1 Metody řazení ve vnitřní a vnější paměti

Řazení: Nechť je dána množina M = {m_1,m_2,…,m_n} a nechť se každý prvek skládá ze dvou složek Mj = [kj,hj], j= 1,2,..,n, kde k_j je klíč prvku, h_j je hodnota prvku. Nechť klíče k_j jsou z jistého univerza U, na němž je definována relace úplného uspořádání ≤. Řazením rozumíme vytvoření takové n-tice (permutace) prvků množiny M; $M_{\leq}$ = [m_{i1},m_{i2},…,m_{in}], že $k_{i1} \leq k_{i2} \leq \ldots k_{in}$. N-tici $M_{\leq}$ nazveme seřazenou podle klíče k_j a uspořádání ≤.

Stabilní řazení:
- když budou klíče dvou prvků shodné, tak prvek, který přišel na vstup později, musíme zařadit za prvek dřívější.

Třídění:

Nechť $M = \{ m_j \}$ je množina dvojic $m_j = [k_j , h_j]$, kde k_j je klíč a h_j hodnota. ≡ je relace ekvivalence na univerzu U hodnot klíčů k_j.

Pak tříděním rozumíme rozklad množiny M na podmnožiny M_t prvků ekvivalentních podle klíče k_i.

Platí $M = \cup_t M_t$, $M_p \cap M_q = \phi$ pro všechna $p \neq q$

$M_t = \{ m_1^t , m_2^t , \ldots m_k^t \}$, $k_i^t \equiv k_j^t$, $i,j \in 1..k$

1.1 Výběrem – Select Sort (stabilní)

1. z množiny vyberu nejmenší / největší prvek
2. vyměním ho s prvním / posledním
3. množinu zmenším o první / poslední prvek
4. pokračuji 1. dokud není množina prázdná

Počet porovnání C(n) = (n^2 – n) / 2 => O(n) = O(n^2)

Počáteční hodnoty klíčů :	10	12	4	9	18	5
i = 6	10	12	4	9	5	**18**
i = 5	10	5	4	9	**12**	18
i = 4	9	5	4	**10**	12	18
i = 3	4	5	**9**	10	12	18
i = 2	4	**5**	9	10	12	18

1.2 Vkládáním – Insert-sort (stabilní)

Vychází z myšlenky, že v každém kroku vloží zkoumaný prvek na správnou pozici v seřazeném poli. Mějme pole o n prvcích a zkoumaný prvek *i*, 1< *i* <= *n*. Všechny prvky v poli do *i* jsou již srovnány. O prvkách od *i* nic nevíme. Pro vkládaný prvek musíme udělat v poli místo, čili od jeho pozice až do konce seřazeného pole budeme prvky přesouvat, abychom mu uvolnili místo. (to se mi zdá dost dementní – je to dobrý leda když je pole již od počátku skoro seřazené)

Předpokládám, že mám A[n] – neseřazené pole, které chci řadit:

1. i = 2	inicializace
2. j = i–1	
3. temp = A[i]	vezmi prvek na pozici *i*
4. dokud temp < A[j] , A[j+1] = A[j] ; j--	dokud nenalezneš místo pro prvek, tak přesouvej prvky v poli od *i* k začátku (již seřazeného pole) o 1 dál
5. A[j+1] = temp	vlož prvek na správnou pozici
6. i++ a pokračuj 2. až do i=n	zvětši *i* a pokračuj dokud nedojdeš s i do konce

Počáteční hodnoty klíčů :		10	12	4	9	18	5
i = 2	A[0] = 12	10	12	4	9	18	5
i = 3	A[0] = 4	4	10	12	9	18	5
i = 4	A[0] = 9	4	9	10	12	18	5
i = 5	A[0] = 18	4	9	10	12	18	5
i = 6	A[0] = 5	4	5	9	10	12	18

1.3 Zaměňováním – Bubble-sort (stabilní)

```
for( i = 1; i < n ; i++)
    //for( j = 1 ; j < n ; j++)       // Tohle by napsal jen amatér ☺
    for( j = 1 ; j < n - i; j++)      // Ani tohle neni dokonale, jenze more
                                      //   advanced uprava by stala moc prace :)
        if( M[j-1] > M[j] )           // Prohoď je
```

