Алгоритми и структури на податоци

– белешки од предавања –

8. Внатрешни сортирања

8.1 Пребарување низ колекција записи

Нека е дадена колекција на записи (од кои секој може да има повеќе полиња) претставени во форма на датотека или низа. Полињата од таа низа кои овозможуваат записите да се разликуваат (на пример поради својата единственост) се нарекуваат клучни полиња или клучеви. Колекцијата на записи може да се чува на два начина: секвенцијален или несеквенцијален. Секвенцијалниот начин подразбира дека записите се подредени во опаѓачки или растечки редослед, односно дека се сортирани во однос на клучното поле. Несеквенцијалниот начин на чување на колекцијата на записи не го гарантира таквиот нивен редослед, односно записите се чуваат во произволен редослед во однос на клучното поле.

Да претпоставиме дека некоја колекција записи K_1, ...,K_n се чува секвенцијално, и нека записите се составени само од едно поле кое воедно е и клучно. Во терминологијата на низи би рекле: нека е дадена една сортирана низа. Да го разгледаме пребарувањето на запис со клуч К во таа колекција записи (низа).

Еден можен алгоритам за пребарување би бил да се почне од едниот крај на низата и последователно да се споредуваат клучевите се додека (и ако) се најде саканиот. Таквиот алгоритам за ребарување се нарекува секвенцијално пребарување и една можна изведба е дадена со следниот псевдо код:

```
procedure SEQSRCH(n,i,K)
//Prebaruvaj gi vrednostite na klucevite K1, ...,Kn za zapis Ri za koj
Ki = K. Ako ne postoi takov zapis, i se postavuva na 0//
K0 ← K; i ← n
while Ki ≠ K do
i ← i - 1
end
end SEQSRCH
```

Алгоритмот е реализиран така што се воведува еден помошен клуч на нулто место во низата, кој ни овозможува да не се грижиме индексот і да стане толку мал што ќе излезе од рангот на низата. Имено доколку клучниот запис не постои проверката во циклусот ќе заврши за i=0, што според дефиницијата на псевдо кодот е очекуваната вредност за і во случај клучната вредност да не постои во низата. Во тој случај предходниот алгоритам извршува n+1 проверки. Доколку се бара клучот K_i, алгортмот врши $n - i + 1$ проверки (доколку сите клучеви се различни). Средниот број на проверки би бил сума од проверките потребни за

сите можни локации на клучот поделен со бројот на можните локации на клучот или

$$\sum_{1\le i<n}(n-i+1)/n=(n+1)/2$$

За големи вредности на n ова пребарување содржи голем број на проверки, па според тоа не е многу ефикасно.

Предходно опишаниот начин на пребарување не го искористува секвенцијалното (подредената) или сортираното чување на податоците. Доколку ја искористиме идејата на пребарување кое постои кај бинарните пребарувачки стебла алгоритмот би можел да се модифицира на следниот начин под претпоставка дека низата е дадена во неопаѓачки редослед (дозволени се и дупликати на клучевите):

Клучот кој го бараме, К го споредуваме со клучот кој се наоѓа на средината на низата K_m , каде m= n/2. Можни се три случаи:

(i) Ако $K < K_m$, тогаш доколку клучот постои во низата тоа е делот од низата со помали индекси;

(ii) Ако $K = K_m$, тогаш клучот е пронајден;

(iii) Ако $K > K_m$, тогаш доколку клучот постои во низата тоа е делот од низата со поголеми индекси;

Предходните чекори се повторуваат се додека се најде клучот или не постои дел од низата за проверка.

Со секоја проверка за половина се намалува множеството на клучеви кои треба да се проверат. (За разлика од секвенцијалното пребарување каде со секоја проверка множеството на клучеви што остануваат за проверка се намалува за еден.) Поради тоа и во најлош случај овој метод има O (log n) споредби кои треба да се извршат. Една можна изведба на овој алгоритам кој е наречен бинарно пребарување е даден со следниот псевдо код:

```
procedure BINSRCH(n,i,K)
//Prebaruvaj podredeni zapisi R1,...,Rn i klucevite K1 ≤ K2 ≤ … ≤ Kn
za zapis Ri za koj Ki = K; i = 0 ako ne postoi takov zapis inaku Ki =
K. Niz algoritmot, l e najmaliot indeks za koj Kl moze da bide K i u
e najgolemiot indeks za koj Ku moze da bide K//
l ← 1; u ← n
while l ≤ u do
m ← ⌊(l+ u)/2⌋      //presmetaj go indeksot na srednoit zapis//
case
:K > Km: l ← m + 1       //baraj vo povisokata polovina//
:K = Km: i ← m; return
:K < Km: u ← m - 1       //baraj vo poniskata polovina//
end
end
i ← 0    //nema zapis so kluc K//
end BINSRCH
```

Релативно лесно може да се покаже дека доколку се земат средните времиња на извршување, бинарното пребарување е поефикасно од секвенцијалното за $n > 15$, додека во случај на најлоши времиња на извршување бинарното пребарување е подобро за $n > 4$. Обиди се да ги потврдиш овие сознанија!

8.2 Споредување на колекции записи

Да го разгледаме проблемот кој е релативно чест во праксата и кој бара да се провери дали две колекции на записи (вообичаено од два различни извори) се исти. Практична примена на овој проблем може да биде споредување на списоците за воени обврзници добиени од Министерството за внатрешни работи и од Министерството за одбрана.

Доколку се претпостави дека овие колекции на записи се чуваат на несеквенцијален начин, еден очигледен пристап за решавање на овој проблем би бил:

За секој запис од првото множество секвенцијално провери дали тој се наоѓа во второто множество. Очигледно е дека овој алгоритам има $O(n^2)$ проверки односно комплексност во случај кога колекциите имаат n записи.

Доколку се претпостави дека овие колекции на записи се чуваат на секвенцијален начин, еден очигледен пристап за решавање на овој проблем би бил:

Провери дали елементите со исти индекси се еднакви, за што се доволни O(n) споредби.

