# COM2067/ COM267

Bölüm 14: Arama ve Sıralama
**C Kullanarak Veri Yapıları, İkinci Baskı**

- Aranıyor
- Sıralama

# ARAMA GİRİŞİ

- Arama, belirli bir değerin dizi içerisinde bulunup bulunmadığını bulmak anlamına gelir.
- Eğer değer dizi içerisinde mevcutsa aramanın başarılı olduğu söylenir ve arama işlemi o değerin dizi içerisindeki yerini verir.
- Ancak değer dizi içerisinde mevcut değilse arama işlemi uygun bir mesaj görüntüler ve bu durumda aramanın başarısız olduğu söylenir.
- Dizi elemanlarını aramanın iki popüler yöntemi vardır: doğrusal arama ve ikili arama.
- Kullanılması gereken algoritma tamamen değerlerin dizide nasıl organize edildiğine bağlıdır.

# Doğrusal Arama

- Doğrusal arama, sıralı arama olarak da adlandırılır, bir dizide belirli bir değeri aramak için kullanılan çok basit bir yöntemdir.
- Bir eşleşme bulunana kadar aranacak değerin dizinin her elemanıyla tek tek karşılaştırılarak çalışması sağlanır.
- Doğrusal arama çoğunlukla sıralanmamış eleman listelerinde (veri elemanlarının sıralanmadığı dizilerde) arama yapmak için kullanılır.
- Örneğin, bir dizi A[], int A[] = {10, 8, 2, 7, 3, 4, 9, 1, 6, 5}; olarak bildirilir ve başlatılırsa ve aranacak değer VAL = 7 ise, arama, '7' değerinin dizide bulunup bulunmadığını bulmak anlamına gelir.
  - Eğer evet ise, o zaman oluşumunun konumunu döndürür. Burada, POS = 3 (indeks 0'dan başlar).

# Doğrusal Arama

- Şekil 14.1 doğrusal arama algoritmasını göstermektedir.
- Algoritmanın 1. ve 2. Adımlarında POS ve I değerlerini başlatıyoruz.
- Adım 3'te, I, N'den (dizideki toplam eleman sayısı) küçük olana kadar yürütülecek bir while döngüsü yürütülür.
- 4. Adımda, geçerli dizi öğesi ile VAL arasında bir eşleşme bulunup bulunmadığı kontrol edilir.
  - Bir eşleşme bulunursa, dizi öğesinin konumu yazdırılır, aksi takdirde I değeri artırılarak bir sonraki öğeyle VAL değeri eşleştirilir.
  - Ancak, tüm dizi elemanları VAL ile karşılaştırıldıysa ve hiçbir eşleşme bulunamadıysa, bu VAL'in dizide bulunmadığı anlamına gelir.

```
LINEAR_SEARCH(A, N, VAL)
Step 1: [INITIALIZE] SET POS = -1
Step 2: [INITIALIZE] SET I = 1
Step 3:     Repeat Step 4 while I<=N
Step 4:           IF A[I] = VAL
                        SET POS = I
                        PRINT POS
                        Go to Step 6
                  [END OF IF]
                   SET I = I + 1
            [END OF LOOP]
Step 5: IF POS = -1
        PRINT "VALUE IS NOT PRESENT
        IN THE ARRAY"
        [END OF IF]
Step 6: EXIT
```

**Figure 14.1** Algorithm for linear search

# Doğrusal Arama

- ***Doğrusal Arama Algoritmasının Karmaşıklığı***
- Doğrusal arama O(n) sürede yürütülür, burada n dizideki eleman sayısını ifade eder.
  - Açıkçası, doğrusal aramanın en iyi durumu VAL'in dizinin ilk elemanına eşit olduğu durumdur.
    - Bu durumda yalnızca bir karşılaştırma yapılacaktır.
  - Benzer şekilde en kötü durum, VAL'in dizide olmaması veya dizinin son elemanına eşit olması durumunda ortaya çıkar.
    - Her iki durumda da n adet karşılaştırma yapılması gerekecektir.

# Doğrusal Arama

## Programming Example

1. Write a program to search an element in an array using the linear search technique.

```
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
#define size 20 // Added so the size of the array can be altered more easily
int main(int argc, char *argv[]) {
        int arr[size], num, i, n, found = 0, pos = -1;
        printf("\n Enter the number of elements in the array : ");
        scanf("%d", &n);
        printf("\n Enter the elements: ");
        for(i=0;i<n;i++)
        {
                scanf("%d", &arr[i]);
        }
        printf("\n Enter the number that has to be searched : ");
        scanf("%d", &num);
        for(i=0;i<n;i++)
        {
                if(arr[i] == num)
                {
                        found =1;
                        pos=i;
                        printf("\n %d is found in the array at position= %d", num,i+1);
                        /* +1 added in line 23 so that it would display the number in
      the first place in the array as in position 1 instead of 0 */
                        break;
                }
        }
        if (found == 0)
        printf("\n %d does not exist in the array", num);
        return 0;
}
```

# İkili Arama

- İkili arama, sıralı bir listeyle etkili bir şekilde çalışan bir arama algoritmasıdır.
  - Sözlükte kelimeleri nasıl buluruz? Önce sözlüğün ortasında bir yeri açarız. Sonra, o sayfadaki ilk kelimeyi, anlamını aradığımız istenilen kelimeyle karşılaştırırız. Eğer istenilen kelime sayfadaki kelimeden önce geliyorsa, sözlüğün ilk yarısına bakarız, aksi takdirde ikinci yarısına bakarız. Yine, sözlüğün ilk yarısında bir sayfa açarız ve o sayfadaki ilk kelimeyi istenilen kelimeyle karşılaştırırız ve sonunda kelimeyi bulana kadar aynı işlemi tekrarlarız.

# İkili Arama

- Şimdi bu mekanizmanın sıralı bir dizide değer aramak için nasıl uygulandığını ele alalım.
- A[] dizisini şu şekilde beyan edip başlattığımızı düşünün:
- int A[] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
- ve aranacak değer VAL = 9'dur.
- Algoritma şu şekilde ilerleyecektir.
  - BAŞLANGIÇ = 0, SON = 10, ORTA = (0 + 10)/2 = 5
  - Şimdi, VAL = 9 ve A[MID] = A[5] = 5
  - A[5], VAL'den küçüktür, bu nedenle şimdi dizinin ikinci yarısındaki değeri ararız.
  - Yani BEG ve MID değerlerini değiştiriyoruz.
  - Şimdi, BEG = MID + 1 = 6, END = 10, MID = (6 + 10)/2 =16/2 = 8
  - VAL = 9 ve A[ORTA] = A[8] = 8
  - A[8], VAL'den küçüktür, bu nedenle şimdi segmentin ikinci yarısındaki değeri arıyoruz.
  - Yani yine BEG ve MID değerlerini değiştiriyoruz.
  - Şimdi, BEG = MID + 1 = 9, END = 10, MID = (9 + 10)/2 = 9
  - Şimdi, VAL = 9 ve A[MID] = 9.

