Klasifikacija algoritama za sortiranje se može obaviti prema:
1. vremenskoj složenosti (u najgorem, prosečnom i najboljem slučaju) u zavisnosti od veličine ulaza;
2. veličini dodatnog memorijskog prostora koji je potreban da bi se obavilo sortiranje (u zavisnosti od veličine ulaza)
3. stabilnosti.
Stabilnost algoritama ima smisla definisati ako se sortiraju kolekcije složenih stuktura (npr. ako su elementi kolekcije slogovi koji sadrže više podataka kao što su: redni broj, ime, prezime, odeljenje). Stabilan algoritam je onaj koji čuva relativni (međusobni) poredak elemenata kolekcije u slučaju da elementi imaju isti ključ, tj. polje po kome se vrši uređivanje. To znači da ako elementi A i B imaju jednake vrednosti ključa, a u polaznoj strukturi podataka imaju takav raspored da je A ispred B, onda takav raspored mora da važi i u sortiranoj listi.

Sortiranje prosečne linearne složenosti
Sortiranje prosečne linearne složenosti spada u najjednostavnije postupke sortiranja. Naime, potrebno je osigurati dovoljan broj memorijskih lokacija, a onda smestiti elemente na odgovarajuće lokacije.

Neka je dato **n** elemenata, koji su celi brojevi iz opsega od 1 do m, m ≥ n. Rezerviše se *m* lokacija, a onda se za svako *i* broj x_i stavlja na lokaciju x_i koja odogovara njegovoj vrednosti. Posle toga se redom pregledaju sve lokacije i iz njih se pokupe elementi.
Složenost ovog algoritma je *O(m + n)*. Ako je *m = O(n)* dobijamo algoritam za sortiranje linearne složenosti. S druge strane, ako je *m* veliko u odnosu na *n*, onda je i *O(m)* takođe veliko. Pored toga, algoritam zahteva memoriju veličine *O(m)*, što dodatno predstavlja problem ako je *m* veliko.

Counting sort pretpostavlja da svaki član niza od *n* članova jeste ceo broj u intervalu od 0 do *k*, za neki ceo broj *k*. Kada važi da je *k* = *O*(*n*), ovo sortiranje ima vremensku složenost Θ(*n*).
A = nesortirani niz na ulazu
B= sortirani niz na izlazu
C= pomoćni radni niz,
c[i] je broj pojavljivanja broja ***i*** u nizu ***a*** u drugoj petlji nakon učitavanja

COUNTING-SORT(*A*, *B*, *k*)
1 **for** *i* ← 0 **to** *k*
	/* program count;*/
	#include<stdio.h>
	main()
	{int i,j,k,n;
	int a[100],b[100],c[100];
	scanf("%d%d",&n,&k);
	for (i=0;i<n;i++) scanf("%d",&a[i]);

2 **do** *C*[*i*] ← 0	for (i=0;i<k;i++) c[i]=0;
3 **for** *j* ← 1 **to** *length*[*A*] 4 **do** *C*[*A*[*j*]] ← *C*[*A*[*j*]] + 1 5 ▹ *C*[*i*] now contains the number of elements equal to *i*.	for (j=0;j<n;j++) c[a[j]]=c[a[j]]+1;
6 **for** *i* ← 1 **to** *k* 7 **do** *C*[*i*] ← *C*[*i*] + *C*[*i* - 1] 8 ▹ *C*[*i*] now contains the number of elements less than or equal to *i*.	for (i=1;i<k;i++) c[i]=c[i]+c[i-1];

9 **for** *j* ← *length*[*A*] **downto** 1
10 **do** *B*[*C*[*A*[*j*]]] ← *A*[*j*]
11 *C*[*A*[*j*]] ← *C*[*A*[*j*]] – 1
	for (j=n-1;j>=0;j--)
	{
	b[c[a[j]]]=a[j];
	c[a[j]]=c[a[j]]-1;
	}

