2(n)$

Řešení V k-tém kroku řazení je nutno prvek na pozici k zařadit na jemu odpovídající místo v seřazeném úseku nalevo od k (předpokládáme pole indexované 0.. n–1). Tento prvek je však díky původnímu sestupnému seřazení menší než všecny prvky nalevo od něj, tudíž musí být zařazen na úplný začátek, což znamená, že bude porovnán se všemi prvky nalevo od něj jichž je právě k. Viz obrázek.



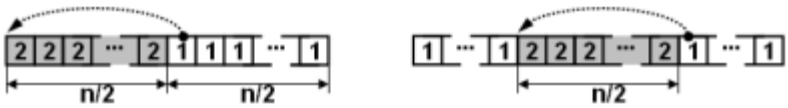
V k tém kroku tedy bude provedeno k porovnání.
Celkový počet porovnání tedy je 0 + 1 + 2 + 3 + ... + (n–1) = n*(n–1)/2. Platí varianta c).

10.

Insert sort řadí (do neklesajícího pořadí) pole o n prvcích, kde v první polovině pole jsou pouze dvojky, ve druhé polovině jsou jen jedničky. Jediné nesprávné označení asymptotické složitosti výpočtu je

a) $\Theta(n)$
b) $\Omega(n)$
c) $O(n^2)$
d) $\Theta(n^2)$
e) $\Omega(n^2)$

Řešení Na obrázku máme znázorněnou situaci před prvním přesunem jedničky první a pak situaci před některým dalším přesunem jedničky.



V každém kroku je nutno jednu jedničku přesunout o $n/2$ pozic doleva a těchto kroků bude $n/2$ (protože jedniček je $n/2$). Počet provedených operací bude tedy úměrný hodnotě výrazu $n/2 \cdot n/2 = n^2/4 \in \Theta(n^2)$. Z toho, že $n^2/4 \in \Theta(n^2)$, vyplývá bezprostředně také, že $n^2/4 \in O(n^2)$, $n^2/4 \in \Omega(n^2)$, $n^2/4 \in \Omega(n)$ a navíc $n^2/4 \notin O(n)$. Pravdivé jsou tedy varianty b) – d), nepravdivá je jedině varianta a).

11.

V poli velikosti n mají všechny prvky stejnou hodnotu kromě posledního, který je menší.
Zatrhněte všechny možnosti, které pravdivě charakterizují asymptotickou složitost Insert Sortu (třídění vkládáním) pracujícího nad tímto konkrétním polem.

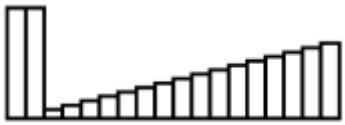
$O(n)$ $\Omega(n)$ $\Theta(n)$ $O(n^2)$ $\Omega(n^2)$ $\Theta(n^2)$

Řešení Insert Sort postupuje od začátku pole a každý prvek zařadí do již seřazeného počátečního úseku pole. protože ale jsou všechny prvky stejně velké, zařazení každého prvku se redukuje na to, že bude ponechán na místě. To je operaces konstantní časovou složitostí, takže prvních n–1 prvků bude zpracováno v čase $\Theta(n)$. Poslední prvek v poli je sice menší, takže patrně nebude zpracován v konstantním čase. Zpracování každého jednotlivého prvku při řazení Insert Sort-em má však složitost $O(n)$, takže celková složitost řazení bude $\Theta(n) + O($

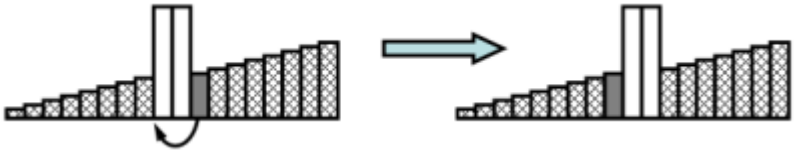
12.
nsert sort řadí do neklesajícího pořadí pole o n prvcích kde hodnoty od třetího do posledního prvku rostou a hodnoty prvních dvou prvků jsou stejné a v poli největší. Jediné nepravdivé označení asymptotické složitosti výpočtu je

a) $O(n)$
b) $\Theta(n)$
c) $O(n^2)$
d) $\Omega(n^2)$
e) $\Omega(n)$

1.



2.

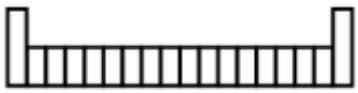


Situace v poli před zahájením řazení je naznačena na prvním obrázku. V obecném kroku algoritmu se nad daným polem provede akce znázorněná na druhém obrázku. Dva velké prvky se posunou o pozici doprava a aktuální (šedý) prvek se posune o dvě pozice doleva. To představuje konstantní počet operací. Rovněž první krok algoritmu, kdy se pouze porovnají první dva prvky, potřebuje konstantní počet operací. Celkem tedy složitost výpočtu je úměrná výrazu *konst* · n. Platí ovšem
konst · $n \in O(n)$ (*konst* · n neroste asymptoticky rychleji než n)
konst · $n \in \Theta(n)$ (to jsme právě zdůvodnili)
konst · $n \in O(n^2)$ (a libovolné vyšší mocniny n)
konst · $n \in \Omega(n)$ (*konst* · n neroste asymptoticky pomaleji než n)
Jediné nepravdivé tvrzení je tudíž obsaženo ve variantě d).

13.
Insert sort řadí do neklesajícího pořadí pole o n prvcích, v němž jsou stejné všechny hodnoty kromě první a poslední, které jsou větší a navzájem stejné. Jediné nepravdivé označení asymptotické složitosti výpočtu je

f) $O(n)$
g) $\Theta(n)$
h) $\Omega(n^2)$
i) $O(n^2)$
j) $\Omega(n)$

1.



2.

Situace v poli před zahájením řazení je naznačena na prvním obrázku. V obecném kroku algoritmu se nad daným polem provede akce znázorněná na druhém obrázku. Jeden velký prvek se posune o pozici doprava a aktuální (šedý) prvek se posune o pozici doleva. To představuje konstantní počet operací. Rovněž poslední krok algoritmu, kdy se pouze porovnají poslední dva prvky, potřebuje konstantní počet operací. Celkem tedy složitost výpočtu je úměrná výrazu *konst* · n. Platí ovšem
konst · $n \in O(n)$ (*konst* · n neroste asymptoticky rychleji než n)
konst · $n \in \Theta(n)$ (to jsme právě zdůvodnili)

konst · *n* ∈ $O(n^2)$ (a libovolné vyšší mocniny *n*)
konst · *n* ∈ $\Omega(n)$ (*konst* · *n* neroste asymptoticky pomaleji než *n*)
Jediné nepravdivé tvrzení je tudíž obsaženo ve variantě c).

14.
We perform Insert sort on an array of length *n*.The array values grow from the beginning up to the middle of the array and then they decrease towards the array end. Asymptotic complexity of this process on the given array is

a) $\Theta(n \cdot \log(n))$
b) $O(n \cdot \log(n))$
c) $O(\log(n))$
d) $O(n^2)$
e) $\Theta(n/2 + n/2)$

15.
Napište rekurzivní verzi algoritmu Insert sort. Při návrhu algoritmu postupujte metodou shora dolů.

Řešení Toto je vůbec triviální úloha. Celý algoritmus Insert sortu se skládá ze dvou vnořených cyklů. Aby se stal rekurzivním, změníme prostě vnější cyklus na rekurzivní volání, vše ostatní zachováme beze změny.

```
void insertSortRek(int k, int n) {     // n je počet prvků v poli
  přesuň prvek z pozice k na jemu příslušné místo jako v nerekurzivní verzi;
  if (k < n) insertSortRek(k+1,n);
}
```

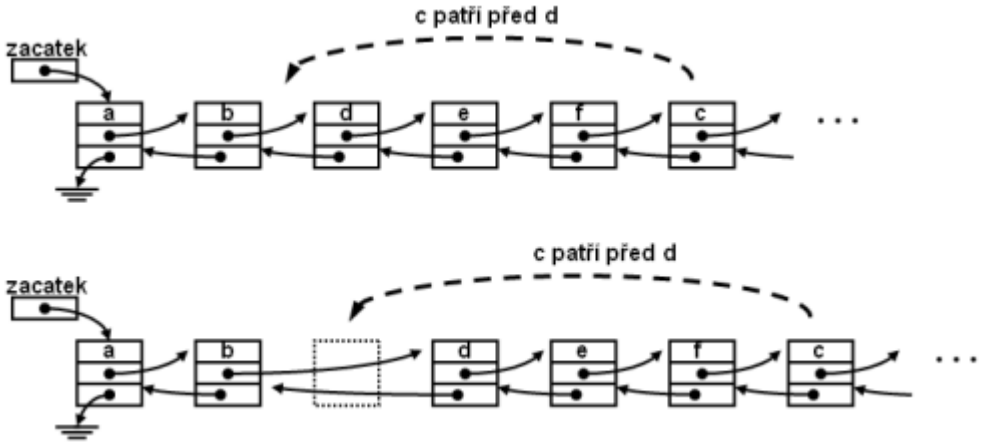
Jak probíhá vkládání prvku na jemu příslušné místo pro jistotu popište také. Celé řazení pak proběhne po zavolání **`insertSortRek(1,n);`**.

16.
Implementujte Insert sort pro obousměrně zřetězený spojový seznam.

Řešení (Pro přehlednost nejprve kód pro řazení v poli)

```
( 1) for (i = 1; i < n; i++) {
( 2)   // find & make place for a[i]
( 3)   insVal = a[i];
( 4)   j = i-1;
( 5)   while ((j >= 0) && (a[j] > insVal)) {
( 6)     a[j+1] = a[j];
( 7)     j--;
( 8)   }
( 9)   // insert a[i]
(10)   a[j+1] = insVal;
(11) }
```

Ve vnitřním cyklu (řádky 6—8) se vyhledává patřičné místo po vkládanou hodnotu insVal a přitom se ostatní hodnoty posunují o jednu pozici doprava, aby tak vkládané hodnotě uvolnily místo v poli. Mnozí respondenti zachovali tuto strategii i při použití zřetězeného seznamu, přestože je v tomto případě naprosto zbytečná. Stačí pouze přesunout celý prvek seznamu (i s jeho hodnotou) na patřičné místo a ostatní relevantní prvky seznamu (i se svými hodnotmi) se posunou o jednu pozici doprava zcela automaticky. Snad to dostatečně ilustruje následující obrázek.