# İkili Arama

- Bu algoritmada BEG ve END'in elemanını aramak istediğimiz segmentin başlangıç ve bitiş pozisyonları olduğunu görüyoruz.
  - MID, (BEG + END)/2 olarak hesaplanır.
  - Başlangıçta BEG = alt_sınır ve END = üst_sınır.
  - Algoritma A[MID] = VAL olduğunda sonlanacaktır.
  - Algoritma sonlandığında POS = MID yapacağız.
    - POS, değerin dizide bulunduğu konumdur.
  - Ancak, VAL A[MID]'e eşit değilse, BEG, END ve MID değerleri VAL'in A[MID]'den küçük veya büyük olmasına bağlı olarak değişecektir.
    - a) Eğer VAL < A[MID] ise, VAL dizinin sol segmentinde mevcut olacaktır. Bu nedenle, END değeri END = MID – 1 olarak değişecektir.
    - (b) Eğer VAL > A[MID] ise, VAL dizinin sağ segmentinde mevcut olacaktır. Bu nedenle, BEG değeri BEG = MID + 1 olarak değişecektir.
  - Son olarak, eğer VAL dizide mevcut değilse, o zaman sonunda END, BEG'den daha az olacaktır. Bu olduğunda, algoritma sonlanacak ve arama başarısız olacaktır.

# İkili Arama

- Şekil 14.2 ikili arama algoritmasını göstermektedir.
- 1. Adımda BEG, END ve POS değişkenlerinin değerlerini başlatıyoruz.
- Adım 2'de, BEG değeri END değerinden küçük veya eşit olana kadar while döngüsü yürütülür.
- 3. Adımda MID değeri hesaplanır.
- 4. Adımda, MID'deki dizi değerinin VAL'e (dizide aranacak öğe) eşit olup olmadığını kontrol ediyoruz.
  - Eğer bir eşleşme bulunursa POS değeri yazdırılır ve algoritmadan çıkılır.
    - Ancak, bir eşleşme bulunamazsa ve A[MID] değeri VAL'den büyükse, END değeri değiştirilir; aksi takdirde, A[MID] değeri VAL'den büyükse, BEG değeri değiştirilir.
- Adım 5'te POS değeri = -1 ise VAL dizide mevcut değildir ve algoritma çıkmadan önce ekrana uygun bir mesaj yazdırılır.

```
BINARY_SEARCH(A, lower_bound, upper_bound, VAL)
Step 1: [INITIALIZE] SET BEG = lower_bound
        END = upper_bound, POS = - 1
Step 2: Repeat Steps 3 and 4 while BEG <= END
Step 3:           SET MID = (BEG + END)/2
Step 4:           IF A[MID] = VAL
                        SET POS = MID
                        PRINT POS
                        Go to Step 6
                  ELSE IF A[MID] > VAL
                        SET END = MID - 1
                  ELSE
                        SET BEG = MID + 1
                  [END OF IF]
        [END OF LOOP]
Step 5: IF POS = -1
            PRINT "VALUE IS NOT PRESENT IN THE ARRAY"
        [END OF IF]
Step 6: EXIT
```

**Figure 14.2** Algorithm for binary search

# İkili Arama

- ***İkili Arama Algoritmasının Karmaşıklığı***
- İkili arama algoritmasının karmaşıklığı f(n) şeklinde ifade edilebilir; burada n dizideki eleman sayısıdır.
- Algoritmanın karmaşıklığı yapılan karşılaştırma sayısına bağlı olarak hesaplanır.
- İkili arama algoritmasında her karşılaştırmada arama yapılması gereken segmentin boyutunun yarı yarıya azaldığını görüyoruz.
- Bu nedenle, dizide belirli bir değeri bulmak için yapılacak toplam karşılaştırma sayısının şu şekilde verildiğini söyleyebiliriz:
- $2^{f(n)} > n$ veya $f(n) = \log_2 n$

# İkili Arama

**PROGRAMMING EXAMPLE**

2. Write a program to search an element in an array using binary search.

```
##include <stdio.h>
#include <stdlib.h>
#include <conio.h>
#define size 10 // Added to make changing size of array easier
int smallest(int arr[], int k, int n); // Added to sort array
void selection_sort(int arr[], int n); // Added to sort array
int main(int argc, char *argv[]) {
        int arr[size], num, i, n, beg, end, mid, found=0;
        printf("\n Enter the number of elements in the array: ");
        scanf("%d", &n);
        printf("\n Enter the elements: ");
        for(i=0;i<n;i++)
        {
                scanf("%d", &arr[i]);
        }
        selection_sort(arr, n);  // Added to sort the array
        printf("\n The sorted array is: \n");
        for(i=0;i<n;i++)
        printf(" %d\t", arr[i]);
        printf("\n\n Enter the number that has to be searched: ");
        scanf("%d", &num);
        beg = 0, end = n-1;
        while(beg<=end)
        {
                mid = (beg + end)/2;
                if (arr[mid] == num)
                {
                        printf("\n %d is present in the array at position %d", num, mid+1);
                        found =1;
                        break;
                }
```

# İkili Arama

```
                    else if (arr[mid]>num)
                    end = mid-1;
                    else
                    beg = mid+1;
            }
            if (beg > end && found == 0)
            printf("\n %d does not exist in the array", num);
            return 0;
    }

    int smallest(int arr[], int k, int n)
    {
            int pos = k, small=arr[k], i;
            for(i=k+1;i<n;i++)
            {
                    if(arr[i]< small)
                    {
                            small = arr[i];
                            pos = i;
                    }
            }
            return pos;
    }
    void selection_sort(int arr[],int n)
    {
            int k, pos, temp;
            for(k=0;k<n;k++)
            {
                    pos = smallest(arr, k, n);
                    temp = arr[k];
                    arr[k] = arr[pos];
                    arr[pos] = temp;
            }
    }
```

# Enterpolasyon Arama

- Enterpolasyon araması, sıralanmış bir dizide belirtilen bir değeri bulan bir arama tekniğidir.
- Enterpolasyon araması kavramı, bir telefon rehberinde isimleri veya bir kitabın girişlerinin sıralandığı anahtarları arama şeklimize benzer.
  - Örneğin, bir telefon rehberinde “Bharat” ismini aradığımızda, bu ismin en solda olacağını biliyoruz, dolayısıyla listeyi her defasında ikiye bölerek ikili arama tekniğini uygulamak iyi bir fikir değildir.
  - İlk geçişte aşırı solu taramaya başlamalıyız.
- İnterpolasyon aramasının her adımında, bulunacak değer için kalan arama alanı hesaplanır.
- Hesaplama, arama uzayının sınırlarında bulunan değerler ve aranacak değer esas alınarak yapılır.
- Tahmini konumda bulunan değer daha sonra aranan değerle karşılaştırılır.
- Eğer iki değer eşitse arama tamamlanmıştır.

# Enterpolasyon Arama

- Ancak değerlerin eşit olmaması durumunda karşılaştırmaya bağlı olarak kalan arama alanı tahmini konumdan önceki veya sonraki kısma indirgenir.
- Böylece enterpolasyon aramasının ikili arama tekniğine benzediğini görüyoruz.
  - Ancak, iki teknik arasındaki önemli fark, ikili aramanın her zaman kalan arama alanının orta değerini seçmesidir. Tahmini konumda bulunan değer ile aranacak değer arasındaki karşılaştırmaya dayanarak değerlerin yarısını atar.
  - Ancak enterpolasyon aramasında, enterpolasyon, aranan öğeye yakın bir öğeyi bulmak için kullanılır.