	for (i=0;i<n;i++) printf("%d ",b[i]); }

1. Uporedite vremensku složenost algoritma COUNTING SORT i vremensku složenost algoritma QUICK SORT.

$\Theta(k)$ **for** $i \leftarrow 1$ **to** k **do** $C[i] \leftarrow 0$ $\Theta(n)$ **for** $j \leftarrow 1$ **to** n **do** $C[A[j]] \leftarrow C[A[j]] + 1$ $\Theta(k)$ **for** $i \leftarrow 2$ **to** k **do** $C[i] \leftarrow C[i] + C[i-1]$ $\Theta(n)$ **for** $j \leftarrow n$ **downto** 1 **do** $B[C[A[j]]] \leftarrow A[j]$ $C[A[j]] \leftarrow C[A[j]] - 1$ $\Theta(n + k)$	Vremenska složenost algoritma COUNTING SORT je $\Theta(n+k)$. Kada važi da je $k = O(n)$, ovo sortiranje ima vremensku složenost $\Theta(n)$. COUNTING SORT nije primer algoritma zasnovanog na upoređivanju i ne možemo koristiti stablo odlučivanja kao model koji smo koristili za algoritam QUICK SORT.

2. Sortirati niz 1 4 5 1 4 2 1 primenom algoritma COUNTING-SORT

Posle 2. petlje nakon učitavanja, evo kako izgleda niz A i niz broja pojava C:

A	***1***	***4***	***5***	***1***	***4***	***2***	***1***	**C**	*3*	*1*	*0*	*2*	*1*

Posle 3. petlje nakon učitavanja, evo kako izgleda niz C(**c[i]** je broj elemenata manjih ili jednakih od ***i***.):

C	*3*	*4*	*4*	*6*	*7*

Posle 4. petlje nakon učitavanja, evo kako izgledaju nizovi B, C:
U ovoj petlji se formira niz b, tako što se, najpre, na c[i]-to mesto u niz b postavi taj broj a[i], pa pošto je ono sada popunjeno, c[i] se smanjuje za jedan. Postupak se ponavlja dok se ne formira ceo niz b.
Dakle, n=7, a[n]=1 i element a[7]=1 se postavi na c[1]=3.poziciju u nizu b, a potom c[1]=c[1]-1=2.
Potom se i=7 smanji za 1, tj. element a[7-1]=2 se postavi na poziciju c[2]=4 u nizu b,...

B			*1*					*C*	***2***	*4*	*4*	*6*	*7*
B			*1*	*2*				*C*	*2*	***3***	*4*	*6*	*7*
B			*1*	*2*		*4*		*C*	*2*	*3*	*4*	***5***	*7*
B		*1*	*1*	*2*		*4*		*C*	***1***	*3*	*4*	*5*	*7*
B		*1*	*1*	*2*		*4*	*5*	*C*	*1*	*3*	*4*	*5*	***6***
B		*1*	*1*	*2*	*4*	*4*	*5*	*C*	*1*	*3*	*4*	***4***	*6*
B	***1***	***1***	***1***	***2***	***4***	***4***	***5***	*C*	***0***	*3*	*4*	*4*	*6*

2. Konstruišite algoritam koji će preprocesirati niz celih brojeva sa n članova u intervalu od 0 do *k u vremenu* $\Theta(n + k)$, a potom vremenu $O(1)$ *odrediti koliko članova niza je u intervalu* $[a, b]$.
Korak 1: Kao u algoritmu counting sort konstruisati niz C. (složenost $\Theta(n + k)$)
Korak 2: Broj članova u intervalu $[a \,..\, b]$ je $C[b] - C[a - 1]$, pri čemu smatramo da $C[-1]$ je 0. (složenost O(1))

3. Kreirajte modifikovanu verziju count2 counting sort-a tako što ćete **odrediti min i max niza a.** Prebrojavanje pojavljivanje članova niza ***a*** pamtite u pomoćnom nizu ***o*** prolaskom kroz elemente niza u intervalu od **min** do **max**.

```
/* program count2;*/
#include<stdio.h>
#include<limits.h>
#define MAX 200
```

```
main()
{
int i,j,n,min,max,k;
int a[MAX],o[4*MAX];
scanf("%d",&n);
min=INT _MAX;max=0; k=0;
for (j=0;j<=4*MAX;j++) o[j]=0;

for(i=1;i<=n;i++)
 {
  scanf("%d",&a[i]); o[a[i]]+=1;
  if (a[i]>max) max=a[i];
  if (a[i]<min) min=a[i];
 }

 for (i=min;i<=max;i++) if (o[i]!=0) for (j=1;j<=o[i];j++)
                                      {
                                        k=k+1;  a[k]=i;
                                       }

for (i=1;i<=n;i++) printf("%d ",a[i]);
}
```