Budou nám stačit ukazatele **uk_i** pro vnější cyklus a **uk_j** pro vnitřní cyklus a pomocny ukazatel **presouvany**, pomocí kterého budeme manipulovat s přesouvaným prvkem.
Doplnění chybějících deklarací v uvedeném pseudokódu necháváme na čtenáři.

```
prvekSeznamu insertSort (prvekSeznamu zacatek) {
if (zacatek == null) return;
if (zacatek.next == null) return;
uk_i = zacatek.next;
while (uk_i != null) {
  uk_j = uk_i.prev;
  // najdi misto pro zarazovany prvek, na nejz ukazuje uk_i
  while ((uk_j != null) && (uk_j.value > uk_i.value))
     uk_j = uk_j.prev;
  // misto nalezeno, vkladej:

  if (uk_j.next == uk_i)
      // pokud uz byl prvek pod uk_i na spravnem miste
      uk_i = uk_i.next;  // tak jen posunem uk_i

  else { // nastane opravdovy presun
     // nejprve ovsem prvek z jeho mista korektne vyjmeme...:
     presouvany = uk_i;
     uk_i = uk_i.next;   // nutny posun pro dalsi beh vnejsiho cyklu

     // uz jen manipulaci s ukazateli
     // presuneme presouvany prvek kam patri
        // uvolnime stavajici vazby...
     presouvany.prev.next = presouvany.next;
     if (presouvany.next != null)
        presouvany.next.prev = presouvany.prev;
        // ... a vkladame jinam:
        // ale co kdyz musime vkladat na uplny zacatek?
     if (uk_j == null) {
        presouvany.prev = null;
        presouvany.next = zacatek.next;
        zacatek.prev = presouvany;
        zacatek = presouvany;
     }
     else { // vkladame ZA uk_j
        presouvany.next = uk_j.next;
        presouvany.prev = uk_j;
        uk_j.next = presouvany;
        preouvany.next.prev = presouvany;
     } // konec vkladani ZA uk_j
  } // konec presunu
```

```
} // konec vnejsiho cyklu
return zacatek; // ktery se mohl v prubehu razeni zmenit
}
```

Poznámka. Také by příkaz **`uk_i = uk_i.next;`** mohl být v kódu spíše jen jednou před koncem vnějšího cyklu...

---------- QUICK SORT ----------

1.
Proveďte (v mysli nebo na papíře) jeden průchod uvedeným polem, který v Quick Sortu rozdělí prvky na „malé“ a „velké“. Jako pivotní hodnotu zvolte hodnotu posledního prvku v poli. Napište, v jakém pořadí budou po tomto průchodu prvky v poli uloženy, a vyznačte, kde bude pole rozděleno na „malé“ a „velké“.

6 10 8 5 7 2 3 9 1 4

Řešení 4 1 3 2 | 7 5 8 9 10 6

- ○

2.
Opakujte předchozí úlohu, s tím, že za pivotní hodnotu zvolíte hodnotu prvního prvku v poli

Řešení 4 1 3 5 2 | 7 8 9 10 6

3.
Quick sort řadí následující čtyřprvkové pole čísel.

2 4 1 3

Jako pivotní hodnotu volí první v pořadí tj. nejlevější. Jak bude pole rozděleno na „malé“ a „velké“ hodnoty po jednom průchodu polem? (Svislá čára naznačuje dělení.)

a) **1 | 4 2 3**
b) **1 2 | 3 4**
c) **1 2 | 4 3**
d) **2 1 | 3 4**

Řešení (Reference v textu odkazují na kód níže.)
Nejprve se určí pivot (=2) a nastaví se indexy L = 1, P = 4.

```
L       P
2 4 1 3
```

I. Dokud L vidí hodnotu ostře menší než pivot, pohybuje se doprava, takže v tomto příipadě jenom zůstane stát (řádek 6 v kódu).

II. Dokud P vidí hodnotu ostře větší než pivot, pohybuje se doleva, takže 3 nechá být a zastaví se u hodnoty 1 (řádek 7 v kódu).

```
L     P
2 4 1 3
```

III. Pokud je L < P (a to teď je), prohodí se prvky pod L a P

```
L     P
1 4 2 3
```

a vzapětíi se povinně L posune o krok doprava a R o krok doleva (řádky 8-10 v kódu).

```
  LP
1 4 2 3
```

IV. Teď dochází algoritmus k podmínce vnějšího cyklu while(iL <= iR) (na řádku 15) což znamená, že vnější cyklus ještě nekončí a bude se opakovat počínaje řádkem 6:

V. Dokud L vidí hodnotu ostře menší než pivot, pohybuje se doprava, takže v tomto příipadě jenom zůstane stát (řádek 6 v kódu).

VI. Dokud P vidí hodnotu ostře větší než pivot, pohybuje se doleva, takže 4 nechá být a zastaví se u hodnoty 2 (řádek 7 v kódu).

```
P    L
1    4    2    3
```

VII. Nyní neplatí podmínka na řádku 8 (iL < iR) a neplatí ani podmínka v části else na řádku 12 (iL == iR), takže se neprovede vůbec nic.

VIII. Podminka opakovani vnejsiho cyklu na radku 15 (iL <= iR) již neplatí, cyklus končí a přechazí se k rekurzivnímu voláni. "Malé" prvky se nalezají na začátku pole až do indexu P včetně, takže obsahují hodnotu: 1. "Velké“ prvky se nalezají na konci pole počínaje indexem L, takže obsahují hodnoty 4, 2, 3 (v tomto pořadí).

Pole je tedy rozděleno mezi prvním a druhým prvkem stejně jako ve variantě a).

Kód QuickSortu

```
 1 void qSort(Item a[], int low, int high) {
 2   int iL = low, iR = high;
 3   Item pivot = a[low];
 4
 5   do {
 6     while (a[iL] < pivot) iL++;
 7     while (a[iR] > pivot) iR--;
 8     if (iL < iR) {
 9       swap(a,iL, iR);
10        iL++; iR--;
11      }
12      else {
13        if (iL == iR) { iL++; iR--;}
14      }
15    while( iL <= iR);
16
17    if (low < iR)  qSort(a, low, iR);
18    if (iL < high) qSort(a, iL, high);
19  }
```

4.

Quick sort řadí následující čtyřprvkové pole čísel.

```
     3   2   1   4
```

Jako pivotní hodnotu volí první v pořadí tj. nejlevější. Jak bude pole rozděleno na „malé“ a „velké“ hodnoty po jednom průchodu polem? (Svislá čára naznačuje dělení.)

a) **1 | 2   4   3**
b) **1   2 | 3   4**
c) **1   2 | 4   3**
d) **2   3 | 1   4**
e) **1   2   3 | 4**

Řešení (Reference v textu odkazují na kód výše.)

Nejprve se určí pivot (=3) a nastaví se indexy L = 1, P = 4.

```
L            P
3   2   1   4
```

I. Dokud L vidí hodnotu ostře menší než pivot, pohybuje se doprava, takže v tomto příipadě jenom zůstane stát (řádek 6 v kódu).

II. Dokud P vidí hodnotu ostře větší než pivot, pohybuje se doleva, takže 3 nechá být a zastaví se u hodnoty 1 (řádek 7 v kódu).

```
L       P
3   2   1   4
```

III. Pokud je L < P (a to teď je), prohodí se prvky pod L a P

```
L       P
1   2   3   4
```

a vzapětíi se povinně L posune o krok doprava a R o krok doleva (řádky 8-10 v kódu).

```
    LP
1   2   3   4
```

IV. Teď dochází algoritmus k podmínce vnějšího cyklu while(iL <= iR) (na řádku 15) což znamená, že vnější cyklus ještě nekončí a bude se opakovat počínaje řádkem 6:

V. Dokud L vidí hodnotu ostře menší než pivot, pohybuje se doprava, takže v tomto případě se posune o krok dále nad hodnotu 3 (rovnou pivotu) (řádek 6 v kódu).

```
    P   L
1   2   3   4
```

VI. . Dokud P vidí hodnotu ostře větší než pivot, pohybuje se doleva, takže v tomto případě neudělá nic, protože vidí hodnotu (2) menší než pivot (3) (řádek 7 v kódu).

```
    P   L
1   2   3   4
```

VII. Nyní neplatí podmínka na řádku 8 (iL < iR) a neplatí ani podmínka v části else na řádku 12 (iL == iR), takže se neprovede vůbec nic.

VIII. Podmínka opakováaní vnějšího cyklu na řádku 15 (iL <= iR) již neplatí, cyklus končí a přechazí se k rekurzivnímu voláni. "Malé" prvky se nalezají na začátku pole až do indexu P včetně, takže obsahují j hodnoty: 1 a 2. "Velké“ prvky se nalezají na konci pole počínaje indexem L, takže obsahují hodnoty 3, 4 (v tomto pořadí).

Pole je tedy rozděleno mezi druhým a třetím prvkem stejně jako ve variantě b).

5.

Quick sort řadí pole s n prvky, přičemž první polovina pole obsahuje pouze hodnoty 50, druhá polovina pole obsahuje pouze hodnoty 100. Asymptotická složitost tohoto řazení je

a) $\Theta(n)$
b) $O(n)$
c) $O(n \cdot \log(n))$
d) $\Theta(n^2)$
e) $\Omega(n^2)$

Po prvním prúchodu rozdělí quick sort pole na dvě poloviny — první bude obsahovat padesátky, druhá stovky. Nyní se bude každý úsek skládat z totožných hodnot. Quick sort řadí nejrychleji, pokud při každém průchodu rozdělí právě řazený úsek na dvě stejně velké části. To je ale i případ řazení úseku který obsahuje shodné hodnoty, na „malé“ a „velké“. Všechny hodnoty jsou rovné pivotovi, tudíž v každém kroku se vymění prvky odpovídající levému a pravému ukazateli a oba ukazatele se posunou o pozici blíže ke středu úseku. Ukazatelé se tak setkají se právě uprostřed a úsek bude rozdělen na dvě stejné části (má-li lichý počet prvků, prostřední prvek bude pominut).
Quick sort nemůže řadit rychleji než v čase úměrném $n \cdot \log(n)$, tudíž jeho asymptotická služitost v tomto případě bude právě $\Theta(n \cdot \log(n))$. Varianta a) a b) představují množiny funkcí, které rostou asymptoticky

vesměs pomaleji než n·log(n), a naopak varianty d) a e) představují množiny funkcí rostoucích vesměs asymptoticky rychleji než n·log(n). Zbývá jen správná varianta c).