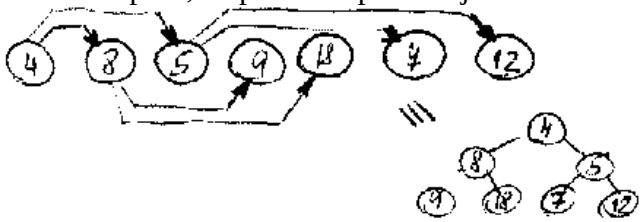
Špatně napsaný bubble sort je neefektivní, protože pokaždé (*i* pro správně seřazení pole) se zaměňování provede *n*-1 krát. Jelikož po jednom průchodu se nám vždy největší prvek dostane na konec pole, můžeme algoritmus vylepšit a zapamatovat si index poslední záměny. Nad tímto indexem jsou prvky s větší hodnotou a jsou již uspořádány (protože tam neproběhlo žádné prohození). v příštím kroku nejedeme s indexem *j* do konce pole ale jen do nalezeného indexu poslední změny v minulém kroku. Algoritmus ukončíme, pokud nedošlo k žádné změně.

Počáteční hodnoty klíčů :	10	12	4	9	18	11
j = 1	10	12	4	9	18	11
j = 2	10	4	12	9	18	11
j = 3	10	4	9	12	18	11
j = 4	10	4	9	12	18	11
j = 5	10	4	9	12	11	18

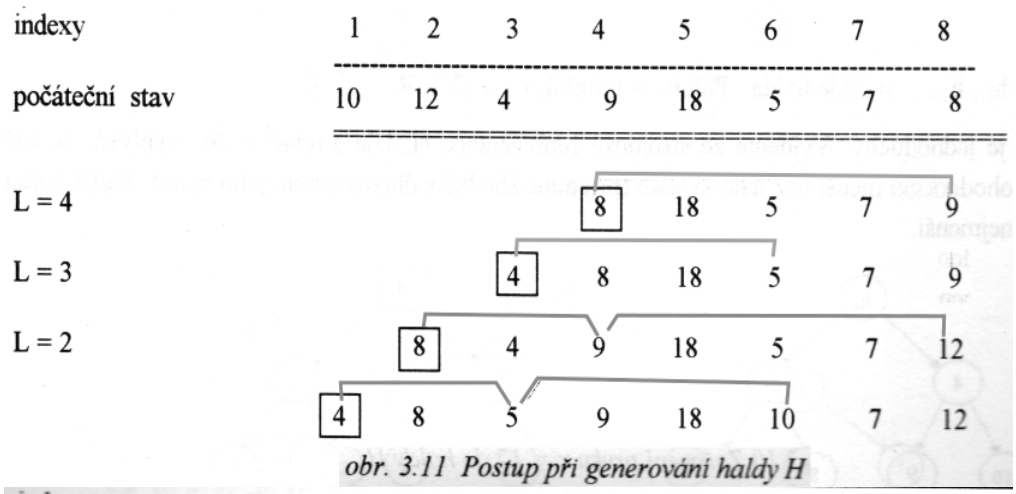
Shaker Sort – upravený vylepšený bubble sort. V každém kroku měníme směr procházení a vždy jedeme od indexu poslední změny z minulého kroku do indexu poslední změny v daném směru.

1.4 Heap-sort (nestabilní)

Hromada je reprezentací binarniho stromu v sekvencnim poli.
Prvek na prvnim miste (index 1) je korenem a má minimální / maximální hodnotu.
Pro hromadu platí, že prvek na pozici *i* je menší / větší nežli prvky na pozici *2i* a *2i+1*.