Ефикасноста на вториот пристап е неспоредливо подобра од ефикасноста на првиот пристап. Уште повеќе, наскоро ќе се потврди дека дури и кога колекциите на записи се несеквенцијални, подобро е прво тие да се сортираат, а потоа да се примени вториот алгоритам за споредба на сортирани колекции записи. Комплексноста на тој пристап во општ случај ќе биде еднаква на комплексноста на двете сортирања $O(n\log_2 n)$ собрана со линеарна комплексност:

$$O(n\log_2 n) + O(n\log_2 n) + O(n) = O(2n\log_2 n+n) = O(n\log_2 n) < O(n^2)$$

8.3 Сортирање на колекции записи

Во предходниот текст беа прикажани две важни примени на сортираните колекции на записи: (i) сортираните записи овозможуваат поефикасно пребарување и (ii) сортираните записи овозможуваат побрзо споредување. Сортирањето наоѓа примена и во решавањето на многу други проблеми. Според некои автори, се претпоставува дека на операциите на сортирање се троши помеѓу 25% и 50% од процесирачкото време на компјутерите, во зависност од доменот на нивната примена. Поради тоа проблемот на сортирање и алгоритмите кои го решаваат (реализираат) имаат голема важност во

компјутерските науки. За жал, не постои најдобар алгоритам за сортирање. Во продолжение ќе бидат разгледани неколку методи и ќе бидат дадени дискусии во кои случаи кој метод дава подобри резултати.

Проблемот на сортирање формално може да се дефинира како барање на пермутација на колекцијата записи за која важи дека клучевите (клучните полиња) се секвенцијални.

Алгоритмите за сортирање може да бидат внатрешни и надворешни.

Алгоритмите за внатрешно сортирање се користат кога колекцијата на записи е доволно мала за да може целата да се смести во оперативната меморија. Алгоритмите за внатрешно сортирање традиционално се базирани на споредување на вредности. Но постојат и алгоритми кои не се базирани на споредување на вредности.

Доколку колекцијата на записи целосно не може да се смести во оперативната меморија се користат така наречени алгоритми за надворешно сортирање.

8.4 Алгоритми за внатрешно сортирање со помош на споредување

Сортирање преку наоѓање на максимум (MAXIMUM ENTRY SORT)

Наједноставен начин за сортирање на колекција на записи е преку наоѓање на минималната (максималната) вредност на клучевите на записите и негово ставање на прво место во резултантната листа. Доколку резултантната листа е еднаква со појдовната (што е препорачливо поради штедење на меморискиот простор), ставањето на прво место може да се реализира со замена на вредностите на записите на првиот елемент и максималниот елемент. Оваа постапка потоа се повторува за сите останати записи (во вториот чекор почнувајќи од вториот запис, во третиот чекор почнувајќи од третиот запис и така натаму), се додека појдовната низа не се испразни, а со тоа се резултира во сортирана низа. Алгоритмот е многу едноставен и поради тоа е добар за самостојна изработка.

Реализацијата на овој алгоритам бара извршување на n - 1 замени и вкупно (n - 1) + (n - 2) + . . . + 2 + 1 споредби за наоѓање на максималниот елемент во сите итерации кои се потребни за да се изврши алгоритмот. Од тука следи дека времетраењето на извршување на алгоритмот е многу големо што резултира во неефикасен алгоритам. Пресметката на времетраењето на алгоритмот е дадена со:

$(n - 1) + (n - 1) + (n - 2) + \ldots + 2 + 1 = (n - 1) + [n(n - 1)]/2 = O(n^2)$.

Сортирање како со меури (BUBBLE SORT)

Алгоритмот за сортирање ***bubble sort*** работи според начинот на кој при мешање на делови со различни големини, најмалите доаѓаат на дното, додека најголемите доаѓаат на површината. Реализацијата на овој алгоритам е многу едноставна, иако идејата која се реализира не е. Овој алгоритам често се употребува за сортирање во школски услови поради едноставноста тој да се реализира. Тој се имплементира во повеќе поминувања низ листата што се поставува. Во секое поминување се споредуваат соседните елементи и доколку има потреба (доколку поголемиот елемент е на лево, а сакаме низа во опаѓачки редослед или обратно) си ги заменуваат местата. Со ваквата промена на местата, се гарантира дека во една итерација најмалиот (односно најголемиот) елемент ќе се најде на почетокот на низата. Во втората итерација (поминување) истото се врши со останатите елементи. Доколку во некое поминување не се случи измена, може да се заклучи дека низата е сортирана. На слика 8.1 е даден изгледот на некоја низа која се сортира на овој начин после две поминувања.

	Податок
1	101
2	44
3	62
4	90
5	15
6	50
7	44
8	160
9	202
10	156
11	17
12	36
13	427
14	250
15	320
16	11
17	612
18	57
19	6
20	33

(а) Почетна низа

Податок
101
62
90
44
50
44
160
202
156
17
36
427
250
320
15
612
57
11
33
6

(б) после прво поминување

Податок
101
90
62
50
44
160
202
156
44
36
427
250
320
17
612
57
15
33
11
6

3-от најмал податок

(в) после второ поминување

слика 8.1 Илустрација на Bubble Sort

Една можна имплементација на ова сортирање е дадена со следниот псевдо код.

```
procedure BUBBLESORT (R,n)
//Sortira n zapisi smesteni vo podatocna niza R vo opagjacki
redosled.//
done ← FALSE;
while (not done) do
{
done ← TRUE;
for i ← 1 to n-1 do
if K_{i+1} > K_i then begin
        interchange (K_{i+1}, K_i );
        done = FALSE;
end if
end while
end BUBBLESORT
```

Доколку низата е веќе сортирана, bubble сортирањето се извршува со едно поминување (односно со n-1 споредби), па во најдобар случај (кој е тривијален) има време на извршување O(n). Во најлош случај потребно е да се извршат n-1 поминувања и $(n - 1) + (n - 2) + \ldots + 1 = [n(n - 1)]/2$ споредби и промени. Од тука, најлошото време на извршување на овој алгоритам за сортирање е од редот $O(n^2)$. Иако ова време е исто со времето на сортирање преку наоѓање на максимум, битно е да се нагласи дека сортирањето со наоѓање на максимум секогаш се извршува во време опишано со $O(n^2)$, додека времетраењето на bubble сортирањето зависи од почетната сортираност на низата. Колку е низата повеќе сортирана, толку времетраењето на сортирањето е пократко. Сепак и ова сортирање е пример за сортирање кое во принцип не е задоволително ефикасно.