# Enterpolasyon Arama

- Enterpolasyon arama algoritması Şekil 14.3'te verilmiştir.

```
INTERPOLATION_SEARCH (A, lower_bound, upper_bound, VAL)

Step 1: [INITIALIZE] SET LOW = lower_bound,
        HIGH = upper_bound, POS = -1
Step 2:    Repeat Steps 3 to 4 while LOW <= HIGH
Step 3:         SET MID = LOW + (HIGH - LOW) ×
                ((VAL - A[LOW]) / (A[HIGH] - A[LOW]))
Step 4:         IF VAL = A[MID]
                  POS = MID
                  PRINT POS
                  Go to Step 6
                 ELSE IF VAL < A[MID]
                   SET HIGH = MID - 1
                  ELSE
                   SET LOW = MID + 1
                [END OF IF]
        [END OF LOOP]
Step 5: IF POS = -1
              PRINT "VALUE IS NOT PRESENT IN THE ARRAY"
        [END OF IF]
Step 6: EXIT
```

**Figure 14.3** Algorithm for interpolation search

# Enterpolasyon Arama

- Şekil 14.4 ikili arama ve enterpolasyon araması durumunda arama alanının nasıl bölündüğünü görselleştirmemize yardımcı olur.

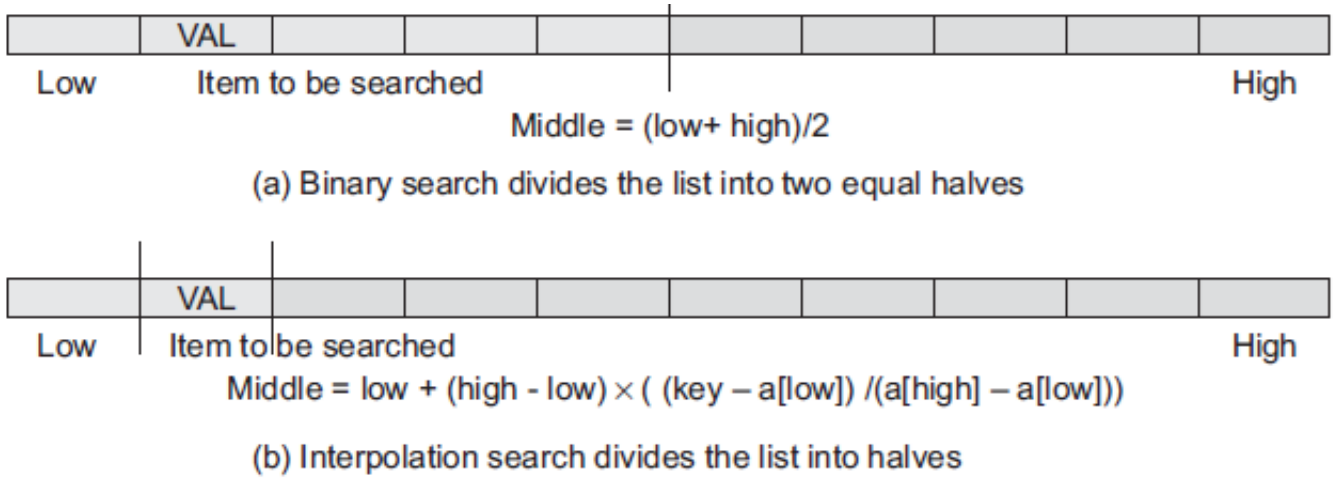



**Figure 14.4** Difference between binary search and interpolation search

# Enterpolasyon Arama

- ***Enterpolasyon Arama Algoritmasının Karmaşıklığı***
- Sıralanacak bir listenin n elemanı düzgün dağılmışsa (ortalama durum), enterpolasyon araması yaklaşık log(log n) karşılaştırma yapar.
- Ancak en kötü durumda, yani elemanlar üstel olarak arttığında, algoritma O(n) adede kadar karşılaştırma yapabilir.

# Enterpolasyon Arama

**Example 14.1** Given a list of numbers `a[] = {1, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21}`. Search for value 19 using interpolation search technique.

***Solution***

```
Low = 0, High = 10, VAL = 19, a[Low] = 1, a[High] = 21
Middle = Low + (High – Low)×((VAL – a[Low]) /(a[High] – a[Low] ))
       = 0 +(10 – 0) × ((19 – 1) / (21 – 1) )
       = 0 + 10 × 0.9 = 9
a[middle] = a[9] = 19 which is equal to value to be searched.
```

# Enterpolasyon Arama

**PROGRAMMING EXAMPLE**

3. Write a program to search an element in an array using interpolation search.

```
#include <stdio.h>
#include <conio.h>
#define MAX 20
int interpolation_search(int a[], int low, int high, int val)
{
        int mid;
        while(low <= high)
        {
                mid = low + (high - low)*((val - a[low]) / (a[high] - a[low]));
                if(val == a[mid])
                      return mid;
                if(val < a[mid])
                      high = mid - 1;
                else
                      low = mid + 1;
        }
        return -1;
}
int main()
{
        int arr[MAX], i, n, val, pos;
        clrscr();
        printf("\n Enter the number of elements in the array : ");
        scanf("%d", &n);
        printf("\n Enter the elements : ");
        for(i = 0; i <n; i++)
                scanf("%d", &arr[i]);
        printf("\n Enter the value to be searched : ");
        scanf("%d", &val);
        pos = interpolation_search(arr, 0, n-1, val);
        if(pos == -1)
                printf("\n %d is not found in the array", val);
        else
                printf("\n %d is found at position %d", val, pos);
        getche();
        return 0;
}
```

# Atla Arama

- Zaten sıralanmış bir listemiz olduğunda, bir değeri aramak için diğer etkili algoritma atlama araması veya blok aramasıdır.
- Atlamalı aramada, istenen değeri bulmak için listedeki tüm elemanları taramak gerekli değildir.
- Biz sadece bir elemanı kontrol ediyoruz ve eğer istenilen değerden küçükse, ileri atlayarak onu takip eden bazı elemanları atlıyoruz.
- Tekrar biraz ileri gidildiğinde eleman kontrol edilir.
  - Eğer kontrol edilen eleman istenilen değerden büyükse, o zaman bir sınır elde ederiz ve istenilen değerin daha önce kontrol edilen eleman ile şu anda kontrol edilen eleman arasında olduğundan emin oluruz.
  - Ancak eğer kontrol edilen eleman aranan değerden küçükse, o zaman tekrar küçük bir sıçrama yapıp işlemi tekrarlıyoruz.
  - Değerin sınırı belirlendikten sonra, değeri ve dizideki konumunu bulmak için doğrusal bir arama yapılır.

# Atla Arama

- Örneğin, bir diziyi ele alalım
- a[] = {1,2,3,4,5,6,7,8,9}.
  - Dizinin uzunluğu 9'dur.
  - Eğer 8 değerini bulmamız gerekiyorsa atlamalı arama tekniği kullanılarak aşağıdaki adımlar gerçekleştirilir.

Step 1: First three elements are checked. Since 3 is smaller than 8, we will have to make a jump ahead

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|

Step 2: Next three elements are checked. Since 6 is smaller than 8, we will have to make a jump ahead

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|

Step 3: Next three elements are checked. Since 9 is greater than 8, the desired value lies within the current boundary

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|

Step 4: A linear search is now done to find the value in the array.

