



# **SIRALAMA ALGORİTMALARI**

# SIRALAMA ALGORİTMALARI

- Sıralama ve arama tekniklerinden pek çok programda yararlanılmaktadır. Günlük yaşamımızda elemanların sıralı tutulduğu listeler yaygın olarak kullanılmaktadır.
- Sıralama, sıralanacak elemanlar bellekte ise internal (içsel), kayıtların bazıları ikincil bellek ortamındaysa external (dışsal) sıralama olarak adlandırılır.

# KABARCIK SIRALAMA (BUBBLE SORT) ALGORİTMASI

- Dizinin elemanları üzerinden ilk elemandan başlayarak ve her geçişte sadece yan yana bulunan iki eleman arasında sıralama yapılır.
- Dizinin başından sonuna kadar tüm elemanlar bir kez işleme tabi tutulduğunda dizinin son elemanı (küçükten büyüğe sıralandığında) en büyük eleman haline gelecektir.

## BUBBLE SORT

- Bir sonraki tarama ise bu en sağdaki eleman dışarıda bırakılarak gerçekleştirilmektedir. Bu dışarıda bırakma işlemi de dış döngüdeki sayaç değişkeninin değerinin her işletimde bir azaltılmasıyla sağlanmaktadır. Sayaç değişkeninin değeri 1 değerine ulaştığında ise dizinin solunda kalan son iki eleman da sıralanmakta ve sıralama işlemi tamamlanmaktadır.
- Bubble sort, sıralama teknikleri içinde anlaşılması ve programlanması kolay olmasına rağmen etkinliği en az olan algoritmalardandır ( $n$  elemanlı  $x$  dizisi için).

## BUBBLE SORT

- Örnek:
- 9, 5, 8, 3, 1. rakamlarının azalan şekilde sıralanmasını kabarcık algoritmasıyla gerçekleştirelim.

# BUBBLE SORT

- 1.Tur:



- 1. tur tamamlandığında en büyük eleman olan 9 en sona yerleşmiş olur ve bir daha karşılaştırmaya gerek yoktur.

# SIRALAMA ALGORİTMALARI-BUBBLE SORT

- 2.Tur:

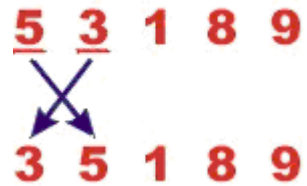
- 



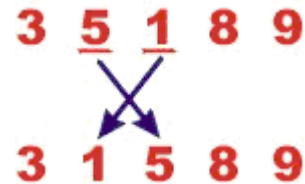
# SIRALAMA ALGORİTMALARI-BUBBLE SORT

- 3.Tur:

5 3 1 8 9  
3 5 1 8 9

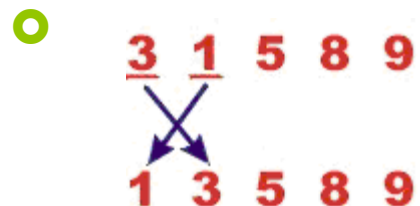


3 5 1 8 9  
3 1 5 8 9



- 4.Tur:

- 3 1 5 8 9  
1 3 5 8 9





## BUBBLE SORT

```
○ public static void bubblesort(int [] x)
○ {   int n = x.Length;   int tut, j, gec;
○   for (gec=0; gec<n-1; gec++)
○       { for(j=0; j<n-gec-1; j++)
○           { if (x[j] > x[j+1])
○               {
○                   tut = x[j];
○                   x[j] = x[j+1];
○                   x[j+1] = tut;
○               }
○           }
○       }
○   }
```

○ en fazla (n-1) iterasyon gerektirir.

## BUBBLE SORT

- Veriler : 25 57 48 37 12 92 86 33
- Tekrar 1 : 25 48 37 12 57 86 33 92
- Tekrar 2 : 25 37 12 48 57 33 86 92
- Tekrar 3 : 25 12 37 48 33 57 86 92
- Tekrar 4 : 12 25 37 33 48 57 86 92
- Tekrar 5 : 12 25 33 37 48 57 86 92
- Tekrar 6 : 12 25 33 37 48 57 86 92
- Tekrar 7 : 12 25 33 37 48 57 86 92

## BUBBLE SORT

- Analizi kolaydır (İyileştirme yapılmamış algorithmada) :
- $(n-1)$  iterasyon ve her iterasyonda  $(n-1)$  karşılaştırma.
- Toplam karşılaştırma sayısı :  $(n-1)*(n-1) = n^2 - 2n + 1 = O(n^2)$
- (Yukarıdaki gibi iyileştirme yapılmış algorithmada etkinlik) :
- iterasyon  $i$ 'de,  $(n-i)$  karşılaştırma yapılacaktır.
- Toplam karşılaştırma sayısı =  $(n-1) + (n-2) + (n-3) + \dots + (n-k)$   
 $= kn - k*(k+1)/2$   
 $= (2kn - k^2 - k)/2$
- ortalama iterasyon sayısı,  $k$ ,  $O(k.n)$  olduğundan  $= O(n^2)$

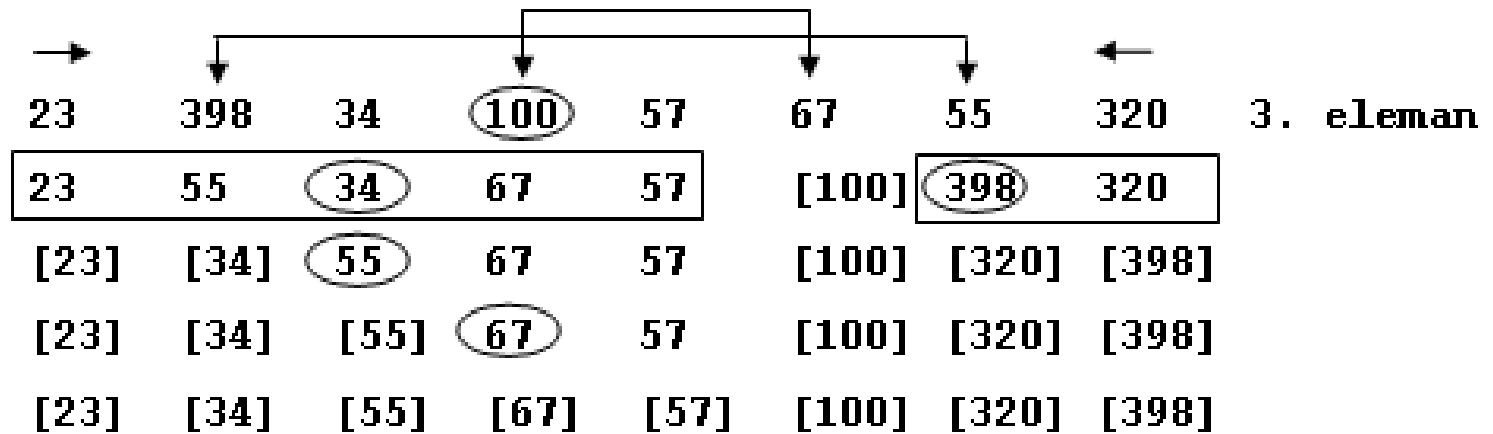
## BUBBLE SORT

- Performans:
- Kabarcık sıralama algoritması ortalama  $N^2/2$  karşılaştırma ve  $N^2/2$  yer değiştirme işlemi gerçekleştirir ve bu işlem sayısı en kötü durumda da aynıdır.