Сортирање со вметнување (INSERTION SORT)

Основната идеа на овој алгоритам се состои во реализација на процедура за вметнување на запис R (со клуч K) на право место во предходно сортирана колекција од записи $R_1, R_2, \ldots, R_i$, со клучеви за кои важи ($K_1 \leq K_2, \ldots, \leq K_i$). На тој начин се постигнува резултантната колекција од i+1 записи исто да биде сортирана. Псевдо кодот даден во продолжение го илустрира овој алгоритам.

```
procedure INSERT (R,i)
//Vnesi zapis R so kluc K vo sortiranata sekvenca R_o, ...,R_i na takov
nacin sto i rezultantnata sekvenca e isto taka sortirana spored
klucot K. Pretpostavuvame deka R_o e zapis (koj e virtuelen) za koj K
≥ K_o//
j ← i
while K < K_j do
//pomesti go R_j edno mesto nagore za R da se vmetne levo od R_j//
R_{j+1} ← R_j; j ← j - 1
end
R_{j+1} ← R
end INSERT
```

Алгоритмот претпоставува дека постои некој виртуелен запис R_o со клуч $K_o = -\infty$ (односно клуч кој е секогаш помал од останатите клучеви). Тоа е потребно за да се упрости проверката за крај на циклусот.

Доколку се апсолвира алгоритмот за вметнување на запис на право место во сортирана колекција на записи, алгоритмот за сортирање со вметнување е прилично едноставен. Тој започнува со сортирана колекција од два записа R_o и R_1 (поради фактот дека $K_0 < K_1$) и продолжува со сукцесивно вметнување на записите R_2,R_3, ... R_n во таа сортирана секвенца од записи. Од тука, за да се сортира низа од n записи потребни се n - 1 вметнувања. Псевдокодот за ова сортирање INSORT е даден во продолжение

```
procedure INSORT(R,n)
//Gi sortira zapisite R1, ...,Rn vo neopagjacki redosled na vrednosta
na klucot K.Se pretpostavuva n > 1//
Ko ← -∞       //Kreira virtuelen zapis Ro za koj Ko < Ki, 1 ≤ i ≤ n//
for j ← 2 to n do
T ← Rj
call INSERT(T, j - 1)     //vmetni gi zapisite R2 do Rn//
end
end INSORT
```

Пример: За n = 5 и влезна колекција дадена како низа (5,4,3,2,1) која сакаме да ја сортираме во растечки редослед се добива следното најлошо сценарио за извршување на сортирањето со вметнување:

```
- ∞, 5, 4, 3, 2, 1                    [почетна секвенца]
- ∞, 4, 5, 3, 2, 1                          i = 2
- ∞, 3, 4, 5, 2, 1                          i = 3
- ∞, 2, 3, 4, 5, 1                          i = 4
- ∞, 1, 2, 3, 4, 5                          i = 5
```

Пример: За n = 5 и влезна низа (2, 3, 4, 5, 1) се добива следното сценарио за извршување на сортирањето со вметнување :

```
- ∞, 2, 3, 4, 5, 1                    [почетна секвенца]
- ∞, 2, 3, 4, 5, 1                          i = 2
- ∞, 2, 3, 4, 5, 1                          i = 3
- ∞, 2. 3, 4, 5, 1                          i = 4
- ∞, 1, 2, 3, 4, 5                          i = 5
```

Алгоритмот INSERT(R,i) во најлош случај прави i + 1 споредби пред да го изврши вметнувањето со што има комплексност за најлошо време на извршување од редот $O(i)$. INSORT ја повикува INSERT за i = 1,2, ...,n - 1 со што се добива збирно најлошо време на извршување од редот

$$O\left(\sum_{i=1}^{n-1} i\right) = O(n^2)$$

Релативно лесно се покажува дека ова е комплексноста и на средното време на извршување на алгоритмот.

Поради слабата ефикасност (релативна бавност), но голема едноставност за реализација, сортирањето со вметнување вообичаено се применува кога бројот на записи кои треба да се сортираат не е поголем од 30.

БРЗО СОРТИРАЊЕ (QUICKSORT)

Алгоритмот за брзо сортирање (како што кажува и неговото име) е алгоритам за сортирање кој има меѓу најдобрите времиња на извршување. Неговата основна идеа е запис со даден клуч да се позиционира на право место во однос на целата колекција записи, овозможувајќи сите записи што се на лево од него да имаат клучеви помали (или евентуално еднакви) од неговиот клуч и обратно, сите записи што се на десно од него да имаат клучеви кои се поголеми (или евентуално еднакви) од неговиот клуч. Така, по позиционирањето на клучот се добиваат две поднизи кои може да се сортираат на ист начин.

Рекурзивен псевдо код за извршување на овој алгоритам за сортирање е даден во продолжение.

```
procedure QSORT(m,n)
//Sortira zapisi R_m, ...,R_n vo neopagjacki redosled od klucot K.
Klucot K_m e slucajno izbran kako kontrolen kluc. Pokazuvacite i i j
se koristat da gi podelat podkolekciite od zapisi za da vo sekoe
vreme K_l ≤ K, l < i i K_l ≥ K, l > j. Se pretpostavuva deka K_m ≤ K_{n+1}//

if m < n
then [i ← m; j ← n + 1; K ← K_m]
loop
        repeat i ← i + 1 until K_i ≥ K;
        repeat j ← j - 1 until K_j ≤ K;
        if i < j
                then call INTERCHANGE (R(i),R(j))
        else exit loop
forever
call INTERCHANGE (R(m),R(j))
call QSORT (m,j - 1)
call QSORT (j + 1, n)]
end QSORT
```

Пример: За 10 влезни записи со клучеви (26, 5, 37, 1, 61, 11, 59, 15, 48, 19) во продолжение е даден изгледот на низата после секој повик на QSORT. Средните загради укажуваат на поднизите кои допрва треба да се сортираат.

