

SIRALAMA SORTING

Yrd. Doç. Dr. Aybars UĞUR

Giriş

Sıralama ve arama tekniklerinden pek çok programda yararlanılmaktadır. Günlük yaşamımızda elemanların sıralı tutulduğu listeler yaygın olarak kullanılmaktadır. Telefon rehberindeki bir kişinin telefon numarasının bulunması bir arama (search) işlemidir. Telefon rehberlerindeki kayıtlar kişilerin soyadlarına göre sıralıdır. Bir telefon numarasının kime ait olduğunu bulmaya çalışmak da bir başka arama işlemidir. Eğer telefon rehberi kişilerin soyadlarına göre alfabetik olarak değil de telefon numaralarına göre kronolojik olarak sıralı olursa bu, arama işlemini basitleştirir. Kütüphanelerdeki kitapların özel yöntemlere göre sıralanarak raflara dizilmesi, bilgisayarın belleğindeki sayıların sıralanması da yapılacak arama işlemlerini hızlandırır ve kolaylaştırır. Genel olarak eleman toplulukları, bilgisayarda da, telefon rehberi gibi örneklerde de daha etkin erişmek (aramak ve bilgi getirmek) üzere sıralanır ve sıralı tutulurlar.

Anahtar; İçsel ve Dışsal Sıralama

- Eleman (kayıt, yapı ...) toplulukları genelde (her zaman değil) bir anahtara göre sıralı tutulurlar. Bu anahtar genelde elemanların bir alt alanı yani üyesidir (Elemanlar soyada göre sıralı olursa soyadı anahtardır, numaraya göre sıralı olursa numara anahtardır, nota göre olursa not alanı anahtardır). Elemanlar topluluğu içindeki her elemanın anahtar değeri kendinden önce gelen elemanın anahtar değerinden büyükse artan sırada, küçükse azalan sırada sıralıdır denilir (ilgili anahtara göre).
- Sıralama, sıralanacak elemanlar bellekte ise internal (içsel), kayıtların bazıları ikincil bellek ortamındaysa external (dışsal) sıralama olarak adlandırılır.

Sıralama ve Arama

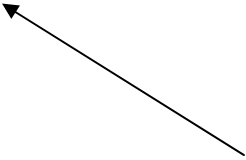
Sıralama ve arama arasında bir ilişki olduğundan bir uygulamada ilk soru sıralama gerekli midir? olmalıdır. Arama işlemleri yoğun ise, sıralamanın veya o şekilde tutmanın getireceği yük, sıralı olmayan kayıtlar üzerindeki arama işlemlerinin getireceği toplam yük yanında çok hafif kalacaktır. Bu karar verilirse, arkasından sıralamanın nasıl yapılacağı ve hangi sıralama yöntemlerinin kullanılacağı kararlaştırılmalıdır. Bunun nedeni, tüm diğer yöntemlerden üstün evrensel bir sıralama tekniğinin olmamasındandır.

SIRALAMA TEKNİKLERİNİN ETKİNLİKLERİ ve ANALİZİ

1) Bubble Sort (Kabarcık Sıralama)

(Exchange Sorts kapsamında)

```
void bubble(int x[], int n)
{
    int hold, j, pass; int switched = TRUE;
    for (pass=0; pass<n-1 && switched == TRUE; pass++)
    {
        switched = FALSE;
        for(j=0; j<n-pass-1; j++)
        {
            if (x[j] > x[j+1])
            {
                switched = TRUE;
                hold = x[j];
                x[j] = x[j+1];
                x[j+1] = hold;
            }
        }
    }
}
```



Bubble sort, sıralama teknikleri içinde anlaşılması ve programlanması kolay olmasına rağmen etkinliği en az olan algoritmalardandır

(n elemanlı x dizisi için) :

Bubble Sort'un Zaman Karmaşıklığının Hesaplanması

En fazla (n-1) iterasyon gerektirir.

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 : Önceki turda değişen eleman olmadığından yapılmaz.

Analizi kolaydır (İyileştirme yapılmamış algoritmada) :

(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 algoritmada etkinlik) :

iterasyon i'de, (n-i) karşılaştırma yapılacaktır.