6.
Quick sort řadí pole s n prvky, v němž jsou všechny hodnoty stejné, až na jedinou hodnotu v polovině pole, která je větší. Asymptotická složitost tohoto řazení je

a) $\Theta(n)$
b) $O(n+\log(n))$
c) $\Theta(n+1)$
d) $O(n \cdot \log(n))$
e) $\Omega(n^2)$

V okamžiku, kdy jsou všechny hodnoty v řazeném úseku stejné, postupuje Quick sort při dělení úseku na "velké" a "malé" tak, že vymění prvky pod pravým a levým ukazatelem (oba jsou rovné hodnotě pivota) a posune oba ukazatele o jednu pozici směrem ke středu úseku a celý proces opakuje. Ukazatelé se tak setkají právě uprostřed řazeného úseku. Když řazený úsek obsahuje jediný prvek odlišný od stejných ostatních, je průběh dělení na "malé" a "velké" obdobný, ukazatelé se setkají uprostřed. Quick sort ale řadí nejrychleji, pokud při každém průchodu rozdělí právě řazený úsek na dvě stejně velké části, což je právě náš případ. Navíc, Quick sort nemůže řadit rychleji než v čase úměrném n·log(n), tudíž jeho asymptotická služitost v tomto případě bude právě $\Theta(n \cdot \log(n))$. Varianta a), b) a c) představují množiny funkcí, které rostou asymptoticky vesměs pomaleji než n·log(n), a naopak varianta e) představuje množinu funkcí rostoucích vesměs asymptoticky rychleji než n·log(n). Zbývá jen správná varianta d).

7.
Na vstupu je neseřazené pole prvků pole[počet]. Bez toho, abyste toto pole seřadili, napište funkci, která v něm nalezne nalezne *k*-tý nejmenší prvek (= *k*-tý v seřazeném poli). Použijte metodu rozděl a panuj jako při řazení Quick-sortem, ale celé pole neřaďte. Hodnota *k* bude vstupním parametrem Vašeho algoritmu.

8.
Napište nerekurzivní verzi algoritmu Quick-sort. Při návrhu algoritmu postupujte metodou shora dolů.

Řešení Myšlenka:¨Dělení na „malé" a „velké" hodnoty v aktuálně řazeném úseku bude probíhat stejně jako v rekurzivní verzi. Jedině se změní postup, jakým budeme jednotlivé úseky registrovat. (Mimochodem, pořadí zpracování úseků se také nezmění.)
Použijeme vlastní zásobník, do něhož budeme ukládat meze (horní a dolní — jako dvojici celých čísel) úseků, které mají být zpracovány.
(Objevilo se i korektní řešení, kde autor ukládal na zásobník i meze právě zpracovaného úseku, ale musel si pamatovat více informací a algoritmus byl složitější)

Předpokládejme, že původní pole má meze 1..*n*. Celý algoritmus pak vypadá takto:

```
Vytvoř prázdný zásobník
Ulož dvojici (1,n) na zásobník
While (zásobník je neprázdný) {
   vyjmi z vrcholu zásobníku (pop) meze úseku (Low, High), který budeš teď zpracovávat
   rozděl aktuální úsek (od Low do High) stejně jako v rekurzivní verzi na dva úseky s mezemi
   (Low, iR) a (iL, High)
   if (Low < iR)   vlož (push) dvojici (Low, iR) na zásobník
   if (iL < High)   vlož (push) dvojici (iL, High) na zásobník
}
```

Pro úpný algoritmus ještě popište, jak se dělí úsek na „malé" a „velké".

---------- **MERGE SORT** ----------

1.

Merge sort

a) lze napsat tak, aby nebyl stabilní
b) je stabilní, protože jeho složitost je Θ (n*log(n))
c) je nestabilní, protože jeho složitost je Θ(n*log(n))
d) je rychlý (Θ(n*log(n))) právě proto, že je stabilní
e) neplatí o něm ani jedno předchozí tvrzení

Řešení Rychlost a stabilita řazení nejsou v žádné příčinné souvislosti, odpovědi b), c), d) jsou pouhá „mlha“. Při slévání (vemte si k ruce kód!) můžeme jednoduchou záměnou ostré a neostré nerovnosti v porovnávání prvků způsobit, že při stejných porovnávaných prvcích dostane přednost buď prvek z levé nebo z pravé části řazeného úseku. Když dostanou přednost prvky z pravé části, octnou se nakonec v seřazeném poli před těmi prvky, které byly v levé části a měly stejnou hodnotu. To je ovšem projev nestability. Platí možnost a).

2a.

V určitém problému je velikost zpracovávaného pole s daty rovna rovna $2n-2$, kde n charakterizuje velikost problému. Pole se řadí pomocí Merge Sort-u (řazení sléváním). S použitím $\Theta/O/\Omega$ symboliky vyjádřete asymptotickou složitost tohoto algoritmu nad uvedeným polem v závislosti na hodnotě n. Výsledný vztah předložte v co nejjednodušší podobě.

Řešení Pole velikosti m řadí Merge Sort pokaždé v čase $\Theta(m\cdot\log(m))$.V našem případě je $m = 2n-2$. I kdyby pole mělo velikost právě $2n$ (tedy ještě větší), byla by asymptotická složitost rovna
$\Theta(2n\cdot\log(2n)) = \Theta(n\cdot\log(2n)) = \Theta(n\cdot(\log(2) + \log(n))) = \Theta(n\cdot\log(2) + n\cdot\log(n))$.
Protože výraz $n\cdot\log(2)$ roste asymptoticky pomaleji než $n\cdot\log(n)$ můžeme dále psát:
$\Theta(n\cdot\log(2) + n\cdot\log(n)) = \Theta(n\cdot\log(n))$.
Závěr tedy je: Protože jak pole velikosti n, tak i pole velikosti $2n$ seřadí Merge Sort v asymptoticky stejném čase $\Theta(n\cdot\log(n))$, seřadí i pole velikosti $2n-2$ v témž čase $\Theta(n\cdot\log(n))$.

2b.

V určitém problému je velikost zpracovávaného pole s daty rovna rovna $2n^3$,kde n charakterizuje velikost problému. Pole se řadí pomocí Merge sort-u. S použitím $\Theta/O/\Omega$ symboliky vyjádřete asymptotickou složitost tohoto algoritmu nad uvedeným polem v závislosti na hodnotě n. Výsledný vztah předložte v co nejjednodušší podobě.

Řešení Pole velikosti m řadí Merge Sort pokaždé v čase $\Theta(m\cdot\log(m))$.V našem případě je $m = 2n^3$. Po dosazení získáváme $\Theta(m\cdot\log(m)) = \Theta(2n^3\cdot\log(2n^3)) = \Theta(2n^3\cdot(\log(2)+3\log(n)) = \Theta(2n^3\cdot\log(2)+ 6n^3\cdot\log(n))$.
Výraz log(2) je konstantní, funkce log je roustoucí, proto funkce $6n^3\cdot\log(n)$ roste asymptoticky alespoň stejně rychle nebo rychleji než funkce $2n^3\cdot\log(2)$. Při zvažovaní řádu růstu součtu těchto funkcí můžeme proto funkci $2n^3\cdot\log(2)$ zanedbat a získáváme tak: $\Theta(2n^3\cdot\log(2)+ 6n^3\cdot\log(n)) = \Theta(6n^3\cdot\log(n)) = \Theta(n^3\cdot\log(n))$.

3.

Merge sort řadí pole, které obsahuje $n^3 + \log(n)$ položek. Asymptotická složitost tohoto řazení je

a) $\Theta(n^3 + \log(n))$
b) $\Theta(n^2\cdot\log(n^3))$
c) $\Theta(n^3\cdot\log(n))$
d) $\Theta(n\cdot\log(n))$
e) $\Theta(n^3 + n^2)$

Řešení Výsledek zdůvodněrte sami. K tomu je třeba zejména ukázat, že funkce log(n) a funkce log(n^3 + log(n)) rostou asymptoticky stejně rychle. Může vám pomoci nerovnost $n^3 < n^3 + \log(n) < n^4$.

4a.
Merge sort, který rekurzivně dělí řazený úsek vždy na poloviny, řadí pole šesti čísel: { 8, 1, 7, 6, 4, 2 }. Situace v aktuálně řazeném poli (ať už to bude původní pole nebo pomocné) bude těsně před provedením posledního slévání (merging) následující:

a) 1 7 8 2 4 6
b) 1 2 4 6 7 8
c) 8 7 6 4 2 1
d) 1 2 4 7 8 6
e) 2 4 6 1 7 8

Řešení Zřejmě před posledním sléváním musí levá polovina pole obsahovat ty prvky, které v ní byly původní, nyní však musí být seřazené. Totéž platí pro pravou polovinu, jediná možná varianta je a).

4b.
Merge sort, který rekurzivně dělí řazený úsek vždy na poloviny, řadí pole šesti čísel: { 9, 5, 8, 7, 2, 0 }. Situace v aktuálně řazeném poli (ať už to bude původní pole nebo pomocné) bude těsně před provedením posledního slévání (merging) následující:

a) 0 2 7 5 8 9
b) 0 2 5 7 8 9
c) 2 5 7 0 8 9
d) 7 8 9 0 2 5
e) 5 8 9 0 2 7

Řešení Zřejmě před posledním sléváním musí levá polovina pole obsahovat ty prvky, které v ní byly původní, nyní však musí být seřazené. Totéž platí pro pravou polovinu, jediná možná varianta je e).