R_1	R_2	R_3	R_4	R_5	R_6	R_7	R_8	R_9	R_{10}	m	n
[26	5	37	1	61	11	59	15	48	19]	1	10
[11	5	19	1	15]	26	[59	61	48	37]	1	5
[1	5]	11	[19	15]	26	[59	61	48	37]	1	2
1	5	11	[19	15]	26	[59	61	48	37]	4	5
1	5	11	15	19	26	[59	61	48	37]	7	10
1	5	11	15	19	26	[48	37]	59	[61]	7	8
1	5	11	15	19	26	37	48	59	[61]	10	10
1	5	11	15	19	26	37	48	59	61		

Најлошото време на извршување на овој алгоритам се добива кога низата е веќе сортирана. Во тој случај за секој елемент правиме O(n) споредувања што резултира во $O(n^2)$ најлошо време на извршување. Иако ова време е лошо, во принцип ретко се случува поради својата нелогичност (да се сортира веќе сортирана низа). Во најдобар случај, несортираната подниза што се добива на

лево и десно од избраниот клуч ќе биде приближно еднаква. Бидејќи барањето на позицијата на клучот е O(n), тогаш за времето на сортирање T(n) важи следната итеративна функција:

$$
\begin{aligned}
&T(n) \le cn + 2T(n/2)\\
&\le cn + 2(cn/2 + 2T(n/4))\\
&\le 2cn + 4T(n/4)\\
&:\\
&\le cn \log_2 n + nT(1) = O(n \log_2 n)
\end{aligned}
$$

Со покомплексна анализа може да се докаже дека и средното време на извршување на овој алгоритам е $O(n \log_2 n)$. Поради неговата ефикасност, овој алгоритам се препорачува за сортирање на поголеми колекции од записи.

Сортирање со спојување (MERGE SORT)

Пред да го опишеме алгоритмот за сортирање со спојување, да се задржиме на кратко на алгоритмот за спојување на две предходно сортирани низи $(X_l, ...,X_m)$ и $(X_{m+1}, ...,X_n)$ во нова исто така сортирана низа $(Z_l, ...,Z_n)$. Идејата е релативно едноставна: се почнува од почеток на двете појдовни (на пример неопаѓачко сортирани) низи и во резултантната низа се префрла помалиот од двата елемента. Низата од која е префрлен елементот во резултантната низа, го дава следниот елемент за споредување. Ваквото споредување на парови елементи се повторува се додека една од низите не ги даде сите елементи. Елементите од другата низа потоа едноставно се префрлаат во резултантната низа.

Еден можен псевдо код за ова сортирање на две поднизи е даден во продолжение.

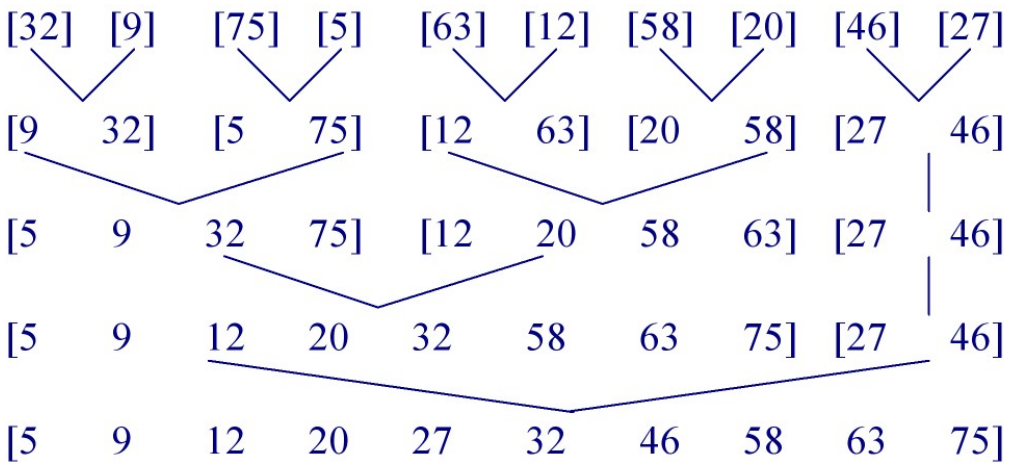
```
procedure MERGE(X,l,m,n,Z)
//(Xl, ...,Xm) i (Xm+1, ...,Xn) se dve sortirani datoteki so klucevi
xl ≤ ... ≤ xm i xm+1 ≤ ... ≤ xn. Tie se spoeni za da se dobie sortirana
datoteka (Zl, ..., Zn) za koja zl ≤ ... ≤zn//

i ← k ← l; j ← m + 1      //i, j i k se pozicii vo trite datoteki//
while i ≤ m and j ≤ n do
        if xi ≤ xj then [Zk ← Xi; i ← i + 1]
        else [Zk ← Xj; j ← j + 1]
k ← k + 1
end
if i > m then (Zk, ...,Zn) ← (Xj, ...,Xn)
else  (Zk, ...,Zn) ← (Xi ...,Xm)
end MERGE
```

Може да се забележи дека спојувањето на низите трае O(n) време.

Алгоритмот за сортирање со двојно спојување на колекција од n записи ги интерпретира тие записи како n низи со по 1 елемент (запис). Тие се спојуваат и формираат n/2 колекции записи (низи) со големина 2 (доколку n е непарен број,

една низа останува со должина 1). Оваа постапка продолжува се додека не резултира во една (сортирана) низа.

Пример: Влезната низа (32, 9, 75, 5, 63, 12, 58, 20, 46, 27) се сортира со помош на сортирање со двојно спојување на начин покажан на следната слика (слика 8.2).



слика 8.2 Илустрација на сортирање со спојување

Од примерот може да се увиди дека сортирањето со двојно спојување врши $\lceil \log_2 n \rceil$ промени над елементите од низата. Бидејќи алгоритмот MERGE ги спојува елементите од низата во линеарно време, секое изминување трае $O(n)$ време. Ова резултира во вкупно време на извршување на алгоритмот за сортирање со двојно спојување од $O(n \log n)$. Докажано е дека ова е едновремено и најлошо и просечно време за извршување на овој алгоритам.