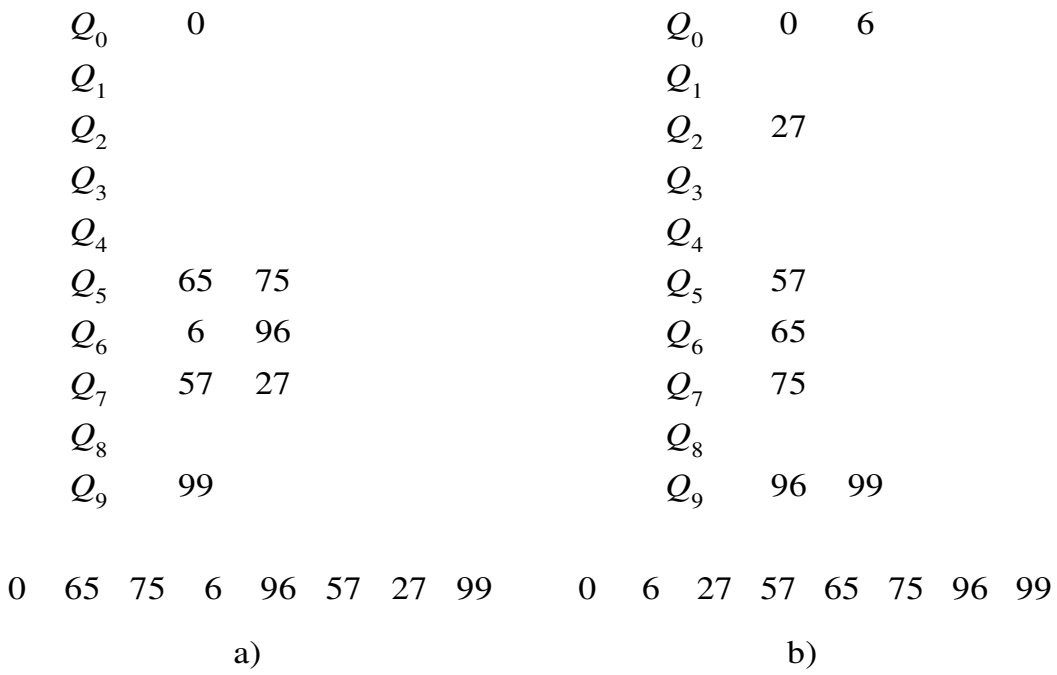
Sortiranje višestrukim razvrstavanjem (radix sort)

Sortiranje višestrukim razvrstavanjem predstavlja prirodno uopštenje najjednostavnijeg postupka sortiranja koji se sastoji u osiguravanju dovoljnog broja lokacija kako bi se svaki element smestio na svoju lokaciju.

Radix sortiranje koristi pozicionu reprezentaciju broja (ili niske karaktera), te odvojeno analizira cifre (ili znakove) na raznim pozicijama. Za demonstraciju ideje algoritma, bez gubitka opštosti, pretpostavlja se da su ključevi decimalni brojevi sa istim brojem od k cifara $d_{k-1}\ d_{k-2}\ ..\ d_0$.

RADIX-SORT(A, d)
1 **for** $i \leftarrow 1$ **to** d
2 **do** use a stable sort to sort array A on digit i

U prvom koraku može da se vrši sortiranje po najmlađoj cifri (iako postoji i metod koji u prvom koraku sortira najstariju cifru). Uzimaju se redom elementi iz neuređenog niza i razvrstavaju u deset redova $Q_0 \ldots Q_9$ po vrednosti najmlađe cifre, tako što se tekući element stavlja na kraj odgovarajućeg reda. Kad se tako obrade svi elementi, onda se svi redovi opet spoje u jedan niz po poretku $Q_0\ Q_1\ ...\ Q_9$. U sledećem koraku se opet uzimaju elementi ovog niza redom i razvrstavaju u redove, ali po vrednosti druge cifre d_1, pa se opet na kraju svi redovi spoje. Postupak se završava kad se izvrši uređenje i po najstarijoj cifri d_{k-1}. Primer rada algoritma je prikazan za dvocifrene ključeve na slici. Stanje posle prvog koraka prikazano je na slici **a**, a konačno stanje na slici **b**.