5.
Merge sort řadí pole obsahující prvky 2 3 4 1 (v tomto pořadí). Celkový počet vzájemných porovnání prvků pole při tomto řazení bude

a) 3
b) 4
c) 5
d) 6
e) 8

6.
Merge sort sorts the array containing elements 2 3 4 1 (in this order). The total number of comparisons of elements in this particular case will be:

f) 3
g) 4
h) 5
i) 6
j) 8

7.
Merge sort řadí pole, které obsahuje $n^3 - 5n$ položek. Asymptotická složitost tohoto řazení je

a) $\Theta(n^3 - \log(n))$
b) $\Theta(n^2 \cdot \log(n))$
c) $\Theta(n \cdot \log(n))$
d) $\Theta(n^3 - n)$
e) $\Theta(n^3 \cdot \log(n))$

8.
Merge sort řadí pole, které obsahuje $2n^3 + 3n^2$ položek. Asymptotická složitost tohoto řazení je

e) $\Theta(2n^3 + \log(n))$
f) $\Theta(n^2 \cdot \log(n^3))$
g) $\Theta(n^3 \cdot \log(n))$
h) $\Theta(n \cdot \log(n))$
i) $\Theta(n^3 + n^2)$

9.
Merge sort, který dělí řazený úsek vždy na poloviny, řadí pole šesti čísel: { 2, 9, 4, 8, 1, 6 }.
Situace v aktuálně zpracovaném poli (ať už to bude původní pole nebo pomocné) bude těsně před provedením posledního slévání (merging) následující:

a) 2 4 9 1 6 8
b) 1 2 4 6 8 9
c) 2 9 4 8 1 6
d) 9 8 6 4 2 1
e) 1 9 8 6 4 2

Merge sort je charakteristický tím, že nejprve pole rozdělí na poloviny aniž jakkoli „hýbe“ s daty, pak obě poloviny zvlášť seřadí a jako poslední krok provede operaci nazvanou slévání (merging), jež obě seřazené poloviny sloučí v jedinou seřazenou posloupnost. To znamená, že po posledním slévání bude již pole definitivně seřazené a těsně před posledním sléváním budou seřazeny obě poloviny pole. První polovina pole obsahuje čísla
2, 9, 4, po jejím seřazení to bude 2, 4, 9. Podobně v druhé polovině po jejím seřazení najdeme hodnoty 1, 6, 8. Celkem tak před posledním sléváním bude pole obsahovat posloupnost 2, 9, 4, 1, 6, 8. To je právě varianta a).

10.
Merge sort, který dělí řazený úsek vždy na poloviny, řadí pole šesti čísel: { 3, 6, 1, 7, 2, 5 }.
Situace v aktuálně řazeném poli (ať už to bude původní pole nebo pomocné) bude těsně před provedením posledního slévání (merging) následující:

a) 1 2 3 5 6 7
b) 3 6 1 7 2 5
c) 7 6 5 3 2 1
d) 1 3 6 2 5 7
e) 1 7 6 5 3 2

Merge sort je charakteristický tím, že nejprve pole rozdělí na poloviny aniž jakkoli „hýbe“ s daty, pak obě poloviny zvlášť seřadí a jako poslední krok provede operaci nazvanou slévání (meeting), jež obě seřazené poloviny sloučí v jedinou seřazenou posloupnost. To znamená, že po posledním slévání bude již pole definitivně seřazené a těsně před posledním sléváním budou seřazeny obě poloviny pole. První polovina pole obsahuje čísla
3, 6, 1, po jejím seřazení to bude 1, 3, 6. Podobně v druhé polovině po jejím seřazení najdeme hodnoty 2, 5, 7. Celkem tak před posledním sléváním bude pole obsahovat posloupnost 1, 3, 6, 2, 5, 7. To je právě varianta d).

11.
Implementujte nerekurzivní variantu Merge sortu s využitím vlastního zásobníku. Vaše implementace nemusí usilovat o maximální rychlost, stačí, když dodrží asymptotickou složitost Merge sortu.

Jednotlivé operace zásobníku neimplementujte, předpokládejte, že jsou hotovy, a pouze je vhodně volejte.

Řešení Merge sort řeší svou úlohu pomocí procházení stromu rekurze v pořadí postorder. Je to zřejmé, protože k tomu, aby mohl být zpracován (=seřazen) některý úsek pole, musí být jïž seřazeny obě jeho poloviny, tj řazení muselo proběhnout v potomcích uzlu, který reprezentuje aktuální úsek. Musíme tedy umět zpracovat uzly stromu v pořadí postorder bez rekurze.

Na zásobník si budeme ukládat jednotlivé uzly spolu s informací kolikrát byly již navštíveny. Do zpracování uzlu se dáme až tehdy, když ze zásobníku vyzvedneme uzel s informací že do něho již vstupujeme potřetí (tj přicházíme z jeho již zpracovaného pravého podstromu).
Průchod stromem může proto vypadat např. takto:

```
if (ourTree.root == null) return;
stack.init();
stack.push(ourTree.root, 1);

while (stack.empty() == false) {
   (nodeX, time) = stack.pop();
    if  (isLeaf(nodeX) == true)
         continue;   // Listy ve stromu merge sortu
                     // budou úseky o délce jedna,
                     // ty nijak zpracovávat (= řadit) nebudeme
    if( time == 1) {
          stack.push(nodeX, time+1);          // počet návštěv akt. uzlu
vzrostl
          stack.push(nodeX.left, 1);          // a doleva jdeme poprvé
    }

    if( time == 2) {
          stack.push(nodeX, time+1);          // počet návštěv akt. uzlu
vzrostl
          stack.push(nodeX.right, 1);         // a doprava jdeme poprvé
    }

    if( time == 3)
            zpracuj(nodeX);       // zpracování = sloučení úseku pole
                                  //odpovídajícího uzlu nodeX.
} // end of while
```

Uvedenou kostru algoritmu je jen nutno doplnit:
-- uzel stromu odpovídá řazenému úseku pole, nodeX tedy bude charakterizován a nahrazen dvojicí (horní mez, dolní mez) určující úsek pole.
-- test isLeaf(nodeX) nahradíme testem, zda horní a dolní mez jsou si rovny
-- výpočteme střed řazeného úseku. Levý následník pak bude charakterizován dvojicí čísel (dolní mez, střed), pravý následník dvojicí čísel (střed+1, horní mez)
-- zpracuj (nodeX) nebude nic jiného, než obyčejné slévání obou polovičních úseků (dolní mez, střed) a (střed+1, horní mez) do úseku (dolní mez, horní mez) stejně jako je tomu v rekurzivním Merge sort-u.

Budeme jen potřebovat pomocné pole pro slévání (to v rekurzivní variantě také potřebujeme). Slévání v nejjednodušší variantě sleje oba úseky do pomocného pole a odtud je zkopíruje zpět do řazeného pole.

12.
Merge Sort lze naprogramovat bez rekurze („zdola nahoru“) také tak, že nejprve sloučíme jednotlivé nejkratší možné úseky, pak sloučíme delší úseky atd. Proveďte to.

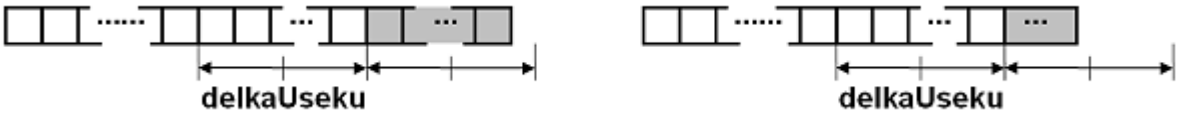
Kdyby pole mělo velikost rovnou mocnině dvou, stačily by nám jen dva jednoduché vnořené cykly, jak to ukazuje následující kód, ve kterém funkce merge(levyKraj, stred, pravyKraj) slévá standardním způsobem levou a pravou polovinu řazeného úseku
[levyKraj, stred] a [stred+1, pravyKraj]:

```
delkaUseku = 2;
while (delkaUseku <= array.length) {
  levyKraj = 0;
  while(levyKraj < array.length-1) {
    pravyKraj = levyKraj+delkaUseku-1;
    stred = PravyKraj-delkaUseku/2;

    merge(levyKraj, stred, pravyKraj);

    levyKraj += delkauseku;
  }
  delkaUseku *= 2;
}
```

V obecném případě ovšem délka pole mocninou nebude a proto poslední aktuálně slévaný úsek v poli může mít kratší délku než všechny ostatní, jak to ukazují následující obrázky.



V případě, že je zbývající úsek (v šedé barvě) delší než polovina délky aktuálně slévaných úseků (levý obrázek), seřadíme jej pomocí jednoho slévání.
V případě, že je zbývající úsek kratší nebo stejně dlouhý jako polovina délky aktuálně slévaných úseků (pravý obrázek), není co slévat, protože úsek již musí být seřazený — po některém z předchozích průchodů s kratší délkou řazených úseků.
Oba uvedené případy jsou zachyceny v následující modifikaci (pseudo)kódu:

```
delkaUseku = 2;
while (delkaUseku <= array.length) {
  levyKraj = 0;
  while(levyKraj < array.length-1) {
    pravyKraj = levyKraj+delkaUseku-1;
    stred = PravyKraj-delkaUseku/2;

    if (PravyKraj <= array.length-1)
        merge(levyKraj, stred, pravyKraj);
     else
        if (stred < array.length-1) {
           pravyKraj = array.length-1;
           merge(levyKraj, stred, pravyKraj);
           }
        else { } // nedelej nic

    levyKraj += delkaUseku;
  }
  delkaUseku *= 2;
}
```