Алгоритмот за сортирање со двојно спојување може да се реализира на повеќе начини. Во продолжение ќе биде дадена една можна реализација која се изведува со две функции (за поминување - MPASS) и за спојување MERGE.

```
procedure MPASS(X,Y,n,l)
//Ovoj algoritam izveduva edno pominuvanje od merge sort. Gi spojuva
sosednite parovi od podoblastite so dolzina l od datoteka X do
datoteka Y. n e brojot na zapisi vo X//

i ← 1
while i ≤ n - 2l + 1 do
call MERGE (X, i, i + l - 1, i + 2l - 1, Y)
i ← i + 2l
end
//spoi ja preostanatata oblast so dolzina < 2l//
if i + l - 1 < n then call MERGE (X, i, i + l - 1, n, Y)
else (Yi, ...Yn) ← (Xi, ...,Xn)
end MPASS
```

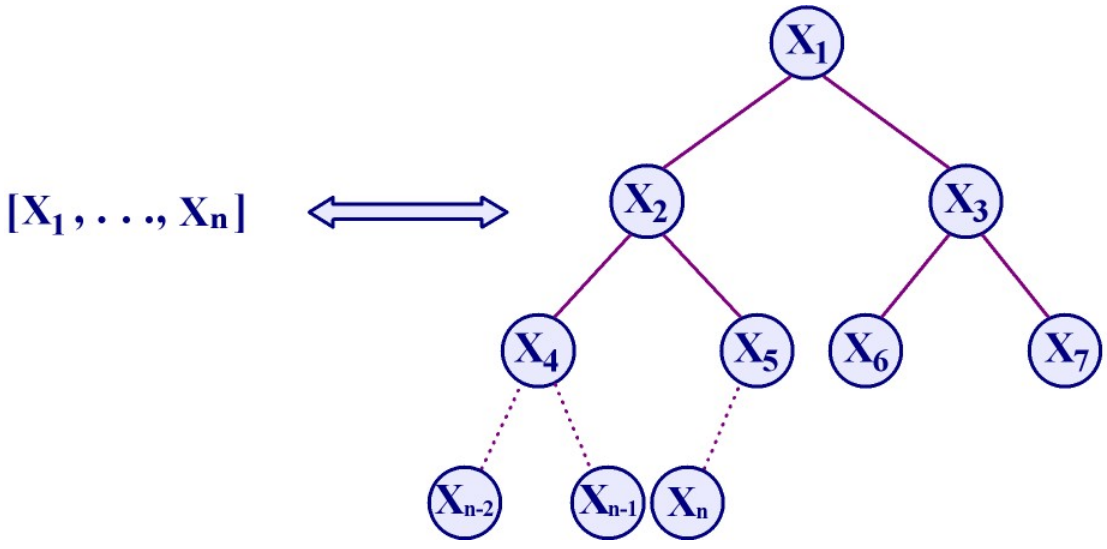
Алгоритмот за сортирање со спојување потоа ја добива следнава форма:

```
procedure MSORT(X,n)
//Sortiraj ja datotekata X = (X1, ...,Xn) vo neopagjacki redosled od
klucevite x1, ...,xn//
declare X(n), Y(n)          //Y e pomosna niza//
//l e goleminata na podoblastite koi momentalno se spojuvaat//
l ← 1
while l < n do
call MPASS(X,Y,n,l)
l ← 2*l
call MPASS(Y,X,n,l)          //promeni ja ulogata na X i Y//
l ← 2*l
end
end MSORT
```

HEAP SORT

Иако сортирањето со спојување има добро време на извршување, тоа бара дополнителен мемориски простор потребен за сместување на резултантната низа во секое изминување. Еден начин да се избегне овој проблем е користење на двојно поврзани листи (потсетете се на лабараториските вежби). Heap сортирањето бара помалку мемориски простор и едновремено постигнува перформанси од ист ранг на големина.

За да го илустрираме овој алгоритам ќе претпоставиме дека низата што треба да се сортира (x_1, ...,x_n) е претставена во форма на бинарно стебло. При тоа стеблото се пополнува од коренот кон листовите на начин на кој ќе биде максимално поплнето (на секој јазел прво се пополнуваат двете деца), како што е илустрирано на слика 8.3.



слика 8.3 Претставување на низа преку бинарно стебло

Тогаш, за да се реализира heap сортирањето потребно е да:

1. Креира heap стебло.

2. Додека heap стеблото не се испразни

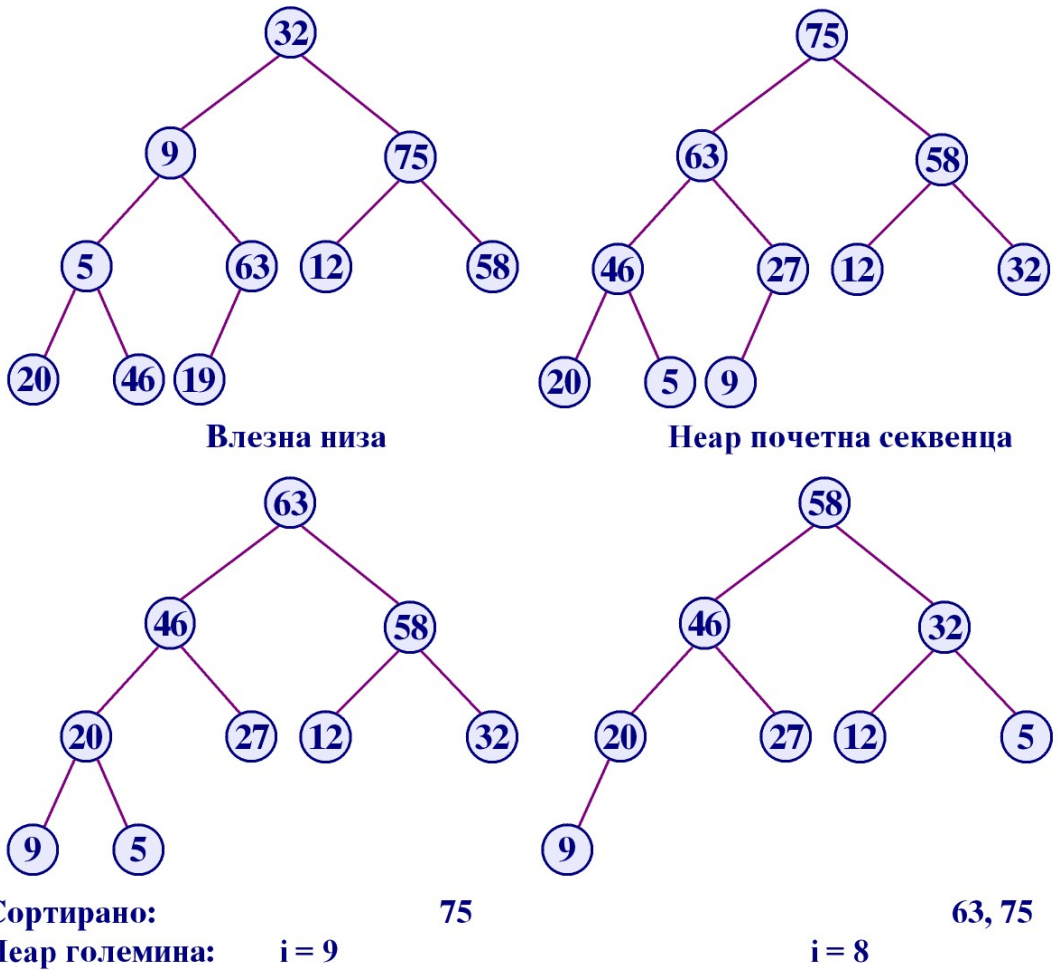
   а. да се смести записот (клучот) од коренот на heap стеблото во резултантната сортирана низа,