# Atla Arama

- Atlama araması için algoritma Şekil 14.5'te

```
JUMP_SEARCH (A, lower_bound, upper_bound, VAL, N)

Step 1: [INITIALIZE] SET STEP = sqrt(N), I = 0, LOW = lower_bound, HIGH = upper_bound, POS = -1
Step 2: Repeat Step 3 while I < STEP
Step 3:         IF VAL < A[STEP]
                  SET HIGH = STEP - 1
                ELSE
                  SET LOW = STEP + 1
                [END OF IF]
                SET I = I + 1
          [END OF LOOP]
Step 4: SET I = LOW
Step 5: Repeat Step 6 while I <= HIGH
Step 6:           IF A[I] = Val
                          POS =  I
                          PRINT POS
                          Go to Step 8
                  [END OF IF]
                  SET I = I + 1
          [END OF LOOP]
Step 7:   IF POS = -1
                PRINT "VALUE IS NOT PRESENT IN THE ARRAY"
          [END OF IF]
Step 8: EXIT
```

**Figure 14.5** Algorithm for jump search

# Atla Arama

- ***Doğrusal Aramaya Göre Atlamalı Aramanın Avantajı***
  - 1000 elemandan oluşan ve elemanların değerleri 0, 1, 2, 3, 4, ..., 999 olan sıralı bir listemiz olduğunu varsayalım, o zaman sıralı arama tam olarak 674 yinelemede 674 değerini bulacaktır.
  - Ancak atlama aramasıyla aynı değer 44 yinelemede bulunabilir. Bu nedenle atlama araması, sıralanmış bir öğe listesinde doğrusal aramadan çok daha iyi performans gösterir.
- ***İkili Aramaya Göre Atlamalı Aramanın Avantajı***
  - İkili aramanın uygulanması şüphesiz çok kolaydır ve O(log n) karmaşıklığındadır, ancak çok fazla sayıda elemana sahip bir liste söz konusu olduğunda karşılaştırma yapmak için listenin ortasına atlamak iyi bir fikir değildir, çünkü aranan değer listenin başındaysa geriye doğru bir (veya daha fazla) büyük adım atılması gerekir.
  - Bu gibi durumlarda, zıplama araması daha iyi performans gösterir çünkü geriye doğru sadece bir kez hareket etmemiz gerekir. Bu nedenle, geriye doğru zıplama ileriye doğru zıplamadan daha yavaş olduğunda, zıplama araması algoritması her zaman daha iyi performans gösterir.

# Atla Arama

- ***Adım Uzunluğu Nasıl Seçilir?***
  - Atlama arama algoritmasının verimli bir şekilde çalışabilmesi için adım için sabit bir boyut tanımlamamız gerekir.
    - Adım büyüklüğü 1 ise algoritma doğrusal arama ile aynıdır.
    - Şimdi, uygun bir adım boyutu bulmak için, öncelikle listenin boyutu (n) ile adımın boyutu (k) arasındaki ilişkiyi bulmaya çalışmalıyız. Genellikle, k $\sqrt{n}$ olarak hesaplanır.
- ***Atlama Aramasının Daha Fazla Optimizasyonu***
  - Şimdiye kadar az sayıda elemana sahip listelerle uğraşıyorduk. Ancak gerçek dünya uygulamalarında listeler çok büyük olabilir. Bu kadar büyük listelerde değeri listenin başından itibaren aramak iyi bir fikir olmayabilir.
  - Daha iyi bir seçenek, aşağıdaki şekilde gösterildiği gibi aramaya k–inci elemandan başlamaktır.

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 ....

Searching can start from somewhere middle in the list rather than from the beginning to optimize performance.

# Atla Arama

- Ayrıca atlama arama algoritmasının performansını, atlama aramasını tekrar tekrar uygulayarak artırabiliriz.
- Örneğin listenin boyutu 1000000 (n) ise.
  - Atlama aralığı daha sonra şu şekilde olacaktır:
  - $\sqrt{n} = \sqrt{1000000} = 1000$.
  - Şimdi, tanımlanan aralık bile 1000 öğeye sahip ve yine büyük bir liste. Yani, atlama araması yeni bir adım boyutuyla tekrar uygulanabilir
  - $\sqrt{1000} \approx 31$.
  - Böylece istenilen aralık çok sayıda değere sahip olduğunda, atlama arama algoritması daha küçük bir adımla tekrar uygulanabilir.
  - Ancak bu durumda algoritmanın karmaşıklığı artık $O(\sqrt{n})$ olmayacak ve logaritmik bir değere yaklaşacaktır.

# Atla Arama

- ***Atlama Arama Algoritmasının Karmaşıklığı***
- Atlama araması, değerin aralığını bulmak için bir adım boyutu (en uygun olarak √n olacak şekilde seçilir) ile dizi içinde atlayarak çalışır.
- Bu aralık belirlendikten sonra doğrusal arama tekniği kullanılarak değer aranır.
- Bu nedenle, atlama arama algoritmasının karmaşıklığı O(√n) olarak verilebilir.

# SIRALAMANIN GİRİŞİ

- Sıralama, bir dizinin elemanlarını artan veya azalan şekilde ilgili bir sıraya göre düzenlemek anlamına gelir.
- Yani, eğer A bir dizi ise, o zaman A'nın elemanları sıralı bir düzende (artan düzende) şu şekilde düzenlenir:
  A[0] < A[1] < A[2] < ..... < A[N].
- Örneğin, şu şekilde bildirilen ve başlatılan bir dizimiz varsa:
  int A[] = {21, 34, 11, 9, 1, 0, 22};
- O zaman sıralanmış dizi (artan düzende) şu şekilde verilebilir:
  A[] = {0, 1, 9, 11, 21, 22, 34;
- Sıralama algoritması, bir listenin elemanlarını belirli bir sıraya koyan bir algoritma olarak tanımlanır. Bu sıra sayısal sıra, sözlüksel sıra veya kullanıcı tarafından tanımlanan herhangi bir sıra olabilir.
- Sıralama algoritmaları, sıralı listelerin doğru şekilde çalışmasını gerektiren arama ve birleştirme algoritmaları gibi diğer algoritmaların kullanımını optimize etmek için yaygın olarak kullanılır.

# Kabarcık Sıralama

- Kabarcık sıralaması, dizi elemanlarını, en büyük elemanı dizi segmentinin en yüksek indeks pozisyonuna tekrar tekrar taşıyarak sıralayan çok basit bir yöntemdir (elemanları artan düzende düzenleme durumunda).
- Kabarcık sıralama yönteminde, dizideki ardışık bitişik eleman çiftleri birbirleriyle karşılaştırılır.
  - Eğer alt indeksteki eleman üst indeksteki elemandan büyükse, iki eleman yer değiştirilerek büyük olan elemandan önce yerleştirilir.
  - Bu işlem, sıralanmamış elemanların listesi tükenene kadar devam edecektir.
- Bu sıralama prosedürüne kabarcık sıralaması denir çünkü öğeler listenin en üstüne "kabarcık" şeklinde çıkar.
- İlk geçişin sonunda, listedeki en büyük eleman uygun pozisyonuna yerleştirilecektir
- **Not Eğer elemanlar azalan düzende sıralanacaksa ilk geçişte en küçük eleman dizinin en yüksek indeksine taşınır.**