Toplam karşılaştırma sayısı = $(n-1)+(n-2)+(n-3)+...+(n-k)$

= $kn - k*(k+1)/2$

= $(2kn - k^2 - k)/2$

VERİ YAPILARI : 11 SIRALAMA (SORTING)

ortalama iterasyon sayısı, k, O(n) olduğundan = $O(n^2)$

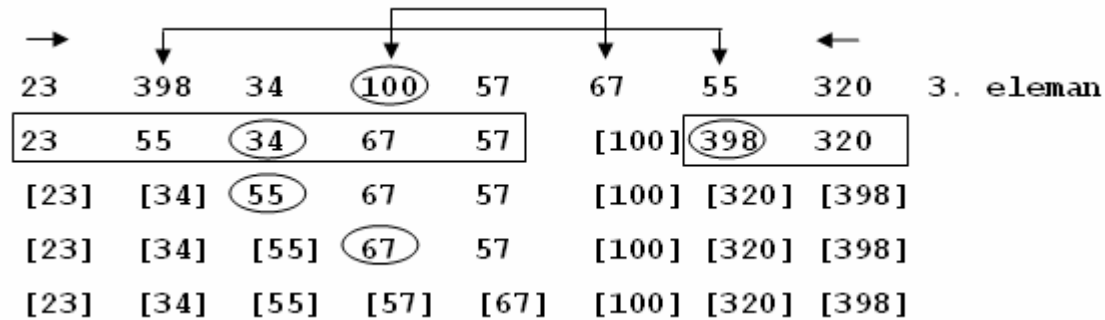
2) Quick Sort (Hızlı Sıralama) (Exchange Sorts kapsamında)

```
#include <stdio.h>
void qsort2(double *left, double *right)
{
    double *p = left, *q = right, w, x=*(left+(right-left>>1));
    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] );
}
```


Quick Sort'un Zaman Karmaşıklığının Hesaplanması

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.



Eğer şanslı isek, seçilen her eleman ortanca 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 sonuçlanacağına $O(n^2)$ olacaktır (en kötü durum işletim zamanı).

3) Straight Selection Sort (Seçmeli Sıralama) (Selection Sorts kapsamında)

Elemanların seçilerek uygun yerlerine konulması ile gerçekleştirilen bir sıralamadır :

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

```
{  
  int i, indx, j, large;  
  for(i=n-1; i>0; i--)  
  {  
    large = x[0];  
    indx = 0;  
    for(j=1; j<=i; j++)  
      if(x[j]>large)  
      {  
        large = x[j];  
        indx = j;  
      };  
    x[indx] = x[i];  
    x[i] = large;  
  };  
}
```

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

Selection Sort'un analizi doğrudandır.

1.turda (n-1),

2.turda (n-2),

3....

(n-1). Turda 1, karşılaştırma yapılmaktadır.

Toplam karşılaştırma sayısı

$$= (n-1) + (n-2) + \dots + 1 = n * (n-1) / 2$$

$$= (1/2)n^2 - (1/2)n = O(n^2)$$

4) Insertion Sort (Eklemeli Sıralama)

(Insertion Sorts kapsamında)

Elemanların sırasına uygun olarak listeye tek tek eklenmesi ile gerçekleştirilen sıralamadır :

```
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;  
    };  
}
```

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

Sıralama Algoritmalarının Karşılaştırılması

Eğer veriler sıralı ise her turda 1 karşılaştırma yapılacaktır ve $O(n)$ olacaktır. Veriler ters sıralı ise toplam karşılaştırma sayısı:
 $(n-1)+(n-2)+...+3+2+1 = n*(n+1)/2 = O(n^2)$ 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. Bu nedenle Selection Sort büyük kayıtlardan oluşan 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.