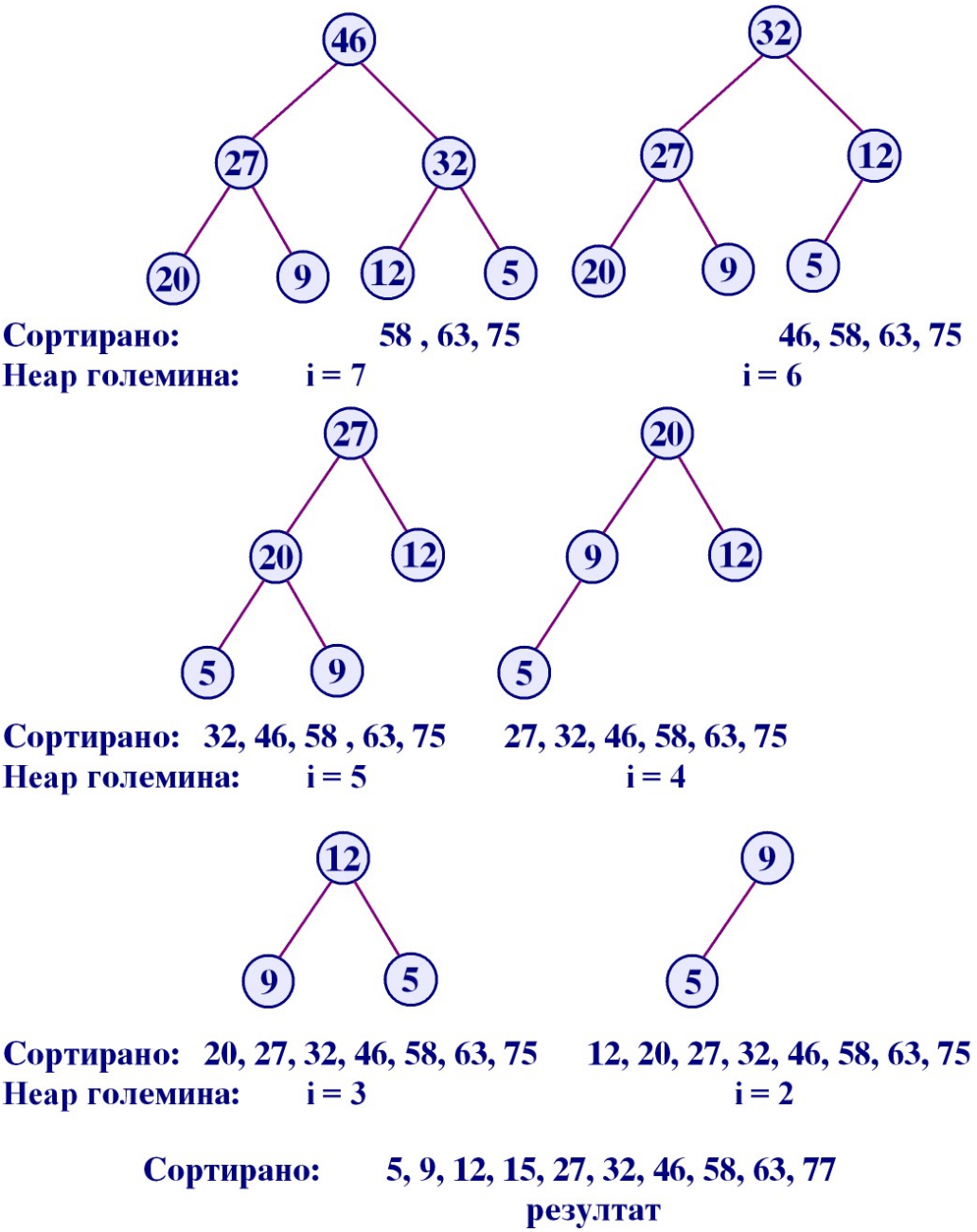
   б. да се извади тој елемент од heap стеблото

   в. повторно да се формира heap стебло.

Heap стебло е комплетно бинарно стебло за кое важи дека вредноста на клучот за јазелот родител е поголема или еднаква на вредноста на клучевите на неговите деца, за секој јазел во стеблото. Од тука произлегува дека коренот е јазел со најголема вредност на клучот.

Пример: Постапката на реализација на heap сортирањето за влезна низа со вредности (32, 9, 75, 5, 63, 12, 58, 20, 46, 27) може графички да се прикаже како на слика 8.4.





слика 8.4 Илустрација на heap сортирањето

Очигледно клучен дел од heap сортирањето е алгоритмот со кој дадено стебло се трансформира во heap стебло. Една реализација на тој алгоритам е дадена со следниот псевдо код. Стеблото е репрезентација на низа, па поради начинот на градење (види ја слика 8.3) важи дека јазелот кој го репрезентира записот R_j има деца R_{2j} и R_{2j+1}.

```
procedure ADJUST (i,n)
//Prilagodi go binarnoto steblo so koren i da go zadovoluva heap
uslovot. Levoto i desnoto podsteblo od i, na pr., so koreni 2i i 2i+
1, go zadovoluvaat heap svojstvoto. Jazlite od stebloto sodrzat
zapisi R, so klucevi K. Nieden jazel nema indeks pogolem od n//

R ← Ri; K ← Ki; j ← 2i
while j ≤ n do
if j < n and Kj < Kj+1 then j ← j + 1    //najdi go maksimumot od
levoto i desnoto dete//
//sporedi go maksimalnoto dete so K. Ako K e maksimumot togas e
zavrseno//
if K ≥ Kj then exit
R⌊j/2⌋ ← Rj; j ← 2j      //pomesti go Rj nagore vo stebloto//
end
R⌊j/2⌋ ← R
end ADJUST
```

Доколку длабочината на бинарното стебло кое се третира во предходниот алгоритам е к, циклусот се извршува најмногу к пати. Поради тоа времетраењето на извршувањето на алгоритмот е со големина $O(k)$.