# Kabarcık Sıralama

- ***Teknik***
- Kabarcık sıralamasının temel çalışma metodolojisi aşağıdaki şekilde verilmiştir:
  a. Geçiş 1'de, A[0] ve A[1] karşılaştırılır, ardından A[1], A[2] ile karşılaştırılır, A[2], A[3] ile karşılaştırılır, vb. Son olarak, A[N–2], A[N–1] ile karşılaştırılır. Geçiş 1, n–1 karşılaştırma içerir ve en büyük öğeyi dizinin en yüksek dizinine yerleştirir.
  b. Geçiş 2'de, A[0] ve A[1] karşılaştırılır, sonra A[1], A[2] ile karşılaştırılır, A[2], A[3] ile karşılaştırılır, vb. Son olarak, A[N–3], A[N–2] ile karşılaştırılır. Geçiş 2, n–2 karşılaştırma içerir ve ikinci en büyük öğeyi dizinin ikinci en yüksek dizinine yerleştirir.
  c. Geçiş 3'te, A[0] ve A[1] karşılaştırılır, ardından A[1], A[2] ile karşılaştırılır, A[2], A[3] ile karşılaştırılır, vb. Son olarak, A[N–4], A[N–3] ile karşılaştırılır. Geçiş 3, n–3 karşılaştırma içerir ve üçüncü en büyük öğeyi dizinin üçüncü en yüksek dizinine yerleştirir.
  d. n–1 Geçişinde, A[0] ve A[1], A[0]<A[1] olacak şekilde karşılaştırılır. Bu adımdan sonra, dizinin tüm elemanları artan düzende düzenlenir.

# Kabarcık Sıralama

- **Örnek 14.2 Kabarcık sıralamasını ayrıntılı olarak ele almak için, aşağıdaki öğelere sahip bir A[] dizisini ele alalım:**

```
A[] = {30, 52, 29, 87, 63, 27, 19, 54}

Pass 1:
 (a) Compare 30 and 52. Since 30 < 52, no swapping is done.
 (b) Compare 52 and 29. Since 52 > 29, swapping is done.
     30, 29, 52, 87, 63, 27, 19, 54
 (c) Compare 52 and 87. Since 52 < 87, no swapping is done.
 (d) Compare 87 and 63. Since 87 > 63, swapping is done.
     30, 29, 52, 63, 87, 27, 19, 54
 (e) Compare 87 and 27. Since 87 > 27, swapping is done.
     30, 29, 52, 63, 27, 87, 19, 54
 (f) Compare 87 and 19. Since 87 > 19, swapping is done.
     30, 29, 52, 63, 27, 19, 87, 54
 (g) Compare 87 and 54. Since 87 > 54, swapping is done.
     30, 29, 52, 63, 27, 19, 54, 87
```

- İlk geçişin s … e yerleştirildiğini gözlemleyin. Diğer tüm elemar

```
Pass 2:
 (a) Compare 30 and 29. Since 30 > 29, swapping is done.
     29, 30, 52, 63, 27, 19, 54, 87
 (b) Compare 30 and 52. Since 30 < 52, no swapping is done.
 (c) Compare 52 and 63. Since 52 < 63, no swapping is done.
 (d) Compare 63 and 27. Since 63 > 27, swapping is done.
     29, 30, 52, 27, 63, 19, 54, 87
 (e) Compare 63 and 19. Since 63 > 19, swapping is done.
     29, 30, 52, 27, 19, 63, 54, 87
 (f) Compare 63 and 54. Since 63 > 54, swapping is done.
     29, 30, 52, 27, 19, 54, 63, 87
```

- İkinci geçiş … ksek indeksine yerleştirildiğini gözlemleyi

# Kabarcık Sıralama

```
Pass 3:
 (a) Compare 29 and 30. Since 29 < 30, no swapping is done.
 (b) Compare 30 and 52. Since 30 < 52, no swapping is done.
 (c) Compare 52 and 27. Since 52 > 27, swapping is done.
     29, 30, 27, 52, 19, 54, 63, 87
 (d) Compare 52 and 19. Since 52 > 19, swapping is done.
     29, 30, 27, 19, 52, 54, 63, 87
 (e) Compare 52 and 54. Since 52 < 54, no swapping is done.
```

- Üçüncü ğ …üncü en yüksek indeksine yerleştirildiğini gözlemleyin. Diğer tüm elemanlar hala sıralanmamış durumdadır.

```
Pass 4:
 (a) Compare 29 and 30. Since 29 < 30, no swapping is done.
 (b) Compare 30 and 27. Since 30 > 27, swapping is done.
     29, 27, 30, 19, 52, 54, 63, 87
 (c) Compare 30 and 19. Since 30 > 19, swapping is done.
     29, 27, 19, 30, 52, 54, 63, 87
 (d) Compare 30 and 52. Since 30 < 52, no swapping is done.
```

- Dördünc …in dördüncü en yüksek indeksin… …hala sıralanmamış durumdadır.

```
Pass 5:
 (a) Compare 29 and 27. Since 29 > 27, swapping is done.
     27, 29, 19, 30, 52, 54, 63, 87
 (b) Compare 29 and 19. Since 29 > 19, swapping is done.
     27, 19, 29, 30, 52, 54, 63, 87
 (c) Compare 29 and 30. Since 29 < 30, no swapping is done.
```

- Beşinci ğ …nci en yüksek indeksine yerleştir… …amış durumdadır.

# Kabarcık Sıralama

```
Pass 6:
 (a) Compare 27 and 19. Since 27 > 19, swapping is done.
     19, 27, 29, 30, 52, 54, 63, 87
 (b) Compare 27 and 29. Since 27 < 29, no swapping is done.
```

- Altıncı geçişin sonunda altıncı en büyük elemanın dizinin altıncı en büyük indeksine yerleştirildiğini gözlemleyin. Diğer tüm elemanlar hala sıralanmamış durumdadır.

```
Pass 7:
 (a) Compare 19 and 27. Since 19 < 27, no swapping is done.
```

- Dikkat ederseniz artık tüm liste sıralanmış durumda.

# Kabarcık Sıralama

- Şekil 14.6'da kabarcık sıralaması algoritması gösterilmektedir.
- Bu algoritmada dış döngü toplam geçiş sayısı olan N–1 içindir.
- İç döngü her geçişte yürütülecektir. Ancak, iç döngünün frekansı her geçişte azalacaktır çünkü her geçişten sonra bir eleman doğru pozisyonunda olacaktır.
- Bu nedenle, her geçişte iç döngü N–I kez yürütülecektir; burada N dizideki eleman sayısı ve I geçiş sayısıdır.

```
BUBBLE_SORT(A, N)

Step 1: Repeat Step 2 For 1 = 0 to N-1
Step 2:     Repeat For J = 0 to N - I
Step 3:                 IF A[J] > A[J + 1]
                        SWAP A[J] and A[J+1]
            [END OF INNER LOOP]
        [END OF OUTER LOOP]
Step 4: EXIT
```

**Figure 14.6** Algorithm for bubble sort

# Kabarcık Sıralama

- ***Kabarcık Sıralamanın Karmaşıklığı***
- Herhangi bir sıralama algoritmasının karmaşıklığı karşılaştırma sayısına bağlıdır.
- Kabarcık sıralamasında toplamda N–1 geçiş olduğunu gördük.
  - İlk geçişte, en yüksek elemanı doğru pozisyonuna yerleştirmek için N–1 karşılaştırma yapılır.
  - Daha sonra 2. Geçişte N–2 karşılaştırma yapılır ve ikinci en yüksek eleman onun pozisyonuna yerleştirilir.
  - Bu nedenle, kabarcık sıralamasının karmaşıklığını hesaplamak için toplam karşılaştırma sayısını hesaplamamız gerekir. Bu şu şekilde verilebilir:
    - $f(n) = (n – 1) + (n – 2) + (n – 3) + ..... + 3 + 2 + 1$
    - $f(n) = n (n – 1)/2$
    - $f(n) = n^2/2 + O(n) = O(n^2)$
- Bu nedenle, kabarcık sıralama algoritmasının karmaşıklığı $O(n^2)$'dir. Bu, kabarcık sıralamayı yürütmek için gereken zamanın $n^2$ ile orantılı olduğu anlamına gelir, burada n dizideki toplam eleman sayısıdır.