---------- **HEAP** ----------

1.
Jedna z následujících posloupností představuje haldu uloženou v poli. Která?

a) 9 5 4 6 3
b) 5 4 2 3 9
c) 3 8 9 5 6
d) 5 1 8 9 1
e) 1 3 6 5 4

2.
Jedna z následujících posloupností představuje haldu uloženou v poli. Která?

a) 3 2 6 7 4
b) 9 3 1 6 4
c) 2 4 8 7 9
d) 6 3 8 7 3
e) 2 7 4 8 1

3.
Pouze jediná z následujících posloupností představuje haldu uloženou v poli. Označte ji.

a) 2 5 6 4 9
b) 2 6 5 4 9
c) 2 6 5 9 4
d) 2 5 4 9 6
e) 2 9 4 5 6

4.
Only one of the following sequences represents a heap stored in an array. Which one?

f) 2 5 6 4 9
g) 2 6 5 4 9
h) 2 6 5 9 4
i) 2 5 4 9 6
j) 2 9 4 5 6

5.
Pouze jediná z následujících posloupností představuje haldu uloženou v poli. Která?

a) 1 4 5 3 8
b) 1 4 3 8 5
c) 1 5 4 3 8
d) 1 5 4 8 3
e) 1 8 3 4 5

6.
Only one of the following sequences represents a heap stored in an array. Which one?

f) 1 4 5 3 8
g) 1 4 3 8 5
h) 1 5 4 3 8

i) 1 5 4 8 3
j) 1 8 3 4 5

7.
One of the following sequences represents a heap stored in an array. Which one is it?

a) 9 5 4 6 3
b) 5 4 2 3 9
c) 3 8 9 5 6
d) 5 1 8 9 1
e) 1 3 6 5 4

8.
Pouze jediná z následujících posloupností představuje haldu. Která?

a) 39475
b) 34975
c) 43579
d) 35749
e) 45379

9.
Only one of the following sequences represents a heap. Which one?

f) 3 9 4 7 5
g) 3 4 9 7 5
h) 4 3 5 7 9
i) 3 5 7 4 9
j) 4 5 3 7 9

10.
Pouze jediná z následujících posloupností představuje haldu. Která?

a) 29386
b) 32689
c) 26398
d) 29863
e) 36289

11.
Only one of the following sequences represents a heap. Which one?

f) 2 9 3 8 6
g) 3 2 6 8 9
h) 2 6 3 9 8
i) 2 9 8 6 3
j) 3 6 2 8 9

12.
Heap sort řadí pole obsahující hodnoty 4 3 1 2. V první fázi řazení z těchto hodnot vytvoří haldu. Ta bude mít tvar

a) 1 2 4 3
b) 2 1 3 4
c) 3 1 2 4

d) 2 1 3 4
e) 2 4 1 3

13.
Implementujte prioritní frontu. Je to datový typ, který má čtyři operace
Init() -- vytvoří prázdnou prioritní frontu
Empty() -- vrací true, je-li fronta prázdná, jinak vrací false
Insert(prvek) -- vloží prvek do fronty
ExtractMin() -- vrátí prvek s nejmenší hodnotou ve frontě

Běžně se prioritní fronta implementuje jako halda. Operace Insert zařadí vkládaný prvek na jemu příslušné místo v haldě (zachová pravidlo haldy). Operace ExtractMin vyjme prvek z vrcholu haldy a haldu opraví.

---------- HEAP SORT ----------

1.
Heap sort řadí pole obsahující hodnoty 4 3 1 2. V první fázi řazení z těchto hodnot vytvoří haldu. Ta bude mít tvar

f) 1 2 4 3
g) 2 1 3 4
h) 3 1 2 4
i) 2 1 3 4
j) 2 4 1 3

2.
Heap sort sorts the array containing elements 4 3 1 2 (in this order). In the first phase of sorting process, the element are rearranged to form a heap. The particular form of this heap is:

k) 1 2 4 3
l) 2 1 3 4
m) 3 1 2 4
n) 2 1 3 4
o) 2 4 1 3

3.
Heap sort řadí pole obsahující hodnoty 3 4 2 1. V první fázi řazení z těchto hodnot vytvoří haldu. Ta bude mít tvar

a) 2 3 1 4
b) 3 1 2 4
c) 1 3 2 4
d) 2 1 3 4
e) 2 4 1 3

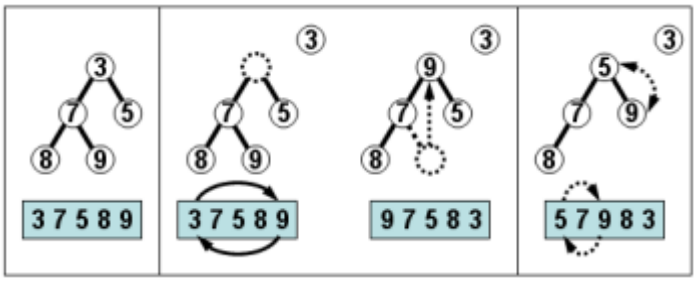
4.
Heap sort sorts the array containing elements 3 4 2 1 (in this order). In the first phase of sorting process the element are rearranged to form a heap. The particular form of this heap is:

f) 2 3 1 4

g) 3 1 2 4
h) 1 3 2 4
i) 2 1 3 4
j) 2 4 1 3

5.
Heap sort řadí pole. Poté, co v první své fázi vytvoří haldu, je v poli uložena posloupnost:
3 7 5 8 9 Algoritmus dále pokračuje v práci. Po zařazení prvního prvku (=3) na jeho definitivní místo, je situace v poli:

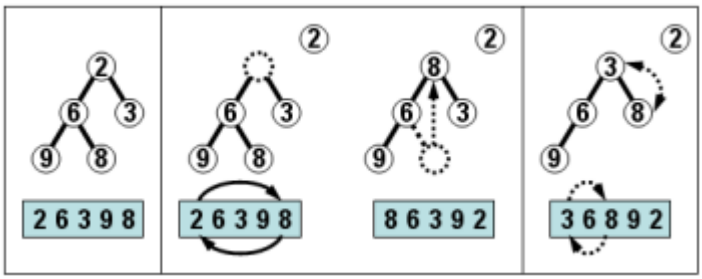
a) 5 7 8 9 3
b) 3 9 8 5 7
c) 7 5 8 9 3
d) 5 7 9 8 3
e) 3 5 7 8 9



Průběh popsané operace snad dostatečně vyjadřují obrázky, tečkované čáry v jednotlivých obrázcích představují stav či akci bezprostředně předchozí.

6.
Heap sort řadí pole. Poté, co v první své fázi vytvoří haldu, je v poli uložena posloupnost:
2 6 3 9 8
Algoritmus dále pokračuje v práci. Po zařazení prvního prvku (= 2) na jeho definitivní místo, je situace v poli:

a) 6 3 8 9 2
b) 2 3 6 9 8
c) 3 6 8 9 2
d) 2 6 3 8 9
e) 3 9 6 8 2



Průběh popsané operace snad dostatečně vyjadřují obrázky, tečkované čáry v jednotlivých obrázcích představují stav či akci bezprostředně předchozí.

7.
Předložíme-li Heap sort-u vzestupně seřazenou posloupnost délky n, bude složitost celého řazení
a) $\Theta(n)$, protože Heap sort vytvoří haldu v čase $\Theta(n)$
b) $\Theta(n^2)$, protože vytvoření haldy bude mít právě tuto složitost
c) $\Theta(n \cdot \log_2(n))$, protože vytvoření haldy bude mít právě tuto složitost
d) $\Theta(n \cdot \log_2(n))$, protože zpracování haldy bude mít právě tuto složitost
e) $\Theta(n)$, protože vytvoření i zpracování haldy bude mít právě tuto složitost

Řešení Tvorba haldy v Heap sortu odpovídá zhruba $n/2$ násobnému volání funkce (procedury, algoritmu...), která opraví „haldu“, jejímž jediným nedostatkem je velikost prvku v kořeni (= jediný nemá vlastnost haldy). Máme-li ovšem dánu vzestupně seřazenou posloupnost, není co kam zařazovat, neb celá posloupnost i každá její souvislá část představuje haldu, takže volání dotyčné funkce nezabere více než konstantní dobu. Na „vytvoření“ haldy v tomto případě padne čas úměrný $konst \cdot n/2$. Celková doba dotyčného řazení je ale $\Theta(n \cdot \log_2(n))$, což je (asymptoticky!) více než $\Theta(n)$ strávených v první fázi, takže delší čas úměrný $n \cdot \log_2(n)$, musí být stráven ve fázi zpracování haldy. Platí varianta d).

8.
Heap sort
a) není stabilní, protože halda (=heap) nemusí být pravidelným stromem
b) není stabilní, protože žádný rychlý algoritmus ($\Theta(n^*\log(n))$) nemůže být stabilní

c) je stabilní, protože halda (=heap) je vyvážený strom
d) je stabilní, protože to zaručuje pravidlo haldy
e) neplatí o něm ani jedno předchozí tvrzení

Řešení Heap sort není stabilní, c) i d) odpadají. b) není pravda, neboť Merge sort má dotyčnou rychlost a stabilní být může (většinou je). Dělení na stabilní a nestabilní a u datových struktur neexistuje (co by to vůbec bylo?), takže ani a) neplatí a zbývá jen e).

9a.

Následující posloupnost představuje haldu o čtyřech prvcích uloženou v poli.

a) 1 3 4 2
b) 1 4 2 3
c) 1 2 4 3
d) 2 3 4 1

Řešení Prvek na 1. pozici musí být menší než prvek na 2. a 3. pozici, možnost d) odpadá. Prvek na 2. pozici musí být menší než prvek na 4. pozici, možnosti a) a b) odpadají. Více podmínek haldy pro 4 prvky nelze formulovat, varianta c) je halda.

9b.

Následující posloupnost představuje haldu o čtyřech prvcích uloženou v poli

a) 1 3 4 2
b) 1 3 2 4
c) 1 4 2 3
d) 2 3 1 4

Řešení Prvek na 1. pozici musí být menší než prvek na 2. a 3. pozici, možnost d) odpadá. Prvek na 2. pozici musí být menší než prvek na 4. pozici, možnosti a) a c) odpadají. Více podmínek haldy pro 4 prvky nelze formulovat, varianta b) je halda.

10a.

Následující posloupnost čísel představuje haldu uloženou v poli. Proveďte první krok fáze řazení v Heap Sortu (třídění haldou), tj.
a) zařaďte nejmenší prvek haldy na jeho definitivní místo v seřazené posloupnosti a
b) opravte zbytek tak, aby opět tvořil haldu.

1 5 2 17 13 24 9 19 23 22

Řešení

a) prohodíme první prvek s posledním:
22 5 2 17 13 24 9 19 23 **1**
b) Prvek 22 necháme „propadnout“ („probublat“) haldou na odpovídající mu místo:
2 5 9 17 13 24 **22** 19 23 1

10b.