Алгоритмот за heap сортирање може да се претстави со следниот псевдо код.

```
procedure HSORT (R,n)
//Oblasta R = (R1, ...,Rn) e sortirana vo neopagjacki redosled od
klucot K//
for i ← ⌊n/2⌋ to 1 by -1 do     //konvertiraj go R vo heap steblo//
call ADJUST (i,n)
end
for i ← n - 1 to 1 by -1 do     //sortiraj go R//
T ← Ri+1; Ri+1 ← R1; R1 ← T      //promeni gi R1 i Ri+1//
call ADJUST (1,i)      //povtorno kreiraj go heap stebloto//
end
end HSORT
```

Под претпоставка дека $2^{k-1} \leq n < 2^k$, каде k е висината на стеблото (бројот на јазли на висина i е 2^{i-1}). Во првиот **for** циклус, алгоритмот ADJUST се повикува по еднаш за секој јазел кој има дете. Поради тоа, времето потребно да се реализира овој циклус е сума (за секое ниво посебно) на бројот на јазли на дадено ниво, по максималниот број на чекори за преместување. Тогаш, времето потрено за извршување на алгоритмот не е поголемо од:

$$\sum_{1 \le i \le k} 2^{i-1}(k-i) = \sum_{1 \le i \le k-1} 2^{k-i-1} i \le n \sum_{1 \le i \le k-1} i/2^i < 2n = O(n)$$

Во вториот **for** циклус се извршуваат n - 1 повици на ADJUST со максимална длабочина $k = \lceil \log_2 (n + 1) \rceil$. Времето потребно за извршување на овој циклус е затоа O(n log n). Следствено, вкупното времетраење на извршувањето на heap сортирањето е $O(n \log n)$.

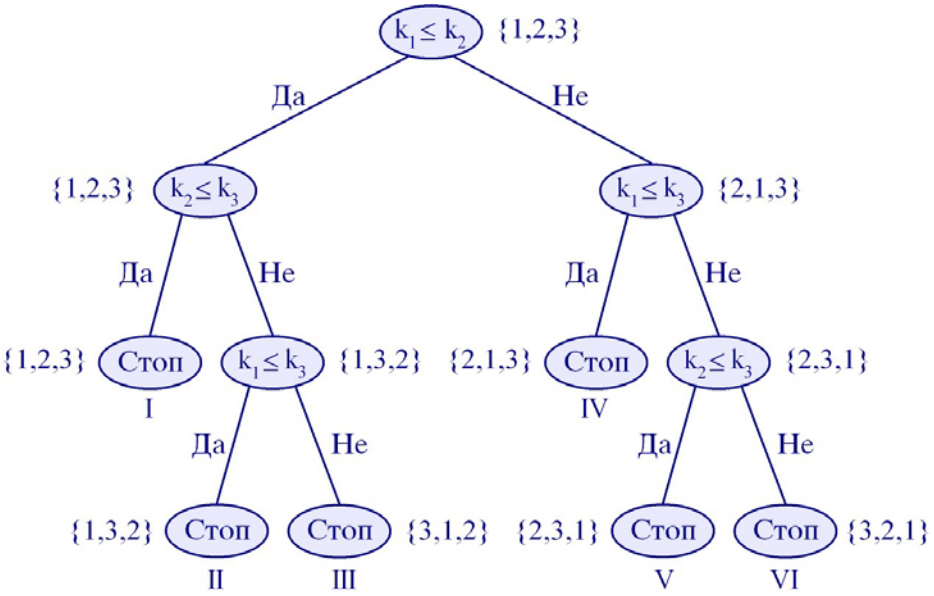
8.5 Генерална анализа на алгоритмите за сортирање со споредување

Предходно опишаните алгоритми го одредуваа редоследот на сортирање со помош на споредување на клучните вредности во множеството записи што треба да се сортира. Ваквите алгоритми за сортирање се нарекуваат алгоритми за сортирање со споредување. Анализата на алгоритмите покажа дека подобрите агоритми имаа просечно времетраење на извршувањето опишано со $O(n \log_2 n)$, а најлошото времетраење беше опишано со $O(n^2)$.

Сосема природно е да се постави прашањето: дали може да се дефинираат алгоритми кои ќе имаат време на извршување подобро од $O(n \log_2 n)$?

Да го разгледаме процесот на сортирање на три елементи со вредности (9, 7, 10). Според дефиницијата на сортирање, сортираната низа од три елементи е една можна пермутација од распоредот на тие три елементи. Да го разгледаме процесот на сортирање со помош на сортирањето со вметнување (иако анализата важи за произволен начин на сортирање). Процесот на сортирање може да се илустрира со помош на така наречено одлучувачко стебло, кое ќе ги опфати сите можни пермутации на трите елементи како можни влезови во процесот на сортирање. Гранките на стеблото укажуваат на можните преместувања како резултат на исполнување на да/не условот од нетерминалните јазли, додека јазлите на стеблото ги даваат можните пермутации на трите клуча. Процесот на сортирање, е процес на движење на стеблото се додека не се дојде до јазли кои ја означуваат посакуваната цел. Листовите ги претставуваат сите можни резултати (состојби во кои може да се најде редоследот на трите записа). Целото стебло е прикажано на слика 8.5. Интересно е да се напомене дека ваквите стебла често се користат при реализацијата на игри, експертски системи и сл.

лист	пермутација	вредности на клучевите за дадена пермутација
I	1 2 3	(7,9,10)
II	1 3 2	(7,10,9)
III	3 1 2	(9,10,7)
IV	2 1 3	(9,7,10)
V	2 3 1	(10,7,9)
VI	3 2 1	(10,9,7)



слика 8.5 Илустрација на стебло на споредби при сортирање на три елементи

Секое одлучувачко стебло кое го илустрира процесот на сортирање на n записи има n! листови (поради тоа што постојат n! можни редоследи на елементите). Бидејќи одлучувачкото стебло е бинарно стебло, со висина *k* има најмногу 2^{k-1} листови.

Од тука следи дека

$2^{k-1} \ge n!$, односно $k \ge \log_2(n!)+ 1$

Од што, поради фактот дека чекорите (времетраењето на) процесот на сортирање соодветствува со висината на одлучувачкото стебло следи дека комплексноста на алгоритмот е сигурно поголема од $\log_2(n!)+ 1$.