## HIZLI SIRALAMA (QUICKSORT)

- **n** elemanlı bir dizi sıralanmak istendiğinde dizinin herhangi bir yerinden  $x$  elemanı seçilir (örnek olarak ortasındaki eleman).  $x$  elemanı  $j$ . yere yerleştğinde  $0$ . ile  $(j-1)$ . yerler arasındaki elemanlar  $x$ 'den küçük,  $j+1$ 'den  $(n-1)$ 'e kadar olan elemanlar  $x$ 'den büyük olacaktır. Bu koşullar gerçekleştirildiğinde  $x$ , dizide en küçük  $j$ . elemandır. Aynı işlemler,  $x[0]-x[j-1]$  ve  $x[j+1]-x[n-1]$  alt dizileri (parçaları) için tekrarlanır. Sonuçta veri grubu sıralanır.

# QUICKSORT



## QUICKSORT

	0	4	8	12	16	20	24	28
○	23	398	34	100	57	67	55	320
○	23	398	34	100	57	67	55	320
○	23	55	34	100	57	67	398	320
○	23	55	34	100	57	67	398	320
○	23	55	34	100	57	67	398	320
○	23	55	34	67	57	100	398	320

# QUICKSORT

○	0	4	8	12	16	20	24	28
○	23	55	34	67	57	100	398	320
○	23	34	55	67	57	100	320	398
○	23	34	55	67	57	100	320	398
○	23	34	55	57	67	100	320	398
○	23	34	55	57	67	100	320	398



## QUICKSORT

- Eğer şanslı isek, seçilen her eleman ortalanca değere yakınsa  $\log_2 n$  iterasyon olacaktır =  **$O(n \log_2 n)$** . Ortalama durumu işletim zamanı da hesaplandığında  $O(n \log_2 n)$ 'dir, yani genelde böyle sonuç verir. En kötü durumda ise parçalama dengesiz olacak ve  $n$  iterasyonla sonuçlanacağında  **$O(n^2)$**  olacaktır (en kötü durum işletim zamanı).

## QUICKSORT – C code

- Pivot orta elaman seçildi
- `#include <stdio.h>`
- `void qsort2 (double *left, double *right){`
- `double *p = left, *q = right, w, x=*(left+(right-left>>1));`
- `/* ilk aders ile son adres deęerinin farkını alıp 2 ye böldükten sonra çıkan deęeri  
tekrar ilk adresle topluyor.adres deęerlerini görmek için`
- `printf("%f ",*(left+(right-left>>1)));`
- `printf("r %d\n",right); printf("l %d\n",left);*/`
- `do {`
- `while(*p<x) p++;`
- `while(*q>x) q--;`
- `if(p>q) break;`
- `w = *p; *p = *q; *q = w;`
- `} while(++p <= --q);`
- `if(left<q) qsort2(left,q);`
- `if(p<right) qsort2(p,right); }`
- `void main(){`
- `double dizi[8] = { 23, 398, 34, 100, 57, 67, 55, 320 };`
- `qsort2 ( &dizi[0], &dizi[7] );`
- `for(int i=0;i<8;i++) printf("%f ",dizi[i]);}`

## QUICKSORT – C# Code

- Başlangıçta Pivot en son elaman seçildi
- `public static void QuickSort(int[] input, int left, int right)`
- `{ if (left < right)`
- `{ int q = Partition(input, left, right);`
- `QuickSort(input, left, q - 1); QuickSort(input, q + 1, right); }`
- `}`
- `private static int Partition(int[] input, int left, int right)`
- `{ int pivot = input[right];`
- `int temp; int i = left;`
- `for (int j = left; j < right; j++)`
- `{ if (input[j] <= pivot)`
- `{ temp = input[j]; input[j] = input[i]; input[i] = temp; i++; }`
- `}`
- `input[right] = input[i]; input[i] = pivot;`
- `return i;`
- `}`

## QUICKSORT – C# Code

```
○ static void Main(string[] args)
○ {
○     int [] dizi = { 23, 398, 34, 100, 57, 67, 55, 320 };
○
○     QuickSort(dizi, 0, dizi.Length - 1);
○
○     for(int i=0;i<8;i++) Console.Write(" "+dizi[i]);
○ }
```

## QUICKSORT – Java Code

- Pivot ilk elaman seçildi
- `import TerminalIO.*;`
- `class QuickSort{`
- `public static void main(String []args){`
- `KeyboardReader input=new KeyboardReader();`
- `int n=input.readInt("enter list size : ");`
- `int[] Arr=new int[n];`
- `for(int i=0;i<n;i++)`
- `Arr[i]=input.readInt("enter elements :");`
- `int pivot1;`
- `pivot1=partition(Arr,0, n-1);`
- `System.out.println("Pivot Value is "+pivot1);`
- 
- `quicksort(Arr, 0, n-1);`
- `for(int j=0; j<Arr.length ; j++)`
- `System.out.println(Arr[j]);`
- `}`
- `}`

## QUICKSORT – Java Code

```

○ public static void quicksort(int array[], int left,int right)
○ { int pivot =partition(array, left, right);
○   if (left<pivot)
○     quicksort(array, left, pivot-1);
○   if (right>pivot)
○     quicksort(array, pivot+1, right);
○ }
○ public static int partition(int numbers[],int left,int right)
○ { int l_hold,r_hold,i;   int pivot; l_hold=left;           r_hold=right;
○   pivot=numbers[left];
○   while(left<right){
○     while((numbers[right]>=pivot)&&(left<right)) right--;
○     if(left!=right) {      numbers[left]=numbers[right]; left++;    }
○     while((numbers[left]<=pivot)&&(left<right)) left++;
○     if(left!=right){ numbers[right]=numbers[left]; right--; }
○   }
○   numbers[left]=pivot;           pivot=left;           left=l_hold;
○   right=r_hold;                  return pivot;
○ } }

```

## SEÇMELİ SIRALAMA (SELECTION SORT)

- Dizideki en küçük elemanı bul, bu elemanı dizinin son (yer olarak) elemanı ile yer değiştir.
- Daha sonra ikinci en küçük elemanı bul ve bu elemanı dizinin ikinci elemanı ile yer değiştir. Bu işlemi dizinin tüm elemanları sıralanıncaya kadar sonraki elemanlarla tekrar et.
- Elemanların seçilerek uygun yerlerine konulması ile gerçekleştirilen bir sıralamadır :

## SEÇMELİ SIRALAMA (SELECTION SORT)

- Örnek: 9, 5, 8, 3, 1. rakamlarının azalan şekilde sıralanmasını seçmeli sıralama algoritmasıyla gerçekleştirelim. (Küçük olanı bul)
- 1. Tur:
- 9 5 8 3 1   9 5 8 3 1   9 5 8 3 1   9 5 8 3 1
- 9 5 8 3 1   9 5 8 3 1   9 5 8 3 1   9 5 8 3 1
- 9 5 8 3 1
- 1 5 8 3 9



## SEÇMELİ SIRALAMA (SELECTION SORT)

- 2. Tur:
- 1 5 8 3 9    1 5 8 3 9    1 5 8 3 9
- 1 5 8 3 9    1 5 8 3 9    1 3 8 5 9
- 3. Tur:
- 1 3 8 5 9    1 3 8 5 9
- 1 3 8 5 9    1 3 5 8 9
- 4. Tur:
- Sıralama tamam

## SEÇMELİ SIRALAMA (SELECTION SORT)-java

```
○ public static int [] selectionsort(int [] A,int n)
○ {
○     int tmp;    int min;
○
○     for(int i=0; i < n-1; i++)
○     {
○         min=i;
○
○         for(int j=i; j < n-1; j++)
○         {
○             if (A[j] < A[min]) {    min=j;    }
○         }
○         tmp=A[i];    A[i]=A[min];    A[min]=tmp;
○     }
○     return A;
○ }
```