# Kabarcık Sıralama

**PROGRAMMING EXAMPLE**

**5.** Write a program to enter *n* numbers in an array. Redisplay the array with elements being sorted in ascending order.

```
#include <stdio.h>
#include <conio.h>
int main()
{
        int i, n, temp, j, arr[10];
        clrscr();
        printf("\n Enter the number of elements in the array : ");
        scanf("%d", &n);
        printf("\n Enter the elements: ");
        for(i=0;i<n;i++)
        {
                scanf("%d", &arr [i]);
        }
        for(i=0;i<n;i++)
        {
                for(j=0;j<n-i-1;j++)
                {
                        if(arr[j] > arr[j+1])
                        {
                                temp = arr[j];
                                arr[j] = arr[j+1];
                                arr[j+1] = temp;
                        }
                }
        }
        printf("\n The array sorted in ascending order is :\n");
        for(i=0;i<n;i++)
                printf("%d\t", arr[i]);
        getch();
        return 0;
}
Output
Enter the number of elements in the array : 10
Enter the elements : 8    9    6    7    5    4    2    3    1    10
The array sorted in ascending order is :
1    2    3    4    5    6    7    8    9    10
```

# Ekleme Sıralaması

- Eklemeli sıralama, sıralanmış dizinin (veya listenin) her seferinde bir eleman olarak oluşturulduğu çok basit bir sıralama algoritmasıdır. Hepimiz bu sıralama tekniğine aşinayız çünkü bunu genellikle briç oynarken bir deste kartı sıralamak için kullanırız.
- Eklemeli sıralamanın arkasındaki temel fikir, her öğeyi son listedeki doğru yerine yerleştirmesidir.
- Bellek tasarrufu sağlamak için, ekleme sıralaması algoritmasının çoğu uygulaması, geçerli veri öğesini zaten sıralanmış değerlerin ötesine taşıyarak ve doğru yerine gelene kadar onu önceki değerle tekrar tekrar değiştirerek çalışır.
- Eklemeli sıralama, hızlı sıralama, yığın sıralaması ve birleştirme sıralaması gibi diğer daha gelişmiş algoritmalarla karşılaştırıldığında daha az verimlidir.

# Ekleme Sıralaması

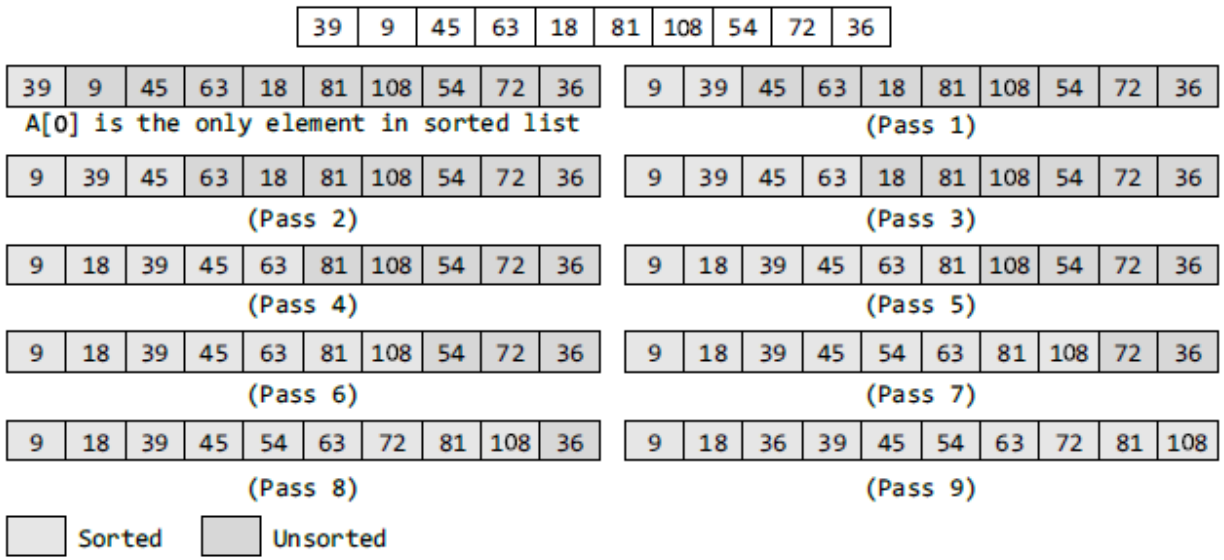
- ***Teknik***
- Eklemeli sıralama şu şekilde çalışır:
  - Sıralanacak değerler dizisi iki kümeye ayrılır. Biri sıralanmış değerleri depolar ve diğeri sıralanmamış değerleri içerir.
  - Sıralama algoritması, sıralanmamış kümede elemanlar kalana kadar devam edecektir.
  - Dizide n eleman olduğunu varsayalım. Başlangıçta, indeksi 0 olan eleman (LB = 0 varsayılarak) sıralanmış kümededir. Elemanların geri kalanı sıralanmamış kümededir.
  - Sıralanmamış bölümün ilk elemanının dizi indeksi 1'dir (LB = 0 ise).
  - Algoritmanın her yinelemesinde, sıralanmamış kümedeki ilk eleman alınır ve sıralanmış kümedeki doğru konuma yerleştirilir.

# Ekleme Sıralaması

- Başlangıçta, A[0] sıralanmış kümedeki tek elemandır. 1. Geçişte, A[1] A[0]'dan önce veya sonra yerleştirilecektir, böylece A dizisi sıralanmış olacaktır. 2. Geçişte, A[2] A[0]'dan önce, A[0] ile A[1] arasına veya A[1]'den sonra yerleştirilecektir. 3. Geçişte, A[3] doğru yerine yerleştirilecektir. N–1. Geçişte, A[N–1] dizinin sıralı kalmasını sağlamak için doğru yerine yerleştirilecektir.

**Example 14.3** Consider an array of integers given below. We will sort the values in the array using insertion sort.

***Solution***



# Ekleme Sıralaması

- Sıralı bir liste olan A[0], A[1], ..., A[K-1]'e bir A[K] elemanı eklemek için, A[K]'yi önce A[K–1] ile, sonra A[K–2], A[K–3] ile karşılaştırmamız gerekir, ta ki A[J] <= A[K] olacak şekilde bir A[J] elemanıyla karşılaşana kadar. A[K]'yi doğru pozisyonuna eklemek için, A[K–1], A[K–2], ..., A[J] elemanlarını bir pozisyon taşımamız ve sonra A[K]'yi $(J+1)^{th}$ konumuna eklememiz gerekir.
- Ekleme sıralaması algoritması Şekil 14.7'de verilmiştir.
- Algoritmada Adım 1, dizideki her bir eleman için tekrarlanacak bir for döngüsü çalıştırır.