Доколку земеме дека важи

$n! = n(n - 1)(n - 2) \ldots (3)(2)(1) \ge (n/2)^{n/2}$,

добиваме

$k \ge \log_2(n!)+ 1 \ge (n/2) \log_2 (n/2) = O(n \log_2 n)$

од што се гледа дека алгоритмите за сортирање кои се базираат на споредување во општ случај не може да имаат подобро време на извршување. Ова може да се објасни и со фактот дека споредувањето е бинарна операција која често коже да се поистовети со пребарување. Да се спореди нешто често значи да се пронајде нешто. Бидејќи перформансите на пребарување во најдобар случај (бинарно пребарување) се $O(n \log_2 n)$, следи дека споредувањето на 1 елемент со n елементи ќе ги има приближно истите перформанси. Ако сортирањето се сведе на n споредувања со n елементи произлегува истиот резултат.

8.6 Сортирање во линеарно време

Доколку треба да се подобрат перформансите на сортирањето, односно сортирањето треба да се изврши во линеарно време, од предходното образложение следи дека мора да се користат алгоритми на сортирање кои не се базираат на споредба на клучеви. Таквите алгоритми обично се применуваат кога множеството на клучеви припаѓа на посебен домен, односно кога за него важат некакви специјални услови.

Сортирање со броење (COUNTING SORT)

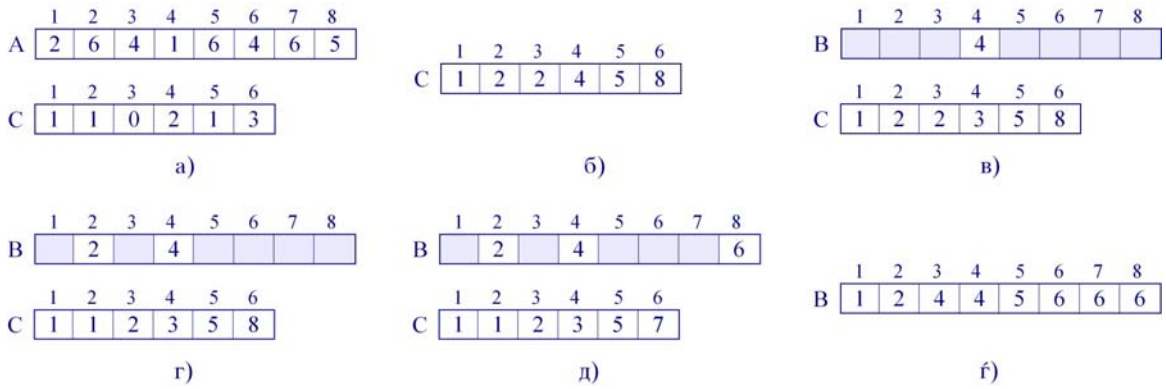
Сортирањето со броење претпоставува дека секој од *n*-те влезни елементи е цел број во опсегот од 1 до к, каде к е произволен цел број. Ако к може да се претстави како линерана функција од бројот на влезни елементи сортирањето се извршува во време $O(n)$.

Основната идеа на сортирањето со броење е слична со идејата присутна кај хеширањето. За даден клуч x, алгоритмот се труди да ги изброи елементите помали од x. Така, на пример, доколку постојат 3 елементи кои се помали од x, x треба да се смести на четвртата позиција во сортираната низа. Оваа идеа трпи мали модификации во случај на дозволено повторување на влезни елементи.

Една можна имплементација на сортирањето со броење, претпоставува дека влезната низа е дадена во форма $A[1 \,..\, n]$ и поради тоа важи $length[A] = n$. Потребни се две дополнителни низи: низата $B[1 \,..\, n]$ го содржи сортираниот излез, додека низата $C[1 \,..\, k]$ обезбедува привремен работен простор.

Пример за сортирањето со броење е дадено на слика 8.6. Влезната низа A[1..8] содржи елементи не поголеми од к=6. Помошната низа ни служи за да во неа се содржи бројот на елементите од низата што треба да се сортира кои се еднакви со индексот на помошната низа. Тоа е можно поради тоа што важи условот A[i] ≤ k. Така на слика 8.6а, се гледа дека C[1] = 1 поради фактот што постои една единица во низата A. Потоа за секој индекс од помошната низа C, ја пресметуваме сумата на елементите со индекси помали или еднакви од тековниот индекс. Така, на слика 8.6б, C[3] = 2 поради тоа што 2=1+1+0 = C[1] + C[2] + C[3], од низата C претставена на слика 8.6а.

Конечно елементот $A[j]$ можеме да го ставиме во коректната позиција во излезната низа B, така што ќе го сместиме на позиција $C[A[j]]$ во низата B. Тоа се должи на фактот што постојат (според предходно пресметаното) $C[A[j]]$ елементи кои се помали или еднакви на $A[j]$. Поради можноста за повторување на елементите во почетната низа, при секое внесување на елемент во низата B, вредноста $C[A[j]]$ се намалува за еден.



слика 8.6 Илустрација на сортирањето со броење

Псевдо кодот кој е имплементација на предходно кажаното е даден во продолжение.

```
COUNTING-SORT(A,B,k)
1 for i ← 1 to k
2     do C[i]← 0
3 for j ← 1 to length[A]
4     do C[A[j]] ← C[A[j]]+1
5 // C[i] sega sodrzi broj na elementi ednakvi so i.
6 for i ← 2 to k
7 do C[i]← C[i]+C[i-1]
8 //C[i] sega sodrzi broj na elementi pomali ili ednakvi so i.
9 for j ← length[A] downto 1
10    do B[C[A[j]]] ← A[j]
11       C[A[j]] ← C[A[j]]-1
```

Да направиме анализа на времето на извршување на овој алгоритам. **for** циклусот во линиите 1-2 има време на извршување $O(k)$, **for** циклусот во линиите 3-4 има време на извршување $O(n)$, **for** циклусот во линиите 6-7 има време на извршување $O(k)$ и **for** циклусот во линиите 9-11 има време на извршување $O(n)$. Од тука, вкупното време на извршување на целиот алгоритам е $O(k + n)$, односно кога к е линеарна функција од бројот на влезни елементи добиваме време на извршување $O(n)$.

Постојат и други сортирања со линеарно време на извршување (кои исто така не се базираат на споредување на клучевите). Примери за такви сортирања се: Radix сортирањето и сортирањето bucket sort.