# SEÇMELİ SIRALAMA (SELECTION SORT) -C

```
○ #include <stdio.h>
○ void selectsort(int x[], int n) {
○     int i, indx, j, large;
○     for(i=0; i<n; i++) {
○         large = x[i];    indx = i;
○         for(j=i+1; j<n; j++)
○             if (x[j] < large) {    large = x[j];    indx = j;    printf("a=%d \n ",x[j]);    }
○
○         x[indx] = x[i];
○         x[i] = large;    }
○     }
○ void main() {
○     int dizi[8] = {25, 57, 48, 37, 12, 92, 86, 33};
○     selectsort( &dizi[0], 8);
○     for(int i=0;i<8;i++) printf("%d\n ",dizi[i]);
○ }
```

## SEÇMELİ SIRALAMA (SELECTION SORT)

- En küçüğe göre sıralama
- Veriler : **25** 57 48 37 **12** 92 86 33
- Tekrar 1 : **12** 57 48 37 **25** 92 86 33
- Tekrar 2 : 12 **25** 48 37 **57** 92 86 33
- Tekrar 3 : 12 25 **33** 37 57 92 86 **48**
- Tekrar 4 : 12 25 33 **37** 57 92 86 48
- Tekrar 5 : 12 25 33 37 **48** 92 86 **57**
- Tekrar 6 : 12 25 33 37 48 **57** 86 **92**
- Tekrar 7 : 12 25 33 37 48 57 **86** 92

Tekrar 8: 12 25 33 37 48 57 86 **92**

- Selection Sort'un analizi doğrudandır.
- turda (n-1),
- turda (n-2),
- ...
- (n-1). Turda 1, karşılaştırma yapılmaktadır.
- Toplam karşılaştırma sayısı  $= (n-1)+(n-2)+\dots+1 = n*(n-1)/2$
- $= (1/2)n^2 - (1/2)n = O(n^2)$

# SEÇMELİ SIRALAMA (SELECTION SORT)

- //Enbüyük elemanı bulup sona atarak selection sıralama

- 

- `#include <stdio.h>`

- `void selectsort(int x[], int n) {`

- `int i, indx, j, large;`

- `for(i=0; i<n; i++) {`

- `large = x[n-i-1]; indx=n-i-1;`

- `for(j=0; j<n-i; j++)`

- `if(x[j]>large) { large = x[j]; indx = j; }`

- `x[indx] = x[n-i-1];`

- `x[n-i-1] = large;`

- `}`

- `}`

- `void main() {`

- `int dizi[8] = {25, 57, 48, 37, 12, 92, 86, 33};`

- `selectsort( &dizi[0], 8);`

- `for(int i=0;i<8;i++) printf("%d\n ",dizi[i]);`

- `}`

-

## SEÇMELİ SIRALAMA (SELECTION SORT)

- En büyüğüne göre sıralama (en sondakini en büyük al)
- Veriler : 25 57 48 37 12 92 86 33
- Tekrar 1 : 25 57 48 37 12 **33** 86 **92**
- Tekrar 2 : 25 57 48 37 12 33 86 92
- Tekrar 3 : 25 33 48 37 12 57 86 92
- Tekrar 4 : 25 33 **12** 37 48 57 86 92
- Tekrar 5 : 25 33 12 37 48 57 86 92
- Tekrar 6 : 25 12 33 37 48 57 86 92
- Tekrar 7 : 12 25 33 37 48 57 86 92

## SEÇMELİ SIRALAMA (SELECTION SORT)

- **Performans :**  
N elemanlı bir dizi için, seçerek sıralama algoritması yaklaşık  $N^2/2$  karşılaştırma ve N yer değiştirme işlemi gerçekleştirmektedir. Bu özelliği seçerek sıralama işlevinin gerçekleştirminden çıkarmak mümkündür.
- Dış döngünün her işletiminde bir tek yer değiştirme işlemi gerçekleştirildiğinden, bu döngü N adet işletildiğinde (N=dizi boyutu) N tane yer değiştirme işlemi gerçekleştirilecektir.
- Bu döngünün her işletiminde ayrıca N-i adet karşılaştırma gerçekleştirildiğini göz önüne alırsak toplam karşılaştırma sayısı  $(N-1)+(N-2)+\dots+2+1 \rightarrow N^2/2$  olacaktır.

## EKLEMELİ SIRALAMA (INSERTION SORT)

- Yerleştirerek sıralama işlevi belirli bir anda dizinin belirli bir kısmını sıralı tutarak ve bu kısmı her adımda biraz daha genişleterek çalışmaktadır. Sıralı kısım işlev son bulunca dizinin tamamına ulaşmaktadır.
- Elemanların sırasına uygun olarak listeye tek tek eklenmesi ile gerçekleştirilen sıralamadır :



# INSERTION SORT

<b>Veriler</b>	25	57	48	37	12	92	86	33
<b>Tekrar 1</b>	25	57	48	37	12	92	86	33
<b>Tekrar 2</b>	25	48	57	37	12	92	86	33
<b>Tekrar 3</b>	25	37	48	57	12	92	86	33
<b>Tekrar 4</b>	12	25	37	48	57	92	86	33
<b>Tekrar 5</b>	12	25	37	48	57	92	86	33
<b>Tekrar 6</b>	12	25	37	48	57	86	92	33
<b>Tekrar 7</b>	12	25	33	37	48	57	86	92

# INSERTION SORT

```
○ void insertsort(int x[], int n)
○ {
○   int i,k,y;
○   for(k=1; k<n; k++)
○   {
○     y=x[k];
○     for(i=k-1; i>=0 && y<x[i]; i--)
○       x[i+1]=x[i];
○     x[i+1]=y;
○   };
○ }
```

# INSERTION SORT

- **Performans :**
- Eğer veriler sıralı ise her turda 1 karşılaştırma yapılacaktır ve  $O(n)$  olacaktır.
- Veriler ters sıralı ise toplam karşılaştırma sayısı:
- $(n-1)+(n-2)+\dots+3+2+1 = n*(n+1)/2 = O(n^2)$  olacaktır.
- Simple Insertion Sort'un ortalama karşılaştırma sayısı ise  $O(n^2)$ 'dir.
- Selection Sort ve Simple Insertion Sort, Bubble Sort'a göre daha etkindir. Selection Sort, Insertion Sort'tan daha az atama işlemi yaparken daha fazla karşılaştırma işlemi yapar.

# INSERTION SORT

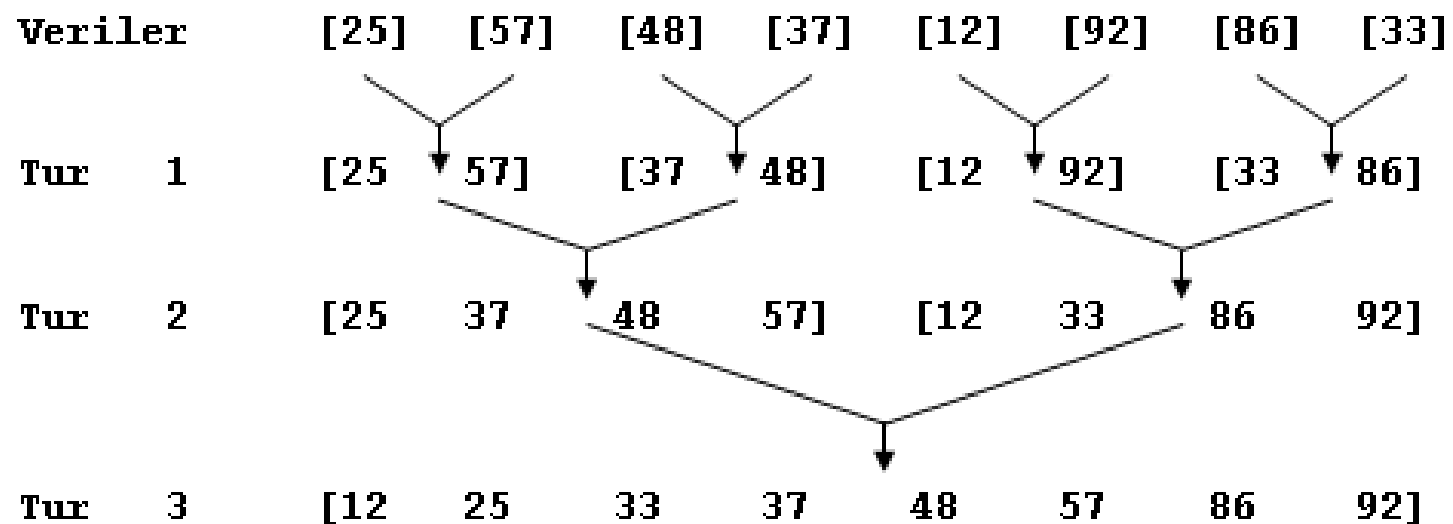
- Bu nedenle Selection Sort, az elemanlı veri grupları için (atamaların süresi çok fazla olmaz) ve karşılaştırmaların daha az yük getireceği basit anahtarlı durumlarda uygundur.
- Tam tersi için, insertion sort uygundur. Elemanlar bağlı listedelerse araya eleman eklemelerde veri kaydırma olmayacağından insertion sort mantığı uygundur.
- $n$ 'in büyük değerleri için quicksort, insertion ve selection sort'tan daha etkindir. Quicksort'u kullanmaya başlama noktası yaklaşık 30 elemanlı durumlardır; daha az elemanın sıralanması gerektiğinde insertion sort kullanılabilir.