Následující posloupnost čísel představuje haldu uloženou v poli. Proveďte první krok fáze řazení v Heap Sortu (třídění haldou), tj.
a) zařaďte nejmenší prvek haldy na jeho definitivní místo v seřazené posloupnosti a
b) opravte zbytek tak, aby opět tvořil haldu.

4 8 11 12 9 18 13 17 21 25

Řešení

a) prohodíme první prvek s posledním:
25 8 11 12 9 18 13 17 21 **4**
b) Prvek 25 necháme „propadnout“ („probublat“) haldou na odpovídající mu místo:
8 9 11 12 **25** 18 13 17 21 **4**

11.

Zadanou posloupnost v poli seřaďte ručně pomocí heap sortu. Registrujte stav v poli po každém kroku. Nejprve po každé opravě částečné haldy od prostředku pole směrem k začátku, tj. při první fázi. Potom také po každé opravě haldy a přenesení prvku z vrcholu haldy na za konec haldy.

23	29	27	4	28	17	1	24	6	30	19

Řešení Pro snažší orientaci připíšeme indexy nad jednotlivé prvky. Podtržítko ukazuje, kde v daném kroku proběhla změna, podsvícené prvky jsou definitivně zařazeny.

Výchozí pole:

1	2	3	4	5	6	7	8	9	10	11
23	29	27	4	28	17	1	24	6	30	19

Tvorba haldy

1	2	3	4	5	6	7	8	9	10	11
23	29	27	4	19	17	1	24	6	30	28

1	2	3	4	5	6	7	8	9	10	11
23	29	27	4	19	17	1	24	6	30	28

1	2	3	4	5	6	7	8	9	10	11
23	29	1	4	19	17	27	24	6	30	28

1	2	3	4	5	6	7	8	9	10	11
23	4	1	6	19	17	27	24	29	30	28

1	2	3	4	5	6	7	8	9	10	11
1	4	17	6	19	23	27	24	29	30	28

Halda je hotova, dále řadíme opakovaným výběrem nejmenšího prvku z haldy.

1	2	3	4	5	6	7	8	9	10	11
4	6	17	24	19	23	27	28	29	30	1

1	2	3	4	5	6	7	8	9	10	11
6	19	17	24	29	23	27	28	30	4	1

1	2	3	4	5	6	7	8	9	10	11
17	19	23	24	29	30	27	28	6	4	1

1	2	3	4	5	6	7	8	9	10	11
19	24	23	28	29	30	27	17	6	4	1

1	2	3	4	5	6	7	8	9	10	11
23	24	27	28	29	30	19	17	6	4	1

1	2	3	4	5	6	7	8	9	10	11
24	29	27	28	30	23	19	17	6	4	1

1	2	3	4	5	6	7	8	9	10	11
27	29	30	28	24	23	19	17	6	4	1

1	2	3	4	5	6	7	8	9	10	11
28	29	30	27	24	23	19	17	6	4	1

1	2	3	4	5	6	7	8	9	10	11
29	30	28	27	24	23	19	17	6	4	1

1	2	3	4	5	6	7	8	9	10	11
29	30	28	27	24	23	19	17	6	4	1

12.
Vysvětlete, jak je nutno modifikovat Heap sort, aby po jeho skončení pole obsahovalo prvky seřazené vzestupně. Algoritmus musí být stejně rychlý, nejen rekurzivně, a nesmí používat žádné další datové struktury nebo proměnné.
Řešení Pravidlo haldy je nutno obrátit, následníkem každého prvku bude vždy prvek s menší nebo stejnou hodotou. Na vrcholu haldy bude prvek s maximální hodnotou. Jinak se v algoritmu nezmění nic, všechny úpravy haldy budou probíhat s ohledem na nové pravidlo haldy zcela analogicky původní variantě.

---------- **RADIX SORT** ----------

1.
We perform Radix sort on an array of *n* strings, the length of each string is *k*. Asymptotic complexity of this process is

a) $\Theta(k^2)$
b) $\Theta(n^2)$
c) $\Theta(k \cdot n)$
d) $\Theta(n \cdot \log(n))$
e) $\Theta(k \cdot n \cdot \log(n))$

2a.
Následující posloupnost řetězců je nutno seřadit pomocí Radix Sortu (přihrádkového řazení). Proveďte první průchod (z celkových čtyř průchodů) algoritmu danými daty a napište, jak budou po tomto prvním průchodu seřazena.

IIYI PIYY YIII YPPP YYYI PYPP PIPI PPYI

Řešení K dispozici budou tři „přihrádky“ odpovídající znakům I, P a Y. První průchod pracuje s posledním symbolem v každém řetězci, takže obsah „přihrádek“ bude:
I : IIYI YIII YYYI PIPI PPYI
P: YPPP PYPP
Y: PIYY

2b.
Následující posloupnost řetězců je nutno seřadit pomocí Radix Sortu (přihrádkového řazení). Proveďte první průchod (z celkových čtyř průchodů) algoritmu danými daty a napište, jak budou po tomto prvním průchodu seřazena.
GKKG KEEG GKGG KGKK KGEE KEEG EGEE GEEG

Řešení K dispozici budou tři „přihrádky“ odpovídající znakům E, G a K. První průchod pracuje s posledním symbolem v každém řetězci, takže obsah „přihrádek“ bude:
E: KGEE EGEE
G: GKKG KEEG GKGG KEEG GEEG
K : KGKK

3.

Pole A obsahuje téměř seřazené řetězce (např. z 99% seřazené), pole B obsahuje řetězce stejné délky, ale zcela neseřazené. Radix sort seřadí

a) A asymptoticky rychleji než B
b) B asymptoticky rychleji než C
c) A asymptoticky stejně rychle jako B, ale použije více paměti pro řazení A
d) A asymptoticky stejně rychle jako B, ale použije více paměti pro řazení B
e) A asymptoticky stejně rychle jako B a použití paměti bude stejné

Řešení Nahlédnutí do kódu nebo do popisu algoritmu dává na srozuměnou, že může platit jedině varianta e), na provádění dobu provádění algoritmu nemají konkrétní hodnoty ve vstupním poli žádný vliv, stejně tak využití paměti je pevně dáno bez ohledu na pořadí dat.

4.

V určitém okamžiku průběhu Radix sortu dostaneme k dispozici pomocná pole z1, k1, d1 (značení podle přednášky, pole se indexují od 0). Máme určit, podle kterého znaku právě řazení probíhá. To určit

a) lze, index tohoto znaku je uložen v k1[0]
b) lze, index tohoto znaku je uložen v d1[0]
c) lze, index tohoto znaku je uložen v z1[0]
d) nelze, protože algoritmus tuto informaci v d1 soustavně přepisuje
e) nelze, pole z1, k1, d1 tuto informaci neregistrují

Řešení Zřejmě platí e), algoritmus jednoho průchodu polem podle libovolného znaku je pevně dán, nezávisí na tom, o kolikátý znak se jedná, a také se to v pomocných polích nikde neregistruje.

5.

V určitém okamžiku průběhu Radix sortu dostaneme k dispozici pomocná pole z1, k1, d1 (značení podle přednášky). Máme určit, zda v tomto okamžiku již byl dokončen jeden z průchodů algoritmu vstupním polem (nezáleží na tom, který) nebo zda je nutno číst ještě nějaké další prvky, než se aktuální průchod ukončí. To určit

a) nelze, protože neznáme pořadí prvků ve vstupním poli
b) nelze, protože některé prvky v poli d1 mohou být stejné
c) nelze, protože některé prvky v polích z1, k1 mohou být stejné
d) lze sečtením délek všech seznamů uložených v poli d1
e) lze sečtením rozdílů odpovídajících si prvků v polích k1 a z1

Řešení Během průchodu se postupně seznamy v poli d1 naplňují a na konci průchodu musí být každý prvek v některém seznamu. Správná odpověď je d), příslušný součet musí být roven velikosti vstupního pole, tedy velikosti pole d1.

6a.

Radix sort řadí pole řetězců {“dda”, “bab”, “ddc”, “aaa”, “bcd”, “dbc”, “bbb”, “add”, “ccd”, “dab”, “bbc” }. napište, jak budou zaplněna jednotlivá pomocná pole (z, k, d, podle přednášky) po prvním průchodu algoritmu tj, po seřazení podle posledního znaku.

Řešení

z	a	b	c	d	k	a	b	c	d	d	1	2	3	4	5	6	7	8	9	10	11
	1	2	3	5		4	10	11	9		4	7	6	0	8	11	10	9	0	0	0

6b.

Radix sort řadí pole řetězců {"dad", "caa", "cad", "aac", "bca", "dbc", "bbd", "ddc", "cda", "dac", "bbc" }. napište, jak budou zaplněna jednotlivá pomocná pole (z, k, d, podle přednášky) po prvním průchodu algoritmu tj, po seřazení podle posledního znaku.

Řešení

z	a	b	c	d	k	a	b	c	d	d	1	2	3	4	5	6	7	8	9	10	11
	2	0	4	1		9	0	11	7		3	5	7	6	9	8	0	10	0	11	0

7a.
Radix sort řadí pole řetězců. V aktuálním okamžiku provádí průchod podle 3 znaku od konce a ještě jej nedokončil. Obsah jednotlivých pomocných polí je uvedený (značení odpovídá přednášce):

z1	a	b	c	k1	a	b	c	d1	1	2	3	4	5	6	7	8	9	10
	3	10	5		1	2	5		0	0	8	0	0	0	2	1	0	7

Napište, jak se obsah bude postupně měnit po zpracování každého z dalších řetězců:
“abbac”(9), ”aaaaa”(4), ”bbbac”(6), čísla v závorkách uvádějí pozici řetězce ve vstupním poli.