Slika: *Radix* sort - stanje posle sortiranja: a) po cifri jedinica i b) po cifri desetica

Suština ovog algoritma je u stabilnosti procesa sortiranja, koja je omogućena ubacivanjem elementa u red na njegov kraj. Prema tome, kada se u *i*-tom koraku vrši sortiranje po cifri d_{i-1}, veći je onaj element koji ima višu cifru na toj poziciji i on ide u odgovarajući red. Elementi sa istom cifrom idu u isti red, ali su oni zbog stabilnosti metoda već uređeni po nižim ciframa. Ovo omogućava spajanje redova bez vođenja računa o njihovim granicama.
Prilikom implementacije algoritma treba voditi računa o tome da se broj elemenata u redovima u pojedinim koracima ne može predvideti. Zato sekvencijalna realizacija redova nije pogodna, već treba pribeći ulančanoj reprezentaciji. Redovi se održavaju kao ulančane liste u koje se novi elementi stavljaju na kraj. Zbog toga, za svaki red treba još obezbediti pokazivače koji pokazuju na početak i kraj liste. Na početku se od neuređenog niza napravi jedna lista iz koje se uzimaju elementi i stavljaju u redove. Posle svakog koraka sve liste se opet nadovezuju u jednu listu, počevši od liste koja odgovara cifri 0 do liste koja odgovara cifri 9.

Performanse

Vremenska složenost metoda očigledno zavisi od broja cifara *k* i od broja elemenata *n*. Kako se u svakom od *k* koraka obrađuju svi elementi, vremenska složenost je reda $O(kn)$. Ako je broj cifara *k* konstantan, onda je složenost linearna. Prema tome, ovo je vrlo efikasan metod za kratke ključeve. Ukoliko su ključevi dugački, performansa opada, pa se tada može usvojiti jedno hibridno rešenje. Princip *radix* sortiranja se može primeniti samo na manji broj najstarijih cifara, što neće potpuno sortirati niz, ali će ga dosta dobro urediti. Završno sortiranje se može obaviti nekim metodom koji je efikasan za prilično uređene nizove, kao što je, na primer, direktno umetanje.
Ako *k* nije konstanta već raste sa *n*, onda se povećava i vremenska složenost. Na primer, ako su ključevi gusti, pa skup ključeva pokriva skoro čitav skup mogućih vrednosti, onda se *k* približava log *n*, a složenost se približava $O(n \log n)$.

```
/* program Radix;*/
#include <stdio.h>
#include <math.h>
 typedef int niz[100];

 int cifra(int a,int mesto)
  {   int i;
   for(i=1;i<=mesto-1;i++)a=a/10;   return(a%10);
   }

 void prolaz(niz a,int n,int mesto)
  {   int pc[10];       niz pomocni;   int i,d;
   for(d=0;d<=9;d++)pc[d]=0;
   for(i=1;i<=n;i++)pc[cifra(a[i],mesto)]+=1;
```

```
/* program Radixz1;*/
#include<stdio.h>
int cifra(int a,int mesto)
 {
  int i;
  for (i=1;i<=mesto-1;i++) a/=10;
  return(a%10);
 }
void prolaz(int a[],int n,int mesto)
{
  int pc[11],i,d,j,k;int pomocni[10][100];
  for (d=0;d<=9;d++)
   {
     pc[d]=0; for (i=1;i<=n;i++) pomocni[d][i]=0;
```

```
  for(d=1;d<=9;d++)pc[d]=pc[d]+pc[d-1];
  for(i=n;i>=1;i--)
     {pomocni[pc[cifra(a[i],mesto)]]=a[i];
      pc[cifra(a[i],mesto)]-=1;
     }
  for(i=1;i<=n;i++)a[i]=pomocni[i];
  }

 void radixSort(niz a,int N)
  { int p;
  for(p=1;p<=5;p++)prolaz(a, N, p);
// pod pretpostavkom INT_MAX 32767
  }


 main()
  {niz x;   int i,n;
   printf("Unesite broj clanova niza: \n");
  scanf("%d",&n);
  printf("Unesite niz ");
  for(i=1;i<=n;i++)scanf("%d",&x[i]);
  radixSort(x,n);
  for(i=1;i<=n;i++)printf("%d ",x[i]);
  }
```

```
  }
 for (i=1;i<=n;i++)
  {
    pc[cifra(a[i],mesto)]++;
    pomocni[cifra(a[i],mesto)][pc[cifra(a[i],mesto)]]=a[i];
  }
 k=0;
for(i=0;i<=9;i++)
 {
   if (pc[i]>0)
    for(j=1;j<=pc[i];j++)
     {
      k++; a[k]=pomocni[i][j];
     }
 }
}
void radixsort(int a[],int n)
{
 int p;
 for (p=1;p<=5;p++) prolaz(a,n,p);
}
main()
{
int i,n;
int x[100];int y[9][100];
scanf("%d",&n);
for (i=1;i<=n;i++) scanf("%d",&x[i]);
radixsort(x,n);
for (i=1;i<=n;i++) printf("%d ",x[i]);
}
```