- Adım 2'de K'ıncı elemanın değerini TEMP'e kaydediyoruz.
- 3. Adımda dizideki $J^{th}$ indeksini ayarlıyoruz.
- 4. Adımda, sıralanmamış listeden gelen yeni elemanın sıralanmış elemanlar listesinde saklanması için alan yaratacak bir for döngüsü yürütülür.
- Son olarak Adım 5'te eleman $(J+1)^{th}$ konumuna kaydedilir.

```
INSERTION-SORT (ARR, N)

Step 1: Repeat Steps 2 to 5 for K = 1 to N - 1
Step 2:      SET TEMP = ARR[K]
Step 3:      SET J = K - 1
Step 4:      Repeat while TEMP <= ARR[J]
                   SET ARR[J + 1] = ARR[J]
                   SET J = J - 1
             [END OF INNER LOOP]
Step 5:      SET ARR[J + 1] = TEMP
         [END OF LOOP]
Step 6: EXIT
```

**Figure 14.7** Algorithm for insertion sort

# Ekleme Sıralaması

- ***Ekleme Sıralamasının Karmaşıklığı***
- Eklemeli sıralama için en iyi durum, dizinin zaten sıralanmış olmasıdır.
  - Bu durumda, algoritmanın çalışma süresi doğrusal bir çalışma süresine sahiptir (yani, O(n)). Bunun nedeni, her yinelemede, sıralanmamış kümedeki ilk öğenin yalnızca dizinin sıralanmış kümesindeki son öğeyle karşılaştırılmasıdır.
- Benzer şekilde, ekleme sıralama algoritmasının en kötü durumu, dizinin ters sırada sıralanması durumunda ortaya çıkar.
  - En kötü durumda, sıralanmamış kümenin ilk elemanı sıralanmış kümedeki hemen hemen her elemanla karşılaştırılmalıdır. Ayrıca, iç döngünün her yinelemesi, bir sonraki elemanı eklemeden önce dizinin sıralanmış kümesinin elemanlarını kaydırmak zorunda kalacaktır. Bu nedenle, en kötü durumda, ekleme sıralamasının ikinci dereceden bir çalışma süresi vardır (yani, $O(n^2)$).
- Ortalama durumda bile, ekleme sıralama algoritması en azından (K–1)/2 karşılaştırma yapmak zorunda kalacaktır. Bu nedenle, ortalama durum da ikinci dereceden bir çalışma süresine sahiptir.

# Ekleme Sıralaması

**PROGRAMMING EXAMPLE**

6. Write a program to sort an array using insertion sort algorithm.

```
#include <stdio.h>
#include <conio.h>
#define size 5
void insertion_sort(int arr[], int n);
void main()
{
        int arr[size], i, n;
        printf("\n Enter the number of elements in the array: ");
        scanf("%d", &n);
        printf("\n Enter the elements of the array: ");
        for(i=0;i<n;i++)
        {
                scanf("%d", &arr[i]);
        }
        insertion_sort(arr, n);
        printf("\n The sorted array is:  \n");
        for(i=0;i<n;i++)
        printf(" %d\t", arr[i]);
        getch();
}
void insertion_sort(int arr[], int n)
{
        int i, j, temp;
        for(i=1;i<n;i++)
        {
                temp = arr[i];
                j = i-1;
                while((temp < arr[j]) && (j>=0))
                {
                        arr[j+1] = arr[j];
                        j--;
                }
                arr[j+1] = temp;
        }
}
```

**Output**

```
Enter the number of elements in the array : 5
Enter the elements of the array : 500 1 50 23 76
The sorted array is :
1    23    20    76    500    6    7    8    9    10
```

# Seçim Sıralaması

- Seçimli sıralama, $O(n^2)$'lik bir karesel çalışma zamanı karmaşıklığına sahip olan bir sıralama algoritmasıdır, bu nedenle büyük listelerde kullanılması verimsizdir.
- Seçmeli sıralama algoritması, eklemeli sıralama algoritmasından daha kötü performans gösterse de basitliğiyle dikkat çekmekte ve bazı durumlarda daha karmaşık algoritmalara göre performans avantajlarına sahiptir.
- Seçimli sıralama genellikle çok büyük nesneler (kayıtlar) ve küçük anahtarlar içeren dosyaları sıralamak için kullanılır.
- ***Teknik***
- N elemanlı bir ARR dizisini ele alalım.
- Seçimli sıralama şu şekilde çalışır:
- Önce dizideki en küçük değeri bulup ilk pozisyona yerleştirin. Sonra, dizideki ikinci en küçük değeri bulup ikinci pozisyona yerleştirin. Tüm dizi sıralanana kadar bu prosedürü tekrarlayın. Bu nedenle,
  - Geçiş 1'de, dizideki en küçük değerin POS konumunu bulun ve ardından ARR[POS] ve ARR[0]'ı değiştirin. Böylece, ARR[0] sıralanır.
  - Geçiş 2'de, N–1 elemanlı alt dizideki en küçük değerin POS konumunu bulun. ARR[POS]'u ARR[1] ile değiştirin. Şimdi, ARR[0] ve ARR[1] sıralanmıştır.
  - N–1 Geçişinde, ARR[N–2] ve ARR[N–1] öğelerinin daha küçük olanının POS konumunu bulun. ARR[0], ARR[1], ..., ARR[N–1] sıralanacak şekilde ARR[POS] ve ARR[N–2] öğelerini değiştirin.

# Seçim Sıralaması

**Example 14.4** Sort the array given below using selection sort.

| 39 | 9 | 81 | 45 | 90 | 27 | 72 | 18 |
|---|---|---|---|---|---|---|---|

| PASS | POS | ARR[0] | ARR[1] | ARR[2] | ARR[3] | ARR[4] | ARR[5] | ARR[6] | ARR[7] |
|---|---|---|---|---|---|---|---|---|---|
| 1 | 1 | 9 | 39 | 81 | 45 | 90 | 27 | 72 | 18 |
| 2 | 7 | 9 | 18 | 81 | 45 | 90 | 27 | 72 | 39 |
| 3 | 5 | 9 | 18 | 27 | 45 | 90 | 81 | 72 | 39 |
| 4 | 7 | 9 | 18 | 27 | 39 | 90 | 81 | 72 | 45 |
| 5 | 7 | 9 | 18 | 27 | 39 | 45 | 81 | 72 | 90 |
| 6 | 6 | 9 | 18 | 27 | 39 | 45 | 72 | 81 | 90 |
| 7 | 6 | 9 | 18 | 27 | 39 | 45 | 72 | 81 | 90 |