1. udělej z pole hromadu (pro *i = 1* až *i = n/2* musí platit A[*i*] <= A[*2i*] a A[*i*] <= A[*2i+1*], pokud toto neplatí prohoď A[*i*] s menším z nich)
2. kořen prohoď s posledním prvkem
3. zmenši pole o poslední prvek
4. proveď seřazení na hromadě (neporovnávám celé pole, ale jen tu větev, která nevyhovuje podmínce)
5. pokračuj 2. dokud není pole prázdné



obr. 3.11 Postup při generování haldy H

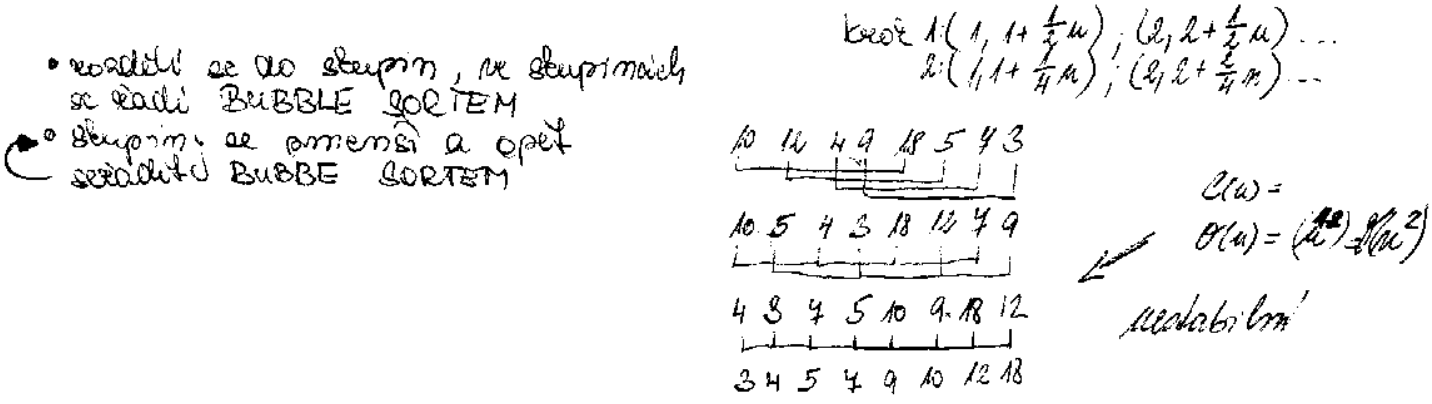
index	1	2	3	4	5	6	7	8
R = N	4	8	5	9	18	10	7	12 |
R = N - 1	5	8	7	9	18	10	12 |	4
R = N - 2	7	8	10	9	18	12 |	5	4
R = N - 3	8	9	10	12	18 |	7	5	4
R = N - 4	9	10	18	12 |	8	7	5	4
R = N - 5	10	12	18 |	9	8	7	5	4
R = N - 6	12	18 |	10	9	8	7	5	4
R = N - 7	18 |	12	10	9	8	7	5	4

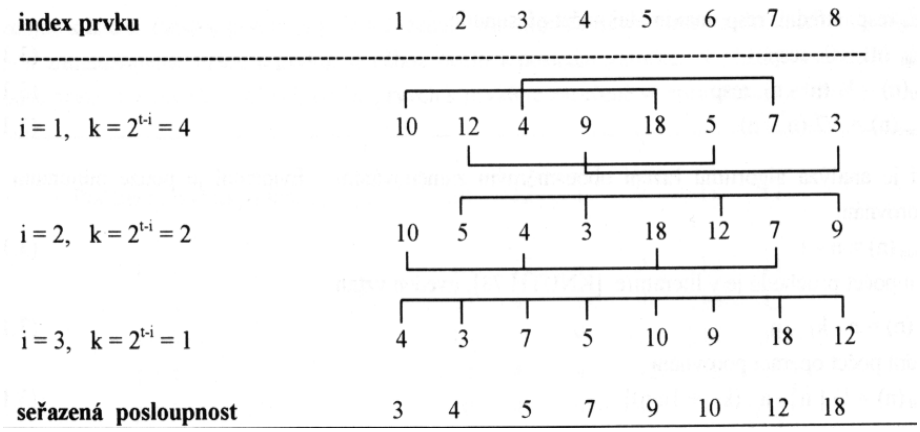
obr. 3.12 Příklad řazení metodou Heap-Sort