## Birleştirmeli Sıralama (MERGE SORT)

- Verinin hafızada sıralı tutulması için geliştirilen sıralama algoritmalarından (sorting algorithms) bir tanesidir.
- Basitçe sıralanacak olan diziyi ikiye elemanı kalan parçalara inene kadar sürekli olarak ikiye böler. Sonra bu parçaları kendi içlerinde sıralayarak birleştirir.
- Sonuçta elde edilen dizi sıralı dizinin kendisidir. Bu açıdan bir parçala fethet (divide and conquer) yaklaşımıdır.

## Birleştirmeli Sıralama (MERGE SORT)

- Sıralı iki veri grubunu birleştirerek üçüncü bir sıralı veri grubu elde etmeye dayanır.



## Birleştirmeli Sıralama (MERGE SORT)

- Sıralanmak istenen verimiz:
- 5,7,2,9,6,1,3,7 olsun.
- Bu verilerin bir oluşumun(composition) belirleyici alanları olduğunu düşünebiliriz. Yani örneğin vatandaşlık numarası veya öğrenci numarası gibi. Dolayısıyla örneğin öğrencilerin numaralarına göre sıralanması durumunda kullanılabilir.
- Birleştirme sıralamasının çalışması yukarıdaki bu örnek dizi üzerinde adım adım gösterilmiştir. Öncelikle parçalama adımları gelir. Bu adımlar aşağıdadır.
- 1. adım diziyi ikiye böl:
- 5,7,2,9 ve 6,1,3,7
- 2. adım çıkan bu dizileri de ikiye böl:
- 5,7 ; 2,9 ; 6,1 ; 3,7

## Birleştirmeli Sıralama (MERGE SORT)

- 3. adım elde edilen parçalar 2 veya daha küçük eleman sayısına ulaştığı için dur (aksi durumda bölme işlemi devam edecekti)
- 4. adım her parçayı kendi içinde sırala
- 5,7 ; 2,9 ; 1,6 ; 3,7
- 5. Her bölünmüş parçayı birleştir ve birleştirirken sıraya dikkat ederek birleştir (1. ve 2. parçalar ile 3. ve 4. parçalar aynı gruptan bölünmüştü)
- 2,5,7,9 ve 1,3,6,7
- 6. adım, tek bir bütün parça olmadığı için birleştirmeye devam et
- 1,2,3,5,6,7,7,9
- 7. adım sonuçta bir bütün birleşmiş parça olduğu için dur. İşte bu sonuç dizisi ilk dizinin sıralanmış halidir.



## MERGE SORT- Java

- Öncelikle birleştirme sıralamasının ana fonksiyonu:
- `public class MergeSort {`
- `private int[] list;`
- `// sıralancak listeyi alan inşa fonksiyonu`
- `public MergeSort(int[] listToSort) {list = listToSort; }`
- `// listeyi döndüren kapsülleme fonksiyonu`
- `public int[] getList() { return list; }`
- `// dışarıdan çağırılan sıralama fonksiyonu`
- `public void sort() { list = sort(list); }`
- `// Özyineli olarak çalışan ve her parça için kullanılan sıralama fonksiyonu`
- `private int[] sort(int[] whole) {`
- `if (whole.length == 1) { return whole; }`
- `else {`
- `// diziyi ikiye bölüyoruz ve solu oluşturuyoruz`
- `int[] left = new int[whole.length/2];`
- `System.arraycopy(whole, 0, left, 0, left.length);`

## MERGE SORT- Java

- //dizinin sağıını oluşturunuz ancak tek sayı ihtimali var
- `int[] right = new int[whole.length-left.length];`
- `System.arraycopy(whole, left.length, right, 0, right.length);`
- `// her iki tarafı ayrı ayrı sıralıyoruz`
- `left = sort(left);`
- `right = sort(right);`
- `// Sıralanmış dizileri birleştiriyoruz`
- `merge(left, right, whole);`
- `return whole;`
- `}`
- `}`

## MERGE SORT- Java

- // birleştirme fonksiyonu
- private void merge(int[] left, int[] right, int[] result) {
- int x = 0;   int y = 0;   int k = 0;
- // iki dizide de eleman varken
- while (x < left.length && y < right.length)
- {   if (left[x] < right[y]) { result[k] = left[x];     x++;   } }
- else {     result[k] = right[y];     y++;   } }
- k++;
- }
- int[] rest;   int restIndex;
- if (x >= left.length) {     rest = right;   restIndex = y;     } }
- else {   rest = left;     restIndex = x;     } }
- for (int i=restIndex; i<rest.length; i++) {   result[k] = rest[i];   k++; }
- }

## MERGE SORT- Java

- `public static void main(String[] args) {`
- `int[] arrayToSort = {15, 19, 4, 3, 18, 6, 2, 12, 7, 9, 11, 16};`
- `System.out.println("Unsorted:");`
- `for(int i = 0; i < arrayToSort.length ; i++){`
- `System.out.println(arrayToSort[i] + " ");`
- `}`
- `MergeSort sortObj = new MergeSort(arrayToSort);`
- `sortObj.sort();`
- `System.out.println("Sorted:");`
- `int [] sirali = sortObj.getList();`
- `for(int i = 0; i < sirali.length ; i++){`
- `System.out.println(sirali[i] + " ");`
- `}`
- `}`
- `}`

## MERGE SORT- C

```

○ #include <conio.h>
○ #define Boyut 8
○ void mergesort(int x[], int n) {
○     int aux[Boyut], i,j,k,L1,L2,size,u1,u2,tur=0; size = 1;
○     while(size<n) {
○         L1 = 0; tur++; k = 0; printf("Tur Sayısı:%d\n",tur); getch();
○         while(L1+size<n) {
○             L2 = L1+size; u1 = L2-1; u2 = (L2+size-1<n) ? L2+size-1 : n-1;
○             for(i=L1,j=L2; i<=u1 && j<=u2; k++){
○                 if(x[i]<=x[j]) aux[k]=x[i++]; else aux[k]=x[j++];          printf("aux[]={%d\n",aux[k]);
○                 }
○             for(;i<=u1;k++) { aux[k] = x[i++]; printf("aux[]={%d\n",aux[k]);}
○             for(;j<=u2;k++) {aux[k] = x[j++]; printf("aux[]={%d\n",aux[k]);}
○             L1 = u2+1;
○             }
○             for(i=L1;k<n;i++){ aux[k++] = x[i]; printf("aux[]={%d\n",aux[k]);}
○             for(i=0;i<n;i++) x[i]=aux[i];
○             size*=2;
○         }
○     }
○ void main() {
○     int dizi[8] = { 25,57, 48, 37, 12, 33, 86,92 }; mergesort( &dizi[0], 8);
○     for(int i=0;i<8;i++) printf("%d\n ",dizi[i]);
○ }

```

## MERGE SORT

- Analiz :  $\log_2 n$  tur ve her turda  $n$  veya daha az karşılaştırma =  $O(n \log_2 n)$  karşılaştırma.
- Quicksort'ta en kötü durumda  $O(n^2)$  karşılaştırma gerektiği düşünülürse daha avantajlı. Fakat mergesort'ta atama işlemleri fazla ve dizi için daha fazla yer gerekiyor.