Zadaci:

1. Sortirati RADIX SORT-om trocifrene brojeve: 329, 457, 657, 839, 436, 720, 355.

Rešenje:

329		720		720		329	
457		355		329		355	
657		436		436		436	
839	→	457	→	839	→	457	
436		657		355		657	
720		329		457		720	
355		839		657		839	

2. Sortirati RADIX SORT-om niz 36,394,9,0,25,381,1,49,38,194,94,16,43,50,81,4.
Prolaz 1

Klasa	0	1	2	3	4	5	6	7	8	9
	0	381		43	394	25	36		38	9
	50	1			194		16			49
		81			94					

					4					

Nakon 1. prolaza, spajanjem klasa(kutija) se dobija niz 0,50,381,1,81,43,394,194,94,4,25,36,16,38,9,49

Prolaz 2

Klasa	0	1	2	3	4	5	6	7	8	9
Sadržaj	0	16	25	36	43	50		-	381	394
	1			38	49				81	194
	4									94
	9									

Nakon 2. prolaza, spajanjem klasa(kutija) se dobija niz :0,1,4,9,16,25,36,38,43,49,50, 381,81, 394,194,94,.

Prolaz 3 (poslednji prolaz, jer su brojevi najviše 3-cifreni)

Klasa	0	1	2	3	4	5	9	7	8	9
Sadržaj	0	194	-	381	-	-	-	-	-	-
	1			394						
	4									
	9									
	16									
	25									
	36									
	38									
	43									
	49									
	50									
	81									
	94									

Nakon 3. prolaza, spajanjem klasa(kutija) se dobija niz: 0,1,4,9,16,25,36,38,43,49,50,81,94,194,381,394

3. Dato je k listi i svaka od njih sadrži n elemenata. Ključevi elemenata su celi brojevi iz opsega $[1,m]$. Pokazati kako se mogu sortirati sve liste tako da vremenska složenost u najgorem slučaju bude $O(kn+m)$.

Rešenje:

1. Numerišemo liste brojevima od 1 do k
2. Primenimo algoritam za sortiranje višestrukim razvrstavanjem na sve elemente koristeći ključ koji se sastoji od dva broja – prvi označava listu iz koje potiče element, a drugi je originalni ključ.

Sortiranje po drugom broju je složenosti O(nk+m).

Sortiranje po prvom broju je složenosti O(nk+k) = O(nk).

Dakle, ukupna složenost je $O(kn+m)$.

Primer:

Neka su nam date tri liste (4,3,9), (8,2,1), (7,5,6)

Liste sa novim ključem: ((1)4, (1)3, (1)9), ((2)8, (2)2, (2)1), ((3)7, (3)5, (3)6)

Nakon prvog prolaza (sortiranje po drugom broju): ((2)1, (2)2, (1)3, (1)4 , (3)5, (3)6, (3)7, (2)8, (1)9)

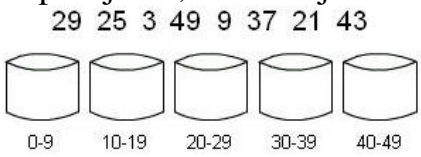
Nakon drugog prolaza (sortiranje po prvom broju) dobijamo: ((1)3, (1)4, (1)9, (2)1 , (2)2, (2)8, (3)5, (3)6, (3)7)

4. Metodom Bucket Sort sortirati brojeve 29, 25, 3, 49, 9, 37, 21, 43.

Rešenje:

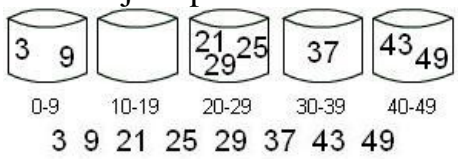
Bucket sort (bin sort) razvrstava elemente niza u korpice. Ovo je, takođe, metoda sortiranja koja nije zasnovana na poređenju (tzv. nekomparativni metod, non-comparison sort).