1.5 S klesajícím krokem – Shell-sort (nestabilní)

Pole rozdělíme do několika disjunktních skupin a tyto skupiny seřadíme bubble sortem. V dalším kroku skupiny zvětším (počet hodnot v každé skupině vzroste, počet skupin klesne) a znovu bubble sortem setřídíme. Takto pokračujeme, až máme skupinu jednu a řadím bubble sortem celé pole. Počet skupin je dobré brát z řady *1, 3, 7, 15, 31, ... , 2i+1* (ve skriptech ale mají zrovna na příkladu ukázány 4 skupiny, pak se zredukujou na 2, a pak na jednu.. viz obrázek na konci Shell-sortu)

Počet porovnání a přesunu je prý úměrný $n^{1.2}$ => O(*n*) = O(n^2).





1.6 Opakovaným tříděním – Quick-sort (nestabilní)

Rekurzivní metoda. Myšlenka spočívá v tom, že vezmu nějakou hodnotu (říká se jí pivot) a prvky pole proházím tak, aby levá část pole měla prvky s hodnotou menší než pivot a pravá větší. Nejlepší je pokud pivot je medián prvků, pak jsou totiž obě části (pravá i levá) stejně veliké. (V praxi se ale spíše používá například aritmetický průměr z několika náhodně vybraných hodnot.) Po proházení aplikuji rekurzivně totéž na levou a pravou část znova (opět volím nového pivota a prohazuji).

Trideni(A,L,R)

1. zvol pivota mezi L a R; i=L; j=R
2. dokud A[*i*] < pivot, *i*++
3. dokud A[*j*] > pivot, *j*--
4. pokud *i* <= *j* prohoď A[*i*] s A[*j*]
5. pokud *i*<= *j* jdi na 3.
6. pokud L<j Trideni(A,L,j)
7. pokud i<R Trideni(A,i,R)

QuickSort(A,n){ Trideni(A,0,n-1); }

```
procedure QUICK_SORT( var A:POLE_PRVKU;      {Řazené pole      }
                          n:integer          {Počet prvků pole} );
        {Procedura seřadí zadané pole metodou Quick-Sort}
 procedure TRIDENI(L,R:INDEX);
  var I,J : INDEX;
      W : PRVEK;
      X : TYP_KLIC;
  begin
    I := L; J := R;
    X := A[(L+R) div 2].KLIC;      { výběr pivota }
    repeat
      while A[I].KLIC < X do I := I+1;
      while X < A[J].KLIC do J := J-1;
      if I <= J then begin          { záměna prvků }
                     W:= A[I]; A[I]:= A[J]; A[J]:= W;
                     I := I+1; J := J-1
                    end;
    until I > J;
    if L < J then TRIDENI(L,J);
    if I < R then TRIDENI(I,R)
  end;  {konec pomocné procedury TRIDENI }
begin
  TRIDENI(1,N)
end;  {konec procedury QUICK_SORT}
```

			po provedení procedury TRIDENI									
L	R	X	pole A								I	J
			1	2	3	4	5	6	7	8		
			4	10	12	9	5	18	7	8		
1	8	9	4	8	7	5	9	18	12	10	5	4
1	4	8	4	5	7	8					4	3
1	3	5	4	5	7						3	1
5	8	18					9	10	12	18	8	7
5	7	10					9	10	12		7	5
			4	5	7	8	9	10	12	18		

tab. 3.1 Postup řazení procedurou QUICK-SORT (první zvýrazněný řádek představuje vstupní řazené pole, poslední řádek představuje seřazené pole).

