1.
Následující dvojici řazení je možno implementovat tak, aby byla stabilní

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

Řešení
K nestabilním řazením náleží z vyjmenovaných 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.
Poznámka:
Někteří z dálkařů si 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.
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).

4.
Heap sort

a) není stabilní, protože halda (=heap) není stabilní datová struktura
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).
Poznámka: Zde bychom ocenili zpětnou vazbu, proč se množství respondentů (35 !!) zdála „nestabilní datová struktura“ tak přitažlivá.

5.
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 $\Theta(n*\log(n))$
d) je rychlý ($\Theta(n*\log(n))$) 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).

6.
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.

7.
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.

8.
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

9.
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**

10.
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 $\Theta(n \cdot \log(n))$.

11.
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“ označené symboly 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

12.
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
V jednom okně editoru si otevřte kód algoritmu uvedený níže (i na přednášce!) a ve druhém sledujte tento text (máte-li toho zapotřebí :-)).
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).

Poznámka.
Někteří respondenti byli přesvědčeni o nejednoznačnosti nabízených variant odpovědí. Prosíme proto každého, kdo postupoval podle odlišné varianty Quick sortu, než která byla uvedena v přednášce, a vyšla mu jiná varianta, aby se přihlásil, rádi mu jeho dosud nepříznivou klasifikaci zlepšíme.

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  }
```

13.
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
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).

14.
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
o

15.
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“.

16.
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 koikrát 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.

17.
Merge Sort lze naprogramovat bez rekurze („zdola nahoru“) také tak, že nejprve sloučíme jednotlivé nejkratší možné úseky, pak sloučíme delší úseky atd. Proveďte to.

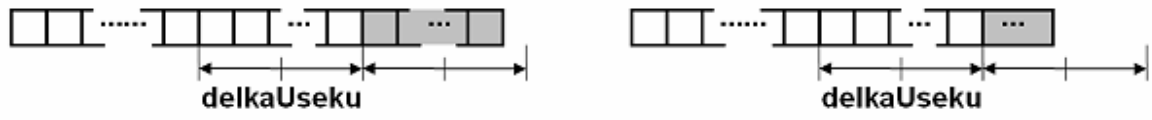
Řešení
Kdyby pole mělo velikost rovnou mocnině dvou, stačily by nám jen dva jednoduché vnořené cykly, jak to ukazuje následující kód, ve kterém funkce merge(levyKraj, stred, pravyKraj) slévá standardním způsobem levou a pravou polovinu řazeného úseku [levyKraj, stred] a [stred+1, pravyKraj]:

```
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;
}
```