## Yığın Sıralaması (HEAP SORT)

- Her düğümün çocuk düğümlerinin kendisinden küçük veya eşit olma kuralını esas alır.
- Sıralama yapısı; dizinin ilk elemanı her zaman en büyük olacaktır.
- Dizi üzerinde  $i$ . elemanla, çocukları  $2i$ . ve  $(2i+1)$  karşılaştırılıp büyük olan elemanlar yer değiştirilecektir.

## HEAP SORT

- Dizinin son elamanları dizinin ortasındaki elamanların **çocuk düğümü** olacağından bu işlem dizinin yarısına kadar yapılır.
- Elde edilen diziye sıralamak için ise dizinin ilk elamanı en büyük olduğu bilindiğinden dizinin son elamanıyla ilk elemanı yer değiştirilerek büyük elaman sona atılır.
- Bozulan diziye yukarıdaki işlemler tekrar uygulanır. Dizi boyutu 1 oluncaya kadar işleme devam edilir.



## HEAP SORT

- Bu algoritmanın çalışma zamanı,  $O(n \log n)$ 'dir.
- En kötü durumda en iyi performansı garanti eder.
- Fakat karşılaştırma döngülerinden dolayı yavaş çalışacaktır.

# HEAP SORT

- `heapify(dizi, i)`
- `{ L = Left(i); R = Right(i);`
- `if (L <= heapsize && x[L] > x[i])`
- `largest = L;`
- `else largest = i;`
- `if (R <= heapsize && x[R] > x[largest])`
- `largest = R;`
- `if (largest != i)`
- `{temp = x[largest]; x[largest] = x[i];`
- `x[i] = temp; heapify(x, largest);`
- `}`
- `}`

# HEAP SORT

- BuildHeap(dizi)
- {
- heapsize = dizi.Length - 1;
- for ( i = 0; i < dizi.Length/2; i++)
- heapify(dizi, i);
- }
  
- Left(i)   { return 2 \* (i + 1) - 1; }
- Right(i) { return 2\*(i+1); }
- Parent(i){ return i/2; }

## HEAP SORT

- HeapExtract\_or\_Sort(dizi)
- {
- BuildHeap(y);
- for (i = heapsize; i >=0; i--)
- {         temp = dizi[0];
- dizi[0] = dizi[i];
- dizi[i] = temp;
- heapsize--;
- heapify(dizi, 0);
- }
- }

## SIRALAMA ALGORİTMALARI-SHELL SORT

- Shell algoritması etkin çalışması ve kısa bir program olmasından dolayı günümüzde en çok tercih edilen algoritmalarından olmuştur. h adım miktarını kullanarak her defasında dizi elemanlarını karşılaştırarak sıralar.
- **Insertion sort** sıralamanın geliştirilmesiyle elde edilmiştir.
- **Azalan artış sıralaması olarak da adlandırılır**. Insertion sort'un aksine ters sıralı dizi dağılımından etkilenmemektedir. Algoritmanın analizi zor olduğundan çalışma zamanları net olarak çıkarılamamaktadır. Aşağıda verilen h dizisi için  $n^{1,5}$  karşılaştırmadan fazla karşılaştırma yapmayacağından dolayı  $\theta(n^{1,25})$ ,  $O(n^{1,5})$  olarak belirlenmiştir.

## Kabuk sıralaması(SHELL SORT)

```

○ void shellsort ( ) {
○   int dizi[5] = {9, 5, 8, 3, 1 }; int n=5; int h=1;
○   while ((h*3+1<n)) h=3*h+1;
○     while (h>0)
○     {
○       for(int i=h-1;i<n;i++)
○       {
○         int b=dizi[i];          int j=i;
○         for(j=i;(j>=h)&&(dizi[j-h]>b);j-=h) dizi[j]=dizi[j-h];
○         dizi[j]=b;
○       }
○     }
○     h/=3;
○   }
○   for(int i=0;i<5;i++) printf("%d\n ",dizi[i]);
○ }
○ void main() { shellsort(); }

```

## SIRALAMA ALGORİTMALARI-SHELL SORT

- 1.Tur,  $h=4$ ,  $i=3$ ,  $i=4$
- 
- 9 5 8 3 1      9 5 8 3 1
- 9 5 8 3 1      1 5 8 3 9
- 2.Tur  $h=1$ ,  $i=0$ ,  $i=1$ ,  $i=2$ ,  $i=3$ ,  $i=4$
- 1 5 8 3 9 1 5 3 8 9 1 3 5 8 9
- 1 5 3 8 9 1 3 5 8 9 1 3 5 8 9

## Basamağa göre sıralama (Radix Sort)

- Sayıları basamaklarının üzerinde işlem yaparak sıralayan bir sıralama algoritmasıdır.
- Şimdiye dek gördüğümüz sıralama algoritmalarının zamana göre etkinlikleri  $O(n^2)$  ya da  $O(n \log n)$  ile ölçülüyordu. Radix Sort, çoğunlukla bu ikisi arasında bir etkinliğe sahiptir.
- Her sıralama algoritmasında olduğu gibi, Radix Sort algoritmasının etkinliği de sıralanacak diziye bağlı olarak değişkenlik gösterir. Ortalama olarak, Etkinlik derecesi  $O(n)$  dir. Dolayısıyla,  $O(n^2)$  grubuna dahil olan sıralama algoritmalarından daha iyidir.



## Basamağa göre sıralama (Radix Sort)

- Sayma sayıları adlar ya da tarihler gibi karakter dizilerini göstermek için kullanılabildiği için *basamağa göre sıralama* algoritması yalnızca sayma sayılarını sıralamak için kullanılan bir algoritma değildir.
- Çoğu bilgisayar veri saklamak için ikilik tabandaki sayıların elektronikteki gösterim biçimlerini kullandığı için sayma sayılarının basamaklarını ikilik tabandaki sayılardan oluşan öbekler biçiminde göstermek daha kolaydır.

## Basamağa göre sıralama (Radix Sort)

- Basamağa göre sıralama algoritması en anlamlı basamağa göre sıralama ve en anlamsız basamağa göre sıralama olarak ikiye ayrılır.
- **En anlamsız basamağa** (Least significant digit) göre sıralama algoritması sayıları en anlamsız (en küçük, en sağdaki) basamaktan başlayıp en anlamlı basamağa doğru yürüyerek sıralarken *en anlamlı basamağa göre sıralama* bunun tam tersini uygular.
- Sıralama algoritmaları tarafından işlenen ve kendi sayı değerlerini gösterebildiği gibi başka tür verilerle de eşleştirilebilen sayma sayılarına çoğu zaman "anahtar" denir.