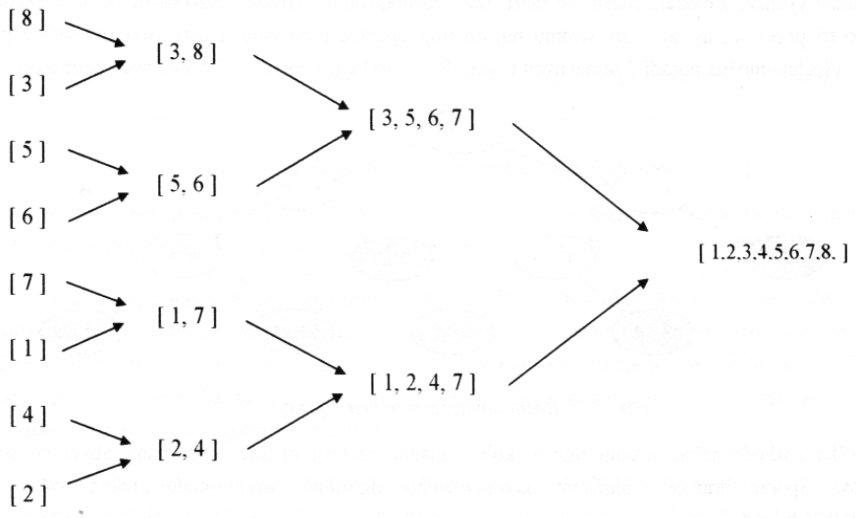
1.7 Slučováním – Merge-sort (stabilní)

Rekurzivní metoda. Založeno na principu Rozděl a Panuj. Máme-li dvě seřazená pole, sloučit je do jednoho seřazeného je snadné (beru postupně prvky z obou polí a porovnávám který ze dvou je menší a ten dám do výsledného pole). K tomu potřebuji pomocné pole délky n.

MergeSort(A){

1. Rozděl A na poloviny
2. pokud je velikost A1 > 1 tak MergeSort(A1)
3. pokud je velikost A2 > 1 tak MergeSort(A2)
4. Sluč A1 a A2

}



obr. 3.15 Příklad řazení metodou Merge-Sort

1.8 Třídění podle řádu – Radix-sort (stabilní)

Řazení funguje pro klíče o základu *b* délky L. Postupujeme tak, že třídíme prvky do *b* tříd od 0-tého do L-1 řádu klíče. Prvky v třídě tvoří frontu. Složitost je O(*n*) = O(L*n*). Pro L kont. je to O(*n*) a v nejhorším případě pro L = $\log_b n$.

1. *i* = 0
2. roztřiď pole podle řádu *i*
3. *i*++ a pokračuj 2 dokud *i* < L
4. Zapiš výsledné pole (prvky se už neberou v pořadí, jak byly na vstupu, ale jak byly seřazeny v předchozím kroku)

Vstup: 125,082,625,324,715,304,065,706,102

	Třídy									
řád	0	1	2	3	4	5	6	7	8	9
0			082		324	125	706			
			102		304	625				
						715				
						065				
1	102	715	324				065		082	
	304		125							
	706		625							
2	065	102		304			625	706		
	082	125		324				715		

1.9 Výpočtem pořadí – Counting-sort (stabilní)

Pracuje na principu, že si vypočítám index, na který se má prvek ve výsledném poli uložit a to tím způsobem, že si alokujeme pole veliké jako je největší hodnota v poli a pak do tohoto pole uložíme, kolikrát se daná hodnota ve vstupním poli opakuje. Musí platit, že hodnoty jsou z intervalu 1..*k*, pokud je k = O(*n*) jde o řazení v lineárním čase (ale s brutálními nároky na paměť – potřebuje pomocné pole velikosti *k* a pole pro výsledek délky *n*).

1. alokuj pomocné pole Pom o velikosti *k* a vynuluj ho
2. for (*i* = 1; *i* <= *n*; *i*++) Pom[A[*i*]]++ — kolikrát je hodnota na vstupu zastoupena
3. for (*i* = 2; *i* <= *n*; *i*++) Pom[*i*] += Pom[*i*-1] — jaký má index ve výstupním poli
4. for (*i* = n; *i* > 0; *i*--) Vys[Pom[A[i]]--] = A[i] — umístění prvku na správnou pozici a aktualizace indexu pro další prvek se stejnou hodnotou