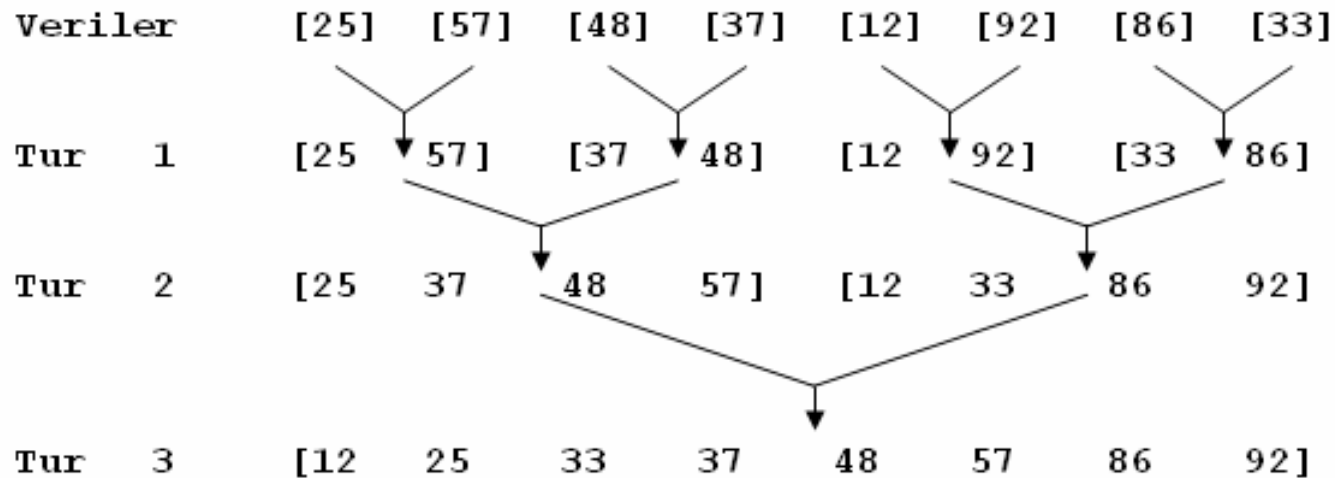
n'in büyük değerleri için quicksort insertion ve selection sort'tan daha etkindir. Quicksort'u kullanmaya başlama noktası yaklaşık 30 elemanlı durumlardır; daha az elemanın sıralanması gerektiğinde insertion sort kullanılabilir.

5) Merge Sort (Birleştirmeli Sıralama)

(Merge Sorts kapsamında)

```
#include <stdio.h>
#define numelts 8
void mergesort(int x[], int n)
{
    int aux[numelts], i,j,k,l1,l2,size,u1,u2;
    size = 1;
    while(size<n) {
        l1 = 0;
        k = 0;
        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++];
            for(;i<=u1;k++) aux[k] = x[i++];
            for(;j<=u2;k++) aux[k] = x[j++];
            l1 = u2+1;
        };
        for(i=l1;k<n;i++)
            aux[k++] = x[i];
        for(i=0;i<n;i++)
            x[i]=aux[i];
        size*=2;
    };
}
```

Merge Sort'un Zaman Karmaşıklığı



- **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 aux dizi için daha fazla yer gerekiyor.**

Java Koleksiyonları ile Dizi Sıralama

```
import java.util.*;

class Sorting
{
    public static void main(String args[])
    {
        // Dizilerin Sıralanması

        String dizi[] = { "Telefon", "Buzdolabı", "Fırın", "Süpürge" };
        Arrays.sort(dizi);

        for(String s : dizi)
            System.out.println(s);

        double sayilar[] = { 2.3, 1.1, 3.5, -1 };
        Arrays.sort(sayilar);

        for(double sayi : sayilar)
            System.out.println(sayi);
    }
}
```

Buzdolab ²
F ² r ² n
S ³ p ³ rge
Telefon
-1.0
1.1
2.3
3.5

Java Koleksiyonları ile Liste Sıralama

```
import java.util.*;

class Sorting2
{
    public static void main(String args[])
    {
        LinkedList<Integer> liste = new LinkedList<Integer>();
        liste.add(5); liste.add(3); liste.add(7);

        Collections.sort(liste);

        for(Integer i : liste)
            System.out.println(i);
    }
}

// Ekran Çıktısı
3
5
7
```


Collections kullanarak Heap Sort

```
import java.util.*;

public class HeapSort
{
    public static void main(String args[])
    {
        Vector<Integer> liste = new Vector<Integer>();
        liste.add(3);
        liste.add(1);
        liste.add(5);
        liste.add(9);
        liste.add(7);
        Collection <Integer> listeS = heapSort(liste);

        Iterator iterator;
        iterator = listeS.iterator();
        while (iterator.hasNext())
        { System.out.print(iterator.next()+" "); }
        System.out.println();
    }
}
```

```
public static <E> List<E> heapSort(Collection<E> c) {
    Queue<E> queue = new PriorityQueue<E>(c);
    List<E> result = new ArrayList<E>();
    while (!queue.isEmpty())
        result.add(queue.remove());
    return result;
}
```

1 3 5 7 9