## Basamağa göre sıralama (Radix Sort)

- En anlamsız basamağa göre sıralamada kısa anahtarlar uzunlardan önce gelirken aynı uzunluktaki anahtarlar sözlükteki sıralarına göre sıralanırlar. Bu sıralama biçimi sayma sayılarının kendi değerlerine göre sıralandıklarında oluşan sırayla aynı sırayı oluşturur.
- Örneğin 1'den 10'a kadar olan sayılar sıralandığında ortaya 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 dizisi çıkacaktır.

## Basamağa göre sıralama (Radix Sort)

- **En anlamlı basamağa göre sıralama** sözcükler ya da aynı uzunluktaki sayılar gibi dizgileri sıralamak için uygun olan sözlükteki sıraya göre sıralar.
- Örneğin "b, c, d, e, f, g, h, i, j, ba" dizisi sözlük sırasına göre "b, ba, c, d, e, f, g, h, i, j" olarak sıralanacaktır. Eğer sözlük sırası değişken uzunluktaki sayılarda uygulanırsa sayılar değerlerinin gerektirdiği konumlara konulmazlar.

## Basamağa göre sıralama (Radix Sort)

A	00001	R	10010	T	10100	X	11000	P	10000	A	00001
S	10011	T	10100	X	11000	P	10000	A	00001	A	00001
O	01111	N	01110	P	10000	A	00001	A	00001	E	00101
R	10010	X	11000	L	01100	I	01001	R	10010	E	00101
T	10100	P	10000	A	00001	A	00001	S	10011	G	00111
I	01001	L	01100	I	01001	R	10010	T	10100	I	01001
N	01110	A	00001	E	00101	S	10011	E	00101	L	01100
G	00111	S	10011	A	00001	T	10100	E	00101	M	01101
E	00101	O	01111	M	01101	L	01100	G	00111	N	01110
X	11000	I	01001	E	00101	E	00101	X	11000	O	01111
A	00001	G	00111	R	10010	M	01101	I	01001	P	10000
M	01101	E	00101	N	01110	E	00101	L	01100	R	10010
P	10000	A	00001	S	10011	N	01110	M	01101	S	10011
L	01100	M	01101	O	01111	O	01111	N	01110	T	10100
E	00101	E	00101	G	00111	G	00111	O	01111	X	11000

## Basamağa göre sıralama (Radix Sort)

- Örneğin 1'den 10'a kadar olan sayılar sıralandığında, algoritma kısa olan sayıların sonuna boş karakter koyarak bütün anahtarları en uzun anahtarla aynı boyuta getireceğinden sonuç 1, 10, 2, 3, 4, 5, 6, 7, 8, 9 olacaktır.
- Taban sıralama algoritmasının en basit hali (iyileştirilmiş (optimized)) aşağıdaki örnekte gösterilmektedir:
- Sıralanmamış sayılarımız :57 43 24 213 44 102 70 37 111 23
  - a. İlk geçişte sayıları birler basamağına göre artan yönde sıralıyoruz. Birler hanesinde aynı değeri alan 43, 213, 23 gibi sayılar, başlangıç dizisindeki veriliş sırasıyla yazılırlar.
  - 70 111 102 43 213 23 24 44 57 37

## Basamağa göre sıralama (Radix Sort)

- **b.** İkinci geçişte, ilk geçişte elde edilen diziyi onlar basamağındaki değerlerine göre sıralıyoruz.
- 102 111 213 23 24 37 43 44 57 70
- **c.** Üçüncü geçişte, ikinci geçişte elde edilen diziyi yüzler basamağındaki değerlerine göre sıralıyoruz.
- 23 24 37 43 44 57 70 102 111 213
- Görüldüğü gibi, en soldaki basamağa göre geçiş bitince, sayılar sıralanmış oluyor.
- Bilgisayar sayıları bellekte 2 tabanına göre tuttuğu için, Radix Sort algoritmasını 2 tabanına göre yazmak, 10 tabanından 2 tabanına dönüşüm zamanının kazandıracaktır.

## Basamağa göre sıralama (Radix Sort)

- Yukarıdaki sayıları 2 tabanına göre yazalım:
- 57      111001
- 43      101011
- 24      110000
- 70      1000110
- 111     1101111
- 102     1100110
- 213    11010101
- 37      100101
- 44      101100
- 23      10111



## Basamağa göre sıralama (Radix Sort)

- Şimdi sırayla şu işleri yapalım:
- İlk geçişte sayıları  $2^0$  (birler) basamağına (en sağdaki basamak) göre,
- ikinci geçişte öncekini  $2^1$  (ikiler) basamağına göre,
- üçüncü geçişte öncekini  $2^2$  (dörtler) basamağına göre,
- dördüncü geçişte öncekini  $2^3$  (sekizler) basamağına göre,
- beşinci geçişte öncekini  $2^4$  (onaltılar) basamağına göre,
- altıncı geçişte öncekini  $2^5$  (otuzikiler) basamağına göre,
- yedinci geçişte öncekini  $2^6$  (altmış dörtler) basamağına göre,
- sekizinci geçişte öncekini  $2^7$  (yüz yirmi sekizler) basamağına göre sıralayalım.
- Son geçişte dizi sıralanmış olur.

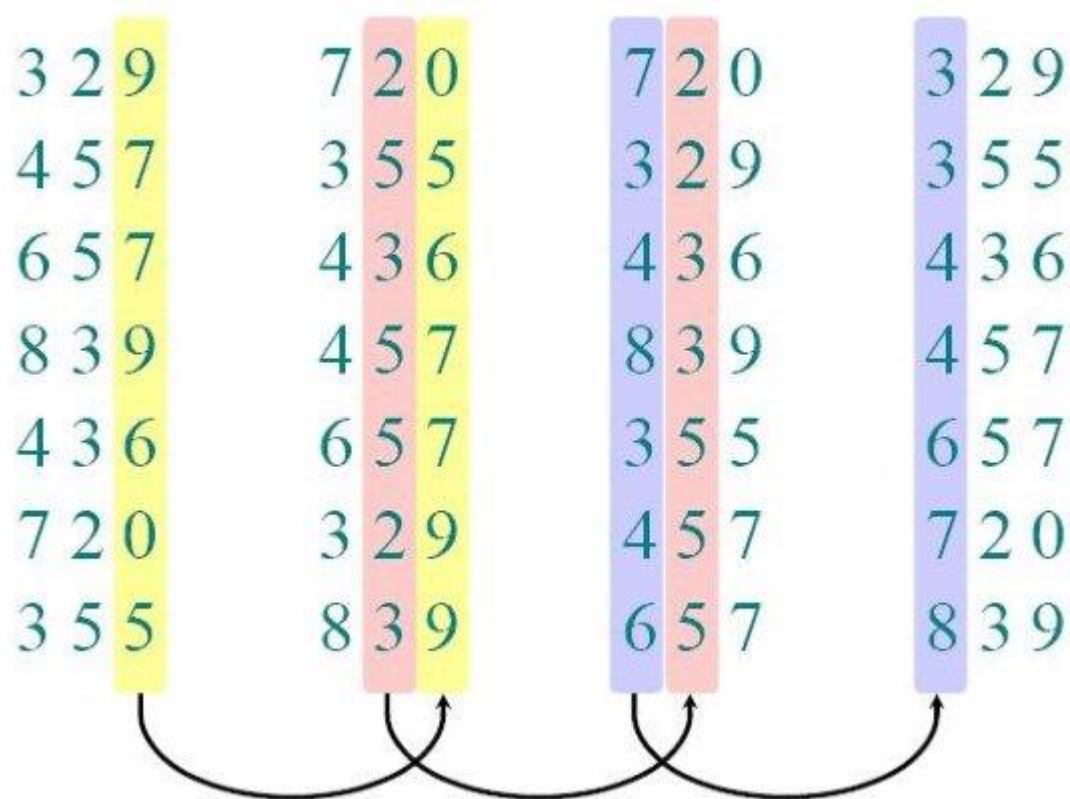
# Basamağa göre sıralama (Radix Sort)

- Bu geçişler aşağıdaki tabloda gösterilmiştir.