Řešení (změny jsou podtržené):
Po zpracování “abbac”(9):

z1	a	b	c	k1	a	b	c	d1	1	2	3	4	5	6	7	8	9	10
	3	10	5		1	6	5		0	6	8	0	0	0	2	1	0	7

Po zpracování ”aaaaa”(4):

z1	a	b	c	k1	a	b	c	d1	1	2	3	4	5	6	7	8	9	10
	3	10	5		4	6	5		4	6	8	0	0	0	2	1	0	7

Po zpracování ”bbbac”(6):

z1	a	b	c	k1	a	b	c	d1	1	2	3	4	5	6	7	8	9	10
	3	10	5		4	9	5		4	6	8	0	0	9	2	1	0	7

7b.
Radix sort řadí pole řetězců. V aktuálním okamžiku provádí průchod podle 4 znaku od konce a ještě jej nedokončil. Obsah jednotlivých pomocných polí je uvedený (značení odpovídá přednášce):

z1	a	b	c	k1	a	b	c	d1	1	2	3	4	5	6	7	8	9	10	11
	8	0	3		1	0	2		0	0	2	0	0	0	9	7	1	0	0

Napište, jak se obsah bude postupně měnit po zpracování každého z dalších řetězců:
“baaab”(11), ”ccaba”(4), ”ababa”(6), ”babab”(5), ”bbbbc”(10), čísla v závorkách uvádějí pozici řetězce ve vstupním poli.

Řešení (změny jsou podtržené):
Po zpracování “baaab”(11):

z1	a	b	c	k1	a	b	c	d1	1	2	3	4	5	6	7	8	9	10	11
	8	0	3		11	0	2		11	0	2	0	0	0	9	7	1	0	0

Po zpracování ”ccaba”(4) :

z1	a	b	c	k1	a	b	c	d1	1	2	3	4	5	6	7	8	9	10	11
	8	0	3		11	0	4		11	4	2	0	0	0	9	7	1	0	0

Po zpracování ”ababa”(6):

z1	a	b	c	k1	a	b	c	d1	1	2	3	4	5	6	7	8	9	10	11
	8	6	3		11	6	4		11	4	2	0	0	0	9	7	1	0	0

Po zpracování ”babab”(5):

z1	a	b	c	k1	a	b	c	d1	1	2	3	4	5	6	7	8	9	10	11
	8	6	3		5	6	4		11	4	2	0	0	0	9	7	1	0	5

Po zpracování ”bbbbc”(10):

z1	a	b	c	k1	a	b	c	d1	1	2	3	4	5	6	7	8	9	10	11
	8	6	3		5	10	4		11	4	2	0	0	10	9	7	1	0	5

8.
Profesor Vylepšil je toho mínění, že by se Radix sort dal využít i pro řazení kladných celých čísel. Navrhuje následující postup. Každé číslo celé bude interpretovat jako posloupnost čtyř znaků reprezentujích jeho zápis v soustavě o základu 256. Jinými slovy, hodnotu každého bytu tohoto čísla bude považovat za samostatný znak. Například číslo 1697043971 se rovná $101 \cdot 256^3 + 38 \cdot 256^2 + 214 \cdot 256^1 + 3 \cdot 256^0$,
takže jej lze v tomto návrhu zapsat jako <101><38><214><3>, kde každé číslo společně s jeho závorkou interpretujeme jako jeden samostatný symbol.
Uvažte, jaké změny je potřeba učinit v kódu Radix sortu, aby bylo možno tento návrh impelementovat a posuďte, nakolik je relevantní pro řazení různě velkých polí celých čísel.

Komentář Změny nejsou veliké, místo řetězců máme na vstupu celá čísla. Většinou v kódu pracujeme s interní datovou reprezentací jednotlivých seznamů vyžadovaných Radix sortem. Ke každému prvku vstupního pole přistupujeme v každém průchodu Radix sortu jen jednou a to jen k jedinému znaku. Zřejmě bude zapotřebí změnit pouze tuto část kódu, abychom mohli zařadit aktuální prvek pole do odpovídajícího seznamu. Nepoužijeme-li bitové operace posunu, postačí nám celočíselné dělení a zbytek po celočíselném dělení. Ke *k*-tému nejmenšímu bytu čísla *n* přistoupíme takto:
byte = (*n* / (256^*k*)) % 256; // pokud nejnižší byte ma index 0.
Když bude řazené pole dosti velké, mohlo by být toto řazení velmi rychlé, rychlejší než řazení s „logaritmickou“ složitostí, počet průchodů polem je roven 4. Máme však dvě pomocná pole následníků, která mají stejnou délku jako vstupní pole, při zpracování jednoho znaku musíme provést alespoň 4 přístupy do polí, zřejmě tedy režijní úkony zaberou dosti času a výhoda začne být patrná až u velkých souborů dat.
V autorově implementaci je toto řazení až o dvacet procent rychlejší než Merge Sort pro pole velikosti alespoň cca 100 000, jsou-li data téměř uspořádaná, je rychlejší i než Quick sort. Pro ještě větší velikosti pole lze uvažovat o rozdělení čísel na pouze dva „znaky“ s použitím soustavy o základu 2^{16}.
Výhodou Radix sortu je také to, že jeho časové nároky nezávisí na konkrétních hodnotách dat, a tedy doba běhu je snáze předvídatelná.

9.
Profesor Omezil se zadíval na datové struktury Radix sortu a prohlásil, že ukazatelé na začátky seznamů (pole Z na přednášce) jsou zbytečné. Navrhuje, aby každý seznam byl cyklický, to jest, aby poslední prvek seznamu odkazoval na první prvek. První prvek pak bude vždy dostupný jako následník posledního prvku. Ukazatele na poslední prvky seznamů je však vhodno v paměti udržovat díky tomu, že se na konec seznamů neustále průběžně přidávají prvky a seznamy se prodlužují.
My si o návrhu profesora Omezila myslíme, že pokud bude fungovat, omezí sice poněkud velikost použité paměti, ale zato zvýší čas provádění kódu, protože referenci na začátek každého seznamu bude nutno průběžně v poli následníků přepisovat, zatímco v původní podobě se při každém průchodu daty zaznamenávají jen jednou.
Implementujte navrhované změny, rozhodněte zda budou funkční a pokud ano, za domácí úkol porovnejte praktické rychlosti obou variant.

Řešení Popis změn je celkem jasný, zbývá je napsat. Fungovat zcela jistě budou, protože žádná podstatná informace se ze seznamů neztrácí, v celém kódu je třeba drobné úpravy na několika řádcích. Nejprve uvádíme původní kód z přednášky poté modifikovaný kód s vyznačenými změnami. Všechna pole jsou v obou variantách indexována od 0. Porovnání rychlostí zůstává za domácí úkol.

```
void step1 (String [] a, int [] z, int [] k, int [] d) {
  int pos = a[0].length()-1;
  int c1;
  for (int i = 0; i < k.length; i++)
    z[i] = k[i] = -1; // empty
  for (int i = 0; i < a.length; i++) {
    c1 = (int) a[i].charAt(pos)-97; // suppose ascii
    if (z[c1] == -1)
       k[c1]= z[c1] = i;
    else {
       d[k[c1]] = i;
       k[c1] = i;
    }
  }
}


void stepk(String [] a, int pos,
```

```
          int [] z, int [] k, int [] d, int [] z1, int [] k1, int [] d1) {
  int j, c2;
  for (int i = 0; i < k.length; i++)
    z1[i] = k1[i] = -1; // empty
  for (int i = 0; i < k.length; i++)
    if (z[i] != -1) {
      j = z[i];
      while (true) {
        c2 = (int) a[j].charAt(pos)-97; // suppose ascii
        if (z1[c2] == -1)  // new list
           k1[c2] = z1[c2] = j;
        else {
           d1[k1[c2]] = j;
           k1[c2] = j;
        }
        if (j == k[i]) break;
        j = d[j];
      }
  }
}


void step1_Omezil(String [] a, int [] k, int [] d) {
  int pos = a[0].length()-1;
  int c1;
  for (int i = 0; i < k.length; i++)
    k[i] = -1; // empty
  for (int i = 0; i < a.length; i++) {
    c1 = (int) a[i].charAt(pos)-97;
    if (k[c1] == -1)
      k[c1] = d[i]= i;
    else {
      d[i] = d[k[c1]];
      d[k[c1]] = i;
      k[c1] = i;
    }
  }
}


void stepk_Omezil(String[] a, int pos, int[] k, int[] d, int[] k1, int[] d1)
{
  int j, c2;
  for (int i = 0; i < k.length; i++)
    k1[i] = -1; // empty
  for (int i = 0; i < k.length; i++)
    if (k[i] != -1) {
      j = d[k[i]];  // first elem  in orig list
      while (true) {
        c2 = (int) a[j].charAt(pos)-97;
        if (k1[c2] == -1)  // new list
          k1[c2] = d1[j] = j;
        else {
          d1[j] = d1[k1[c2]];   // added
          d1[k1[c2]] = j;
          k1[c2] = j;
        }
```

```
        if (j == k[i]) break;
          j = d[j];
      }
    }
}
```

---------- **COUNTING SORT** ----------

1
Uvažujme pole obsahující 1000 navzájem různých čísel v pohyblivé řádové čárce. Counting sort se pro toto pole

a) hodí, protože toto řazení má lineární složitost
b) hodí, protože toto řazení má sublineární složitost
c) hodí, protože čísla lze převést na řetězce
d) nehodí, protože čísla v poli nemusí být celá
e) nehodí, protože čísla jsou navzájem různá

2.
Counting sortem řadíme pole čísel:
8 14 14 7 11 11 6 3 12 11 2 12 14 9 8
Jaký bude počáteční obsah pole četností pro tato data?

Řešení

2	3	4	5	6	7	8	9	10	11	12	13	14
1	**1**	**0**	**0**	**1**	**1**	**2**	**1**	**0**	**3**	**2**	**0**	**3**

3a.
Counting sortem řadíme pole čísel:
50 88 87 87 93 87 23 53 70 89 53 62.
Jaký bude nejmenší a největší index v poli četností?

a) 50 62
b) 0 62
c) 0 93
d) 23 93
e) 50 93

3b.
Counting sortem řadíme pole čísel:
56 54 20 13 44 75 84 39 31 34 68.
Jaký bude nejmenší a největší index v poli četností?

a) 20 56
b) 20 84
c) 1 56
d) 56 84
e) 56 68

4a.