1.10 Reálných čísel – Hybrid-sort (stabilní)

Předpokládejme, že hodnoty jsou z intervalu *<0,1)* a ten je rovnoměrně rozdělen na *k* dílčích intervalů. Jednotlivé prvky roztřídíme do *k* intervalů podle jejich hodnot a to tak, interval = celá_část(*hodnota* * *k*). Tyto intervaly seřadíme některou známou metodou a výsledné intervaly za sebou spojíme.

```
1: for j := 0 to k-1 do  VYTVOŘ PRÁZDNOU TŘÍDU ;{inicializace tříd}
2: for i := 0 to n do    ZAŘAĎ PRVEK p_i do třídy T{ ⌈k.p_i⌉ };
3: for j := 0 to k-1 do  SEŘAĎ TŘÍDU T_j ;
4: VYTVOŘ PRÁZDNÝ SEZNAM S ;
   for j := 0 to k-1 do  PŘIPOJ TŘÍDU T_j k seznamu S;
```

1.11 Řazení sekvenčních souborů

Používá se **Merge-sort**. U ostatních algoritmů se předpokládal přímý přístup => přístup k poli. Disperze souboru (Soubor A rozdělen na soubory B a C). Potřebujeme tedy 3 soubory – označováno také jako 3-souborové slučování.

1.12 Řazení souborů s přímým přístupem

Možno použít předchozí algoritmy (pokud nejsou soubory příliš velké je lepší je nahrát nejdřív do paměti, v opačném případě je dobré použít alespoň nějaký buffer)

1.13 Operační a paměťová složitost algoritmů řazení

Metoda	Stabilita	Asymptotická operační složitost			Paměťová složitost
Select-sort	ANO	n^2	n^2	n^2	n*lp
Insert-sort	ANO	n	n^2	n^2	n*lp
Bubble-sort	ANO	n	n^2	n^2	n*lp
Heap-sort	NE	$n*\log_2 n$	$n*\log_2 n$	$n*\log_2 n$	n*lp
Shell-sort	NE	n^1.2	n^1.2	n^1.2	n*lp
Quick-sort	NE	$n*\log_2 n$	$n*\log_2 n$	n^2	$n*lp+c*\log_2 n$
Merge-sort	ANO	$n*\log_2 n$	$n*\log_2 n$	$n*\log_2 n$	2n*lp
Radix-sort	ANO	n*L	n*L	n*L	n*lp+(n+2z)*li
Counting-sort	ANO	2n+2k	2n+2k	2n+2k	2n+k
Hybrid –sort	?	?	?	?	?

lp – délka prvku pole
li – délka zobrazení typu „index klíče“
z – základ soustavy klíče
L – délka klíče

Metoda řazení	Asymptotická operační složitost			Paměťová složitost	Přístup	Stabilita
	minimální	střední	maximální			
výběrem	n^2	n^2	n^2	$n.l_p$	P	ANO
vkládáním	n	n^2	n^2	$n.l_p$	P	ANO
zaměňováním	n	n^2	n^2	$n.l_p$	P	ANO
bin. vkládání	$n.\log_2 n$	n^2	n^2	$n.l_p$	P	ANO
Heap_Sort	$n.\log_2 n$	$n.\log_2 n$	$n.\log_2 n$	$n.l_p$	P	NE
Quick_Sort	$n.\log_2 n$	$n.\log_2 n$	n^2	$n.l_p + c.\log_2 n$	P	NE
Shell_Sort	$n^{1.2}$	$n^{1.2}$	$n^{1.2}$	$n.l_p$	P	NE
Merge_Sort	$n.\log_2 n$	$n.\log_2 n$	$n.\log_2 n$	$2.n.l_p$	S	ANO
Radix_sort	n.L	n.L	n.L	$n \cdot l_p + (n+2.z) \cdot l_i$	P	ANO

tab. 3.3 Charakteristiky základních metod řazení (P-přímý přístup, S-sekvenční přístup, l_p-délka prvku řazeného pole, l_i-délka zobrazení typu „index prvku", z-základ soustavy klíče, L - délka klíče, l_{pom}-délka proměnných ukládaných do zásobníku)