2 <sup>0</sup> (birler) basamağına göre sıralama	2 <sup>1</sup> (ikiler) basamağına göre sıralama	2 <sup>2</sup> (dörtler) basamağına göre sıralama	2 <sup>3</sup> (sekizler) basamağına göre sıralama
24      11000 70      1000110 102     1100110 44      101100 57      111001 43      101011 111     1101111 213     11010101 37      100101 23      10111	24      11000 44      101100 57      111001 213     11010101 37      100101 70      1000110 102     1100110 43      101011 111     1101111 23      10111	24      11000 57      111001 43      101011 44      101100 213     11010101 37      100101 70      1000110 102     1100110 111     1101111 23      10111	213     11010101 37      100101 70      1000110 102     1100110 23      10111 24      11000 57      111001 43      101011 44      101100 111     1101111
2 <sup>4</sup> (onaltılar) basamağına göre sıralama	2 <sup>5</sup> (otuzikiler) basamağına göre sıralama	2 <sup>6</sup> (altmışdörtler) basamağına göre sıralama	2 <sup>7</sup> (yüz yirmi sekizler) basamağına göre sıralama
37      100101 70      1000110 102     1100110 43      101011 44      101100 111     1101111 23      10111 24      11000 213     11010101 57      111001	70      1000110 23      10111 24      11000 213     11010101 37      100101 102     1100110 43      101011 44      101100 111     1101111 57      111001	23      10111 24      11000 37      100101 43      101011 44      101100 57      111001 70      1000110 213     11010101 102     1100110 111     1101111	23      10111 24      110000 37      100101 43      101011 44      101100 57      111001 70      1000110 102     1100110 111     1101111 213     11010101

## Basamağa göre sıralama (Radix Sort)

● Örnek:



## Basamağa göre sıralama (Radix Sort)

- Örneğin 10'luk sayı tabanında her sayıdan birer tane bulunması halinde her hane için 10 ihtimal bulunur. Bu her ihtimalin ayrı bir hafıza bölümünde (örneğin bir dizi veya bağlı liste) tutulması durumunda sıralama **işlemi en büyük hane sayısı \* n** olmaktadır.
- Örneğin 3 haneli sayılar için  **$O(3n) \sim O(n)$**  olmaktadır. Bu değer zaman verimliliği (time efficiency) arttırırken hafıza verimliliğini (memory efficiency) azaltmaktadır.
- En kötü etkinlik zamanı  **$O(n^2)$**  ile  **$O(n \log n)$**  arası

# SIRALAMA ALGORİTMALARI

Çalışma Süresi	Sorting Algoritması-en iyi
$O(n^2)$	BubbleSort SelectionSort InsertionSort ShellSort
$O(n \log n)$	HeapSort MergeSort QuickSort
$O(n)$	RadixSort BucketSort

# Basamağa göre sıralama (Radix Sort)

```
import java.lang.*;
import java.io.*;

public class RadixSort {

    public static void radixSort(int[] arr) {
        if (arr.length == 0)
            return;
        int[][] np = new int[arr.length][2];
        int[] q = new int[0x100];
        int i, j, k, l, f = 0;
        for (k = 0; k < 4; k++) {
            for (i = 0; i < (np.length - 1); i++)
                np[i][1] = i + 1;
            np[i][1] = -1;
            for (i = 0; i < q.length; i++)
                q[i] = -1;
            for (f = i = 0; i < arr.length; i++) {
                j = ((0xFF << (k << 3)) & arr[i]) >> (k << 3);
                if (q[j] == -1)
                    l = q[j] = f;
                else {
                    l = q[j];
                    while (np[l][1] != -1)

```

Char type,

0 in decimal is 0x0	in hexadecimal
7 in decimal is 0x7	in hexadecimal
8 in decimal is 0x8	in hexadecimal
9 in decimal is 0x9	in hexadecimal
10 in decimal is 0xa	in hexadecimal
11 in decimal is 0xb	in hexadecimal
12 in decimal is 0xc	in hexadecimal
13 in decimal is 0xd	in hexadecimal
14 in decimal is 0xe	in hexadecimal
15 in decimal is 0xf	in hexadecimal
16 in decimal is 0x10	in hexadecimal
17 in decimal is 0x11	in hexadecimal
31 in decimal is 0x1f	in hexadecimal
32 in decimal is 0x20	in hexadecimal
255 in decimal is 0xff	in hexadecimal
256 in decimal is 0x100	in hexadecimal

```

        l = np[l][1];
        np[l][1] = f;
        l = np[l][1];
    }
    f = np[f][1];
    np[l][0] = arr[i];
    np[l][1] = -1;
}
for (l = q[i = j = 0]; i < 0x100; i++)
    for (l = q[i]; l != -1; l = np[l][1])
        arr[j++] = np[l][0];
}

}

public static void main(String[] args) {
    int i;
    int[] arr = new int[15];
    System.out.print("Sırasız dizi: ");
    for (i = 0; i < arr.length; i++) {
        arr[i] = (int) (Math.random() * 1024);
        System.out.print(arr[i] + " ");
    }
    radixSort(arr);
    System.out.print("\nSıralı dizi : ");
    for (i = 0; i < arr.length; i++)
        System.out.print(arr[i] + " ");
}
}

```

# Basamağa göre sıralama (Radix Sort)

## JAVA

### ○ RadixSort.java

- /\* Copyright (c) 2012 the authors listed at the following URL, and/or
- the authors of referenced articles or incorporated external code:
- [http://en.literateprograms.org/Radix\\_sort\\_\(Java\)?action=history&offset=20080201073641](http://en.literateprograms.org/Radix_sort_(Java)?action=history&offset=20080201073641)
- Permission is hereby granted, free of charge, to any person obtaining
- a copy of this software and associated documentation files (the
- "Software"), to deal in the Software without restriction, including
- without limitation the rights to use, copy, modify, merge, publish,
- distribute, sublicense, and/or sell copies of the Software, and to
- permit persons to whom the Software is furnished to do so, subject to
- the following conditions:
- The above copyright notice and this permission notice shall be
- included in all copies or substantial portions of the Software.
- 
- THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND,
- EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
- MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT.
- IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY
- CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT,
- TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE
- SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.
- Retrieved from: [http://en.literateprograms.org/Radix\\_sort\\_\(Java\)?oldid=12461](http://en.literateprograms.org/Radix_sort_(Java)?oldid=12461)
- \*/



## Basamağa göre sıralama (Radix Sort) JAVA

```

○ import java.util.Arrays;
○ class RadixSort
○ {   public static void radix_sort_uint(int[] a, int bits)
○   {   int[] b = new int[a.length];    int[] b_orig = b;    int rshift = 0;
○       for (int mask = ~(1 << bits); mask != 0; mask <= bits, rshift += bits)   {
○           int[] cntarray = new int[1 << bits];
○           for (int p = 0; p < a.length; ++p)
○           {   int key = (a[p] & mask) >> rshift;           ++cntarray[key];           }
○
○           for (int i = 1; i < cntarray.length; ++i)           cntarray[i] += cntarray[i-1];
○           for (int p = a.length-1; p >= 0; --p)
○           {   int key = (a[p] & mask) >> rshift;   --cntarray[key];   b[cntarray[key]] = a[p];   }
○           int[] temp = b; b = a; a = temp;
○       }
○       if (a == b_orig)           System.arraycopy(a, 0, b, 0, a.length);
○   }

```

# Basamağa göre sıralama (Radix Sort)

## JAVA

```
○ public static void main(String[] args)
○ {
○     int[] a = {
○         123,432,654,3123,654,2123,543,131,653,123,
○         533,1141,532,213,2241,824,1124,42,134,411,
○         491,341,1234,527,388,245,1992,654,243,987};
○
○     System.out.println("Before radix sort:");
○     System.out.println(Arrays.toString(a));
○
○     radix_sort_uint(a, 4);
○
○     System.out.println("After radix sort:");
○     System.out.println(Arrays.toString(a));
○ }
○ }
```

# Ödev

- 1-Verilen sıralama algoritmalarının C# veya Java programlarını yazınız. Algoritmaları tur olarak programda karşılaştırınız.
- 2- Kullanılan diğer sıralama algoritmaları hakkında bilgi toplayıp program kodları ile birlikte slayt hazırlayınız.