# Seçim Sıralaması

- Seçimli sıralama algoritması Şekil 14.8'de gösterilmiştir.
- Algoritmada, $K^{th}$ geçişi sırasında ARR[K], ARR[K+1], ..., ARR[N]'den en küçük elemanların POS konumunu bulmamız gerekiyor.
- En küçük öğeyi bulmak için, ARR[K] ile ARR[N] arasında değişen alt dizideki en küçük değeri tutan SMALL değişkenini kullanırız.
- Daha sonra ARR[K]'yi ARR[POS] ile değiştirin.
- Bu işlem dizideki tüm elemanlar sıralanana kadar tekrarlanır.

```
SMALLEST (ARR, K, N, POS)

Step 1: [INITIALIZE] SET SMALL = ARR[K]
Step 2: [INITIALIZE] SET POS = K
Step 3: Repeat for J = K+1 to N-1
            IF SMALL > ARR[J]
                  SET SMALL = ARR[J]
                  SET POS = J
            [END OF IF]
        [END OF LOOP]
Step 4: RETURN POS
```

```
SELECTION SORT(ARR, N)

Step 1: Repeat Steps 2 and 3 for K = 1
        to N-1
Step 2:     CALL SMALLEST(ARR, K, N, POS)
Step 3:     SWAP A[K] with ARR[POS]
      [END OF LOOP]
Step 4: EXIT
```

**Figure 14.8** Algorithm for selection sort

# Seçim Sıralaması

- ***Seçim Sıralamasının Karmaşıklığı***
- Seçimli sıralama, dizideki elemanların orijinal sırasından bağımsız bir sıralama algoritmasıdır.
- Geçiş 1'de, en küçük değere sahip öğeyi seçmek tüm n öğenin taranmasını gerektirir; bu nedenle, ilk geçişte n–1 karşılaştırma gerekir. Daha sonra, en küçük değer ilk pozisyondaki öğeyle değiştirilir.
- 2. Geçişte, ikinci en küçük değeri seçmek için kalan n – 1 elemanın taranması ve benzeri işlemler gerekir.
- Oyleyse,
  - (n – 1) + (n – 2) + … + 2 + 1
  - = n(n – 1) / 2 = $O(n^2)$ karşılaştırmaları

# Seçim Sıralaması

**PROGRAMMING EXAMPLE**

7. Write a program to sort an array using selection sort algorithm.

```
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
int smallest(int arr[], int k, int n);
void selection_sort(int arr[], int n);
void main(int argc, char *argv[]) {
        int arr[10], i, n;
        printf("\n Enter the number of elements in the array: ");
        scanf("%d", &n);
        printf("\n Enter the elements of the array: ");
        for(i=0;i<n;i++)
        {
                scanf("%d", &arr[i]);
        }
        selection_sort(arr, n);
        printf("\n The sorted array is: \n");
        for(i=0;i<n;i++)
        printf(" %d\t", arr[i]);
}
int smallest(int arr[], int k, int n)
{
        int pos = k, small=arr[k], i;
        for(i=k+1;i<n;i++)
        {
                if(arr[i]< small)
                {
                        small = arr[i];
                        pos = i;
                }
        }
        return pos;
}
void selection_sort(int arr[],int n)
{
        int k, pos, temp;
        for(k=0;k<n;k++)
        {
                pos = smallest(arr, k, n);
                temp = arr[k];

                arr[k] = arr[pos];
                arr[pos] = temp;
        }
}
```

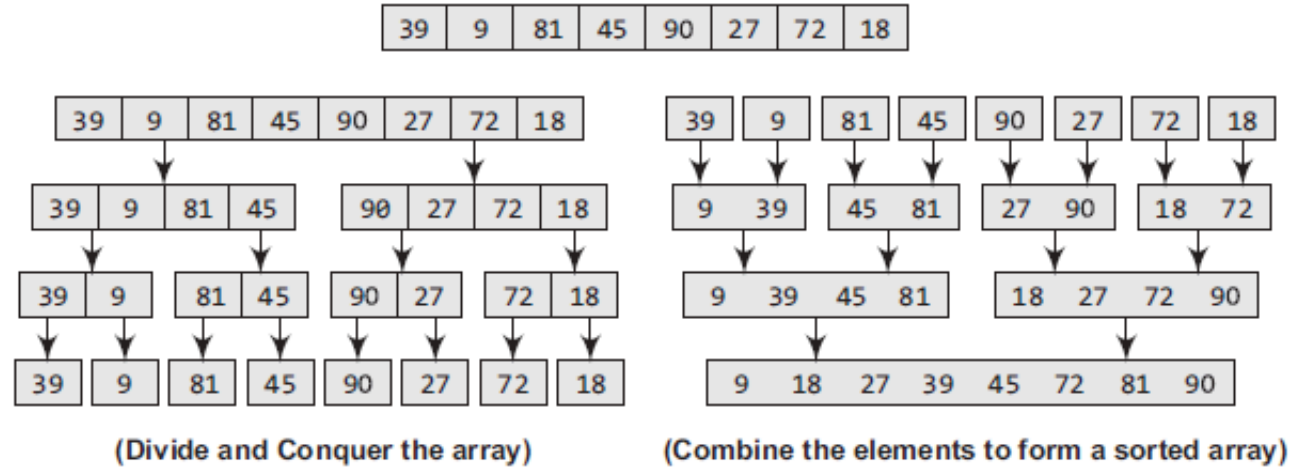
# Birleştirme Sıralaması

- Birleştirme sıralaması, böl, yönet ve birleştir algoritmik paradigmasını kullanan bir sıralama algoritmasıdır.
- ***Bölme, n elemanlı diziyi n/2 elemanlı iki alt diziye ayırmak anlamına gelir. Eğer A sıfır veya bir eleman içeren bir diziyse, o zaman zaten sıralanmıştır. Ancak, dizide daha fazla eleman varsa, A'yı her biri A'nın elemanlarının yaklaşık yarısını içeren A1 ve A2 olmak üzere iki alt diziye bölün.***
- ***Conquer, iki alt diziyi birleştirme sıralaması kullanarak yinelemeli olarak sıralamak anlamına gelir.***
- ***Birleştirme, n/2 boyutundaki iki sıralı alt diziyi birleştirerek n elemanlı sıralı diziyi üretmek anlamına gelir.***

# Birleştirme Sıralaması

**Example 14.5** Sort the array given below using merge sort.

***Solution***




(Divide and Conquer the array) (Combine the elements to form a sorted array)

# Birleştirme Sıralaması

- Birleştirme sıralama algoritması (Şekil 14.9), alt dizileri birleştirerek sıralanmış bir dizi oluşturan bir birleştirme işlevini kullanır.
- Birleştirme sıralama algoritması listeyi yinelemeli olarak daha küçük listeler halinde bölerken, birleştirme algoritması listeyi ele geçirerek tek tek listelerdeki elemanları sıralar.
- Son olarak daha küçük listeler birleştirilerek tek bir liste oluşturulur.

```
MERGE_SORT(ARR, BEG, END)
Step 1: IF BEG < END
            SET MID = (BEG + END)/2
            CALL MERGE_SORT (ARR, BEG, MID)
            CALL MERGE_SORT (ARR, MID + 1, END)
            MERGE (ARR, BEG, MID, END)
        [END OF IF]
Step 2: END
```

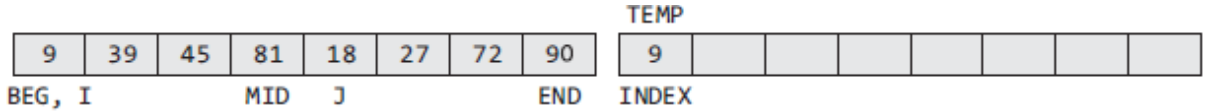
**Figure 14.9** Algorithm for merge sort

# Birleştirme Sıralaması