U prvoj fazi, inicalizuje se niz praznih korpi na 0.



Potom se prolazi kroz niz elemenata i svaki element se stavlja u svoju korpu (vrednost člana niza je adresa korpe).

U svakoj korpi su elementi sortirani i na kraju se elemnti prepisuju iz korpi u originalni niz.




```
#include <stdio.h>
#define NMAX 8

   void bucketSort(int a[]) {
        int i, j;
        int count [7*NMAX]={0};

        for (i = 0; i<NMAX; i++) count[a[i]]++;

        for (i = 0, j = 0; i<7*NMAX; i++)
            for (; count[i] > 0; (count[i])--)
                a[j++] = i;
    }

    int main() {
        int a[] = { 29, 25, 3, 49, 9, 37, 21, 43 }, i;

        bucketSort(a);

        for (i=0; i<NMAX; i++) printf("%d ", a[i]);

        return 0;
    }
```

5. Neka su u prirodnjačkom muzeju prikupljene vredne kolekcije starog kamenja koja se među sosbom razlikuju po težini, boji i sve ove karakteristike su zebeležene u tekstualnoj datoteci.Na, primer plavi kamen težine 23g je u datoteci opisan strukturom {23, plavo}. Vaš program treba da sortira kamenčiće po težini u kategorije (0-10 grama, 11-20 grama, 21-30 grama, 31-40 grama, 41+ grama) koristeći bucket sort. Potom, morate koristiti bucket sort ponovo kako bi razdvojili kamenčiće unutar kategorija po boji i to ovim redosledom:crvena, narandžasta, žuta, zelena, plava, ljubičasta, siva.

6. Dat je niz od n prirodnih brojeva sa više ponavljanja elemenata tako da je broj različitih elemenata u nizu $O(\log n)$.
(a) Konstruisati algoritam za sortiranje ovakvih nizova u kome se izvršava najviše $O(n \log \log n)$ upoređivanja brojeva.
(b) Zbog čega je složenost ovog algoritma manja od donje granice ($n \log n$) za sortiranje niza od n brojeva?

Rešenje 1:

a) Svaki element niza umetnuće se kao jedno polje čvora balansiranog binarnog stabla pretrage. Drugo polje čvora će čuvati broj pojava tog elementa u nizu. Ako je broj različitih elemenata u nizu O(log n), onda je broj čvorova u stablu O(log n),
te je visina stabla O(log log n) => broj upoređivanja je najviše O(n log log n).

Zatim se stablo obilazi inorder obilaskom i njegovi elementi se kopiraju u neopadajućem redosledu u izlazni niz dužine *n* tako što se svaki element kopira onoliko puta kolika je vrednost odgovarajućeg brojačkog polja.
b) Opisani algoritam pod a) nije obuhvaćen modelom stabla odlučivanja, jer je pri konstrukciji algortma bilo od značaj da postoji relativno mali broj različitih elemenata u nizu, tj. vrednosti elemenata niza su bile značajne.

Rešenje 2:

a) Svaki element niza umetnuće se kao jedno polje čvora AVL stabla. Drugo polje čvora će čuvati broj pojava tog elementa u nizu.
Ako je broj različitih elemenata u nizu O(log n), onda je broj čvorova u stablu O(log n),
te je visina stabla O(log log n) => broj upoređivanja je najviše O(n log log n).

Zatim se AVL stablo ispravlja, tj. njegovi elementi se kopiraju u neopadajućem redosledu u izlazni niz dužine *n* tako što se svaki element kopira onoliko puta kolika je vrednost odgovarajućeg brojačkog polja.
b) Opisani algoritam pod a) nije obuhvaćen modelom stabla odlučivanja, jer je pri konstrukciji algortma bilo od značaj da postoji relativno mali broj različitih elemenata u nizu, tj. vrednosti elemenata niza su bile značajne.