Řadíme 14 celých čísel. Těsně předtím, než se začne plnit výstupní pole v Counting sortu, je obsah původního pole četností následující (slabě psaná čísla jsou indexy):

10	11	12	13	14	15	16	17	18	19
1	**3**	**4**	**8**	**10**	**12**	**12**	**12**	**13**	**14**

Napište, jak vypadá výsledné uspořádané pole, za předpokladu, že všechna pole začínají indexem 1.

Řešení

10 11 11 12 13 13 13 13 14 14 15 15 18 19

4b.

Řadíme 15 celých čísel. Těsně předtím, než se začne plnit výstupní pole v Counting sortu, je obsah původního pole četností následující (slabě psaná čísla jsou indexy):

10	11	12	13	14	15	16	17	18	19
0	**1**	**2**	**3**	**7**	**9**	**9**	**13**	**14**	**15**

Napište, jak vypadá výsledné uspořádané pole, za předpokladu, že všechna pole začínají indexem 1.

Řešení

11 12 13 14 14 14 14 15 15 17 17 17 17 18 19

5.

Na začátku Counting sortu je v poli četností uložena právě četnost výskytu každé hodnoty ve vstupním poli. V průběhu řazení se obsah pole četností průběžně mění. Rozhodněte, zda je možno po dokončení celého řazení rekonstruovat z modifikovaného pole četností původní (a ovšem i výsledné) četnosti jednotlivých řazených hodnot.

Řešení Je to téměř možné. Po skončení řazení obsahuje pole četností indexy posledních prvků jednotlivých úseků stejných prvků ve výsledném seřazeném poli. Označme dvě různé bezprostředně po sobě následující hodnoty ve výsledném poli symboly p a q, samotné pole četností označme c. Potom platí, že $c[q] - c[p]$ je právě počet prvků vstupního (i výstupního) pole s hodnotou p. Platí dokonce $c[q] - c[p] = c[p+1] - c[p]$. Potíž je pouze s největší řazenou hodnotou. Označme ji opět p. V poli c je p indexem posledního prvku, takže hodnota potřebného výrazu $c[q] - c[p]$ nebo $c[p+1] - c[p]$ není k dispozici. Prvky o hodnotě p zaplňují výstupní pole od pozice $c[p] - 1$ až do konce, takže pokud známe celkový počet řazených prvků n (tj. délku vstupního nebi výstupního pole), víme, že počet prvků s hodnotou p je roven $n - c[p]$, za přepokladu, že pole jsou indexována od 0. Pokud bychom uvažovali indexaci vstupního a výstupního pole od 1 nebo jiného čísla, byl by postup úvah analogický, přenecháváme to čtenáři jako jednoduché cvičení.

6.

Níže je uveden jednoduchý kód funkce, která implementuje Counting sort, jejím vstupem je neseřazené pole *a*, jejím výstupem je seřazené pole *b*. Obě pole jsou deklarována mimo funkci. Funkce ale musí používat pole četností, které sama deklaruje. Ve stávající implementaci toto pole začíná od nulového indexu a končí na indexu rovném maximální hodnotě v řazeném poli. To je paměťově nevýhodné řešení, protože počáteční část pole četností se nevyužije, pokud je minimální hodnota v řazeném poli větší než nula. Dále toto řešení předpokládá, že všechny řazené hodnoty jsou nezáporné, což je ještě vážnější omezení.

Modifikujte daný kód tak, aby velikost pole četností byla minimální, to jest rovna rozdílu maximální a minimální hodnoty v řazeném poli zvětšené o 1. Zároveň funkce musí být schopna zpracovávat i záporná čísla ve vstupním poli.

```
void countSort0(int a[], int [] b) {
 if (a.length != b.length) return;
 int min = a[0];
 int max = a[0];
 int val;

 // get min and max value in input array
```

```
 for (int i = a.length%2; i < a.length; i += 2 )
   if (a[i] < a[i+1]) {
     if (a[i] < min) min = a[i];
     if (a[i+1] > max) max = a[i+1];
   }
   else {
     if (a[i+1] < min) min = a[i+1];
     if (a[i] > max) max = a[i];
   }
 // frequency array
 int [] freq = new int[max+1];
 for (int i = 0; i < freq.length; i++ )
    freq[i] = 0;        // unnecessary in java

 // fill frequencies
 for (int i = 0; i < a.length; i++ )
   freq[a[i]]++ ;
 for (int i = min+1; i <= max; i++ )
   freq[i] += freq[i-1];

 // sort
 for (int i = a.length-1; i >= 0; i-- ) {
   val = a[i];
   freq[val]--;
   b[freq[val]] = val;
 }
}
```

Řešení Zřejmě se pole četností freq musí chovat tak, jako kdyby v hodnoty ve vstupním poli měly stejný rozsah, ale všechny byly zmenšeny (zvětšeny) o konstatntu tak, aby nejmenší hodnota vstupního pole byla 0.
Příslušná konstatnta je ovšem rovna právě původní minimální hodnotě ve vstupním poli. Při manipulaci s polem frekvencí musíme každou hodnotu vstupního pole snížít o tuto konstantu. Řádky se změnami jsou zvýrazněny.

```
void countSort(int a[], int [] b) {
 if (a.length != b.length) return;
 int min = a[0];
 int max = a[0];
 int val;

 // get min and max of array a
 for (int i = a.length%2; i < a.length; i += 2 )
   if (a[i] < a[i+1]) {
    if (a[i] < min) min = a[i];
    if (a[i+1] > max) max = a[i+1];
   }
   else {
    if (a[i+1] < min) min = a[i+1];
    if (a[i] > max) max = a[i];
   }

 // frequency array
 int [] freq = new int[max-min+1];
 for (int i = 0; i < freq.length; i++ )
```

```
    freq[i] = 0;        // unnecessary in java

  // fill frequencies
 for (int i = 0; i < a.length; i++ )
   freq[a[i]-min]++ ;
 for (int i = 1; i <= freq.length-1; i++ )
   freq[i] += freq[i-1];
 lista(freq);

 // sort
 for (int i = a.length-1; i>= 0; i-- ) {
   val = a[i]-min;              //
   freq[val]--;
   b[freq[val]] = a[i];
 }
}
```

7.

Je dána posloupnost A={ a_1, a_2,..., a_n } uložená v poli *p*, pro níž platí:

- 0 <= a_i <= 20 000 pro všechna *i*
- žádná z hodnot v posloupnosti se neopakuje více než 50-krát
- 10 000 <= n <= 20 000

Napište algoritmus, který tuto posloupnost uspořádá v lineárním čase, přičemž použije maximálně tři průchody posloupností a využije pomocnou paměť o velikosti O(n). Uspořádaná posloupnost bude nakonec uložena v původním poli *p*. Úplná deklarace datových stuktur (2 body), algoritmus

Porovnání řazení

1.

Následující dvojici řazení je možno implementovat tak, aby byla stabilní

a) Heap sort a Insertion sort
b) Selection sort a Quick sort
c) Insertion sort a Merge sort
d) Heap sort a Merge sort
e) Radix sort a Quick sort

Řešení K nestabilním řazením patří z uvedených jen Quick a Heap Sort. Ty se objevují ve všech variantách kromě c).

2.

Poskytneme-li Quick Sort-u a Merge sort-u stejná data k seřazení, platí

a) Quick sort bude vždy asymptoticky rychlejší než Merge Sort
b) Merge sort bude vždy asymptoticky rychlejší než Quick Sort
c) někdy může být Quick sort asymptoticky rychlejší než Merge Sort
d) někdy může být Merge sort asymptoticky rychlejší než Quick Sort
e) oba algoritmy budou vždy asymptoticky stejně rychlé

Řešení Asymptotická složitost Quick sortu je $O(n^2)$ a ono „n^2“ někdy může nastat. U Merge sortu je to $\Theta(n \cdot \log_2(n))$, takže první a poslední možnost odpadá. Quick Sort pracuje většinou v čase úměrném $n \cdot \log_2(n)$, tedy podobně rychle jako Merge sort, takže druhá možnost odpadá. Také je ale je asymptotická rychlost Quick sortu rovna $\Omega(n \cdot \log_2(n))$, tj Quick Sort nebude *nikdy* asymptoticky rychlejší

než „$n \cdot \log_2(n)$“, tedy nebude asymptoticky rychlejší než Merge sort, sbohem třetí možnosti. V případě, že Quick Sort opravdu selže a poběží v čase úměrném n^2, Merge sort neselže a bude tedy asymptoticky rychlejší, platí d).

Poznámka:

Občas si někdo myslí, že $\Theta(n \cdot \log_2(n))$ popisuje něco jako průměrnou složitost, tj. takovou, se kterou se většinou v praxi setkáme. Není to tak, $\Theta(n \cdot \log_2(n))$ indikuje víc, indikuje, že *pokaždé* bude běh algoritmu úměrný $n \cdot \log_2(n)$, i kdyby trakaře padaly. Kdosi při zkoušce vážně tvrdil, že dokáže vhodnou metodou výběru pivota stlačit složitost Quick sortu na $\Theta(n)$. No nazdar!

- ○

3.

Společná vlastnost Insert sort-u, Select sort-u a Bubble sort-u je následující:

a) všechny tyto algoritmy jsou vždy stabilní
b) všechny tyto algoritmy jsou vždy nestabilní
c) všechny tyto algoritmy mají složitost $O(n \cdot \log_2(n))$
d) všechny tyto algoritmy mají složitost $O(n^2)$
e) všechny tyto algoritmy mají složitost $\Theta(n^2)$

Řešení První z uvedených řazení — Insert sort — lze pouhou záměnou ostré a neostré nerovnosti v testu vnitřního cyklu učinit stabilním či nestabilním. Tvrzení variant a) a b) o stabilitě/nestabilitě za všech okolností tak patrně neplatí. Všechny zmíněné algoritmy v nejhorším případě řadí v čase úměrném n^2, funkce $n \cdot \log_2(n)$ roste asymptoticky pomaleji než funkce n^2, tudíž varianta c) rovněž neplatí. Insert sort může řadit v čase úměrném n, má-li vhodná data, takže tvrdit o něm, jako ve variantě e), že pokaždé řadí v čase úměrném n^2, není korektní. Zbývá tak jen správná varianta d).