## Basamağa göre sıralama (Radix Sort)- C++

- `#define NUMELTS 100`
- `# include<stdio.h>`
- `#include<conio.h>`
- `#include<math.h>`
- `void radixsort(int a[],int);`
- `void main() { int n,a[20],i;`
- `printf("enter the number :"); scanf("%d",&n);`
- `printf("ENTER THE DATA -");`
- `for(i=0;i<n;i++) { printf("%d. ",i+1); scanf("%d",&a[i]); }`
- `radixsort(a,n); getch(); }`

## Basamağa göre sıralama (Radix Sort)

- `void radixsort(int a[],int n) {`
- `int rear[10],front[10],first,p,q,k,i,y,j,exp1; double exp;`
- `struct { int info; int next; } node[NUMELTS];`
- `for(i=0;i<n-1;i++)`
- `{ node[i].info=a[i]; node[i].next=i+1; }`
- `node[n-1].info=a[n-1];`
- `node[n-1].next=-1;`
- `first=0;`

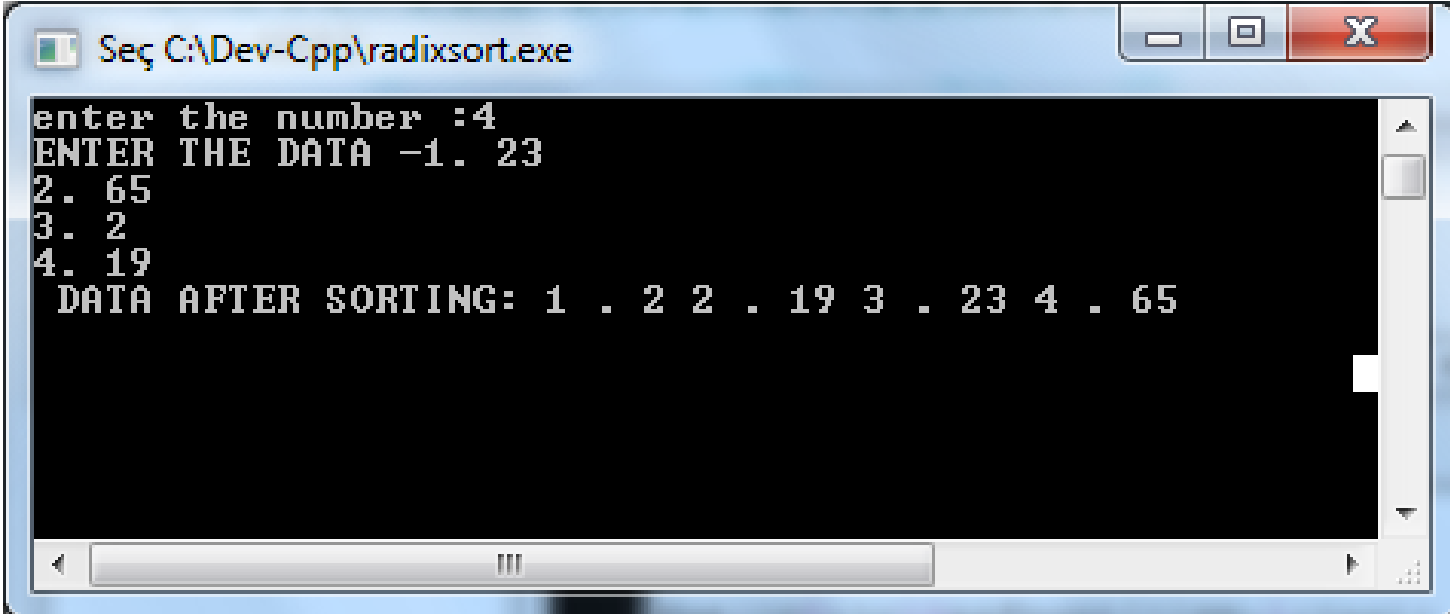
## Basamağa göre sıralama (Radix Sort)

- `for(k=1;k<=2;k++) //consider only 2 digit number`
- `{ for(i=0;i<10;i++)`
- `{ front[i]=-1; rear[i]=-1; }`
- `while(first!=-1) {`
- `p=first; first=node[first].next;`
- `y=node[p].info; exp=pow(10,k-1);`
- `exp1=(int)exp; j=(y/exp1)%10;`
- `q=rear[j];`
- `if(q== -1) front[j]=p;`
- `else`
- `node[q].next=p;`
- `rear[j]=p; }`

## Basamağa göre sıralama (Radix Sort)

- `for(j=0;j<10&&front[j]==-1;j++);`
- `first=front[j];`
- `while(j<=9) {`
- `for(i=j+1;i<10&&front[i]==-1;i++);`
- `if(i<=9) { p=i; node[rear[j]].next=front[i]; }`
- `j=i; }`
- `node[rear[p]].next=-1; }`
- `//copy into original array`
- `for(i=0;i<n;i++) {a[i]=node[first].info; first=node[first].next;}`
- `printf(" DATA AFTER SORTING:");`
- `for(i=0;i<n;i++) printf(" %d . %d",i+1,a[i]); }`

## Basamağa göre sıralama (Radix Sort)



```
Seç C:\Dev-Cpp\radixsort.exe
enter the number :4
ENTER THE DATA -1. 23
2. 65
3. 2
4. 19
DATA AFTER SORTING: 1 . 2 2 . 19 3 . 23 4 . 65
```



# Basamağa göre sıralama (Radix Sort)

## C++

- `public void RadixSort()`
  - `{ // our helper array`
    - `int[] t = new int[Dizi.Length];`
  - `// number of bits our group will be long`
    - `int r = 4; // try to set this also to 2, 8 or 16 to see if it is quicker or not`
  - `// number of bits of a C# int`
    - `int b = 32;`
  - `// counting and prefix arrays`
    - `// (note dimensions  $2^r$  which is the number of all possible values of a r-bit number)`
      - `int[] count = new int[1 << r];`
      - `int[] pref = new int[1 << r];`
  - `// number of groups`
    - `int groups = (int)Math.Ceiling((double)b / (double)r);`
  - `// the mask to identify groups`
    - `int mask = (1 << r) - 1;`
  -

# Basamağa göre sıralama (Radix Sort)

## C++

- // the algorithm:
 

```

for (int c = 0, shift = 0; c < groups; c++, shift += r)
{
    // reset count array
    for (int j = 0; j < count.Length; j++)
        count[j] = 0;

```
- // counting elements of the c-th group
 

```

for (int i = 0; i < Dizi.Length; i++)
    count[(Dizi[i] >> shift) & mask]++;

```
- // calculating prefixes
 

```

pref[0] = 0;
for (int i = 1; i < count.Length; i++)
    pref[i] = pref[i - 1] + count[i - 1];

```
- // from Dizi[] to t[] elements ordered by c-th group
 

```

for (int i = 0; i < Dizi.Length; i++)
    t[pref[(Dizi[i] >> shift) & mask]++] = Dizi[i];

```
- // Dizi[]=t[] and start again until the last group
 

```

t.CopyTo(Dizi, 0);
} // a is sorted
}

```