7. Pokazati kako je moguće sortirati *n* celih brojeva iz opsega $[0, n^2 - 1]$ u vremenu $O(n)$.
Rešenje:

Broj cifara potrebnih za predstavljanje n^2 različitih brojeva u k-arnom brojnm sistemu je $d=\log_k n^2$
Ako n^2 brojeva posmatramo kao n RADIX brojeva, tako da svaka cifra je u intervalu 0..n-1,
važi da $d= \log_n n^2=2$
Dakle, ove 2-cifrene brojeve možemo sortirati RADIX SORTom.
Otuda, ukupna složenost sortiranja je
$O(d(n+k)) = O(2(n+n)) = O(4n) = O(n)$

8. Da li je tačno: Ako bi umesto counting sort, koristili za sortiranje cifri u algoritmu radix sort, bilo koji algoritam za sortiranje, onda će radix sort i dalje korektno obavljati sortiranje.
Mala pomoć: Stabilnost algoritma sortiranja?

9. Da li je tačno: RADIX SORT i dalje radi korektno čak i ako svaku pojedinačni cifru sortiramo algoritmom HEAPSORT umesto COUNTING SORT.
Rešenje: Ne.
HEAPSORT nije stabilam. Ako je prvih i-1 cifara sortirano u algoritmu RADIX SORT, onda sortiranje i-te cifre koristeći stabilan COUNTING SORT neće promeniti redosled dva broja sa istom i-tom cifrom. Takvo svojstvo ne može da se garantuje za HEAPSORT.

10. Da li je tačno: RADIX-SORT se može koristiti da se sortira n elemenata niza sa celobrojnim ključevima koji uzimaju vrednosti iz skupa od 1 do $n^{f(n)}$ u O(nf(n)) vremenskoj složenosti.
Rešenje: Da
Pomoć: $\log_n n^{f(n)}=f(n)$

11. Ako pretpostavimo da su sve korpe koje se koriste u algoritmu BUCKET SORT iste veličine, onda BUCKET SORT ima prosečnu vremensku složenost O(n) nezavisno od distribucije elemenata na ulazu u algoritam.
Rešenje: Ne.
Ako, na primer, izaberemo distribuciju ulaznih elemenata koji prevazilaze jednu od korpi, onda može da se desi da BUCKET SORT ima vremensku složenost najgoreg slučaja sortiranja u jednoj korpi (npr. $O(n^2)$ za INSERTION SORT te korpe).

12. Dat je niz parova celih brojeva <A[i], B[i]>. Kazemo da je par <A[i], B[i]> manji od <A[j], B[j]> ukoliko vazi jedan od sledeca dva uslova:

1. A[i] < A[j]
2. A[i] = A[j], i < j

Mozemo primetiti da ne postoje dva elemente u nizu koja su jednaka. Vas zadatak je da ispisete ovaj niz parova celih brojeva u rastucem redosledu.

Prvi red standardnog ulaza sadrzi prirodan broj N (1 <= N <= 100 000) koji predstavlja broj elemenata niza A i B.
Drugi red sadrzi N prirodnih brojeva, razdvojenih jednim znakom razmaka, koji predstavljaju elemente niza A.
Naredni red sadrzi N prirodnih brojeva, razdvojenih jednim znakom razmaka, koji predstavljaju elemente niza B.
(-10^9 <= A[i], B[i] <= 10^9)

U prvi red standarnog izlaza ispisati niz A posle sortiranja. U sledeci red ispisati niz B posle sortiranja.
Primer

ULAZ	IZLAZ
5 1 -1 1 92 -55 4 5 1 -1 2	 -55 -1 1 1 92 2 5 4 1 -1

Rešenje:
Nama je potrebno da u ovom zadatku iskoriristimo stabilnost algoritma sortiranja.
U biblioteci STL postoji implementacija stabilnog algoritma za sortiranje **stable_sort** koji garantuje da će elementi sa istom vrednosti biti poredjani onim redosledom u kom su bili pre sortiranja.
Detaljan opis STL biblioteke http://www.cplusplus.com/reference/algorithm/

```
#include <cstdio>
#include <algorithm>
#define MAXN 100000
using namespace std;

struct s {
    int a, b;
} niz[MAXN];

bool cmp(s x, s y) {
    return x.a < y.a;
}

int main() {
    int n;
    scanf("%d", &n);

    for (int i = 0; i < n; i++) scanf("%d", &niz[i].a);
    for (int i = 0; i < n; i++) scanf("%d", &niz[i].b);

       stable_sort(niz, niz+n, cmp);

    for (int i = 0; i < n-1; i++) printf("%d ", niz[i].a);
    printf("%d\n", niz[n-1].a);
    for (int i = 0; i < n; i++) printf("%d ", niz[i].b);
    return 0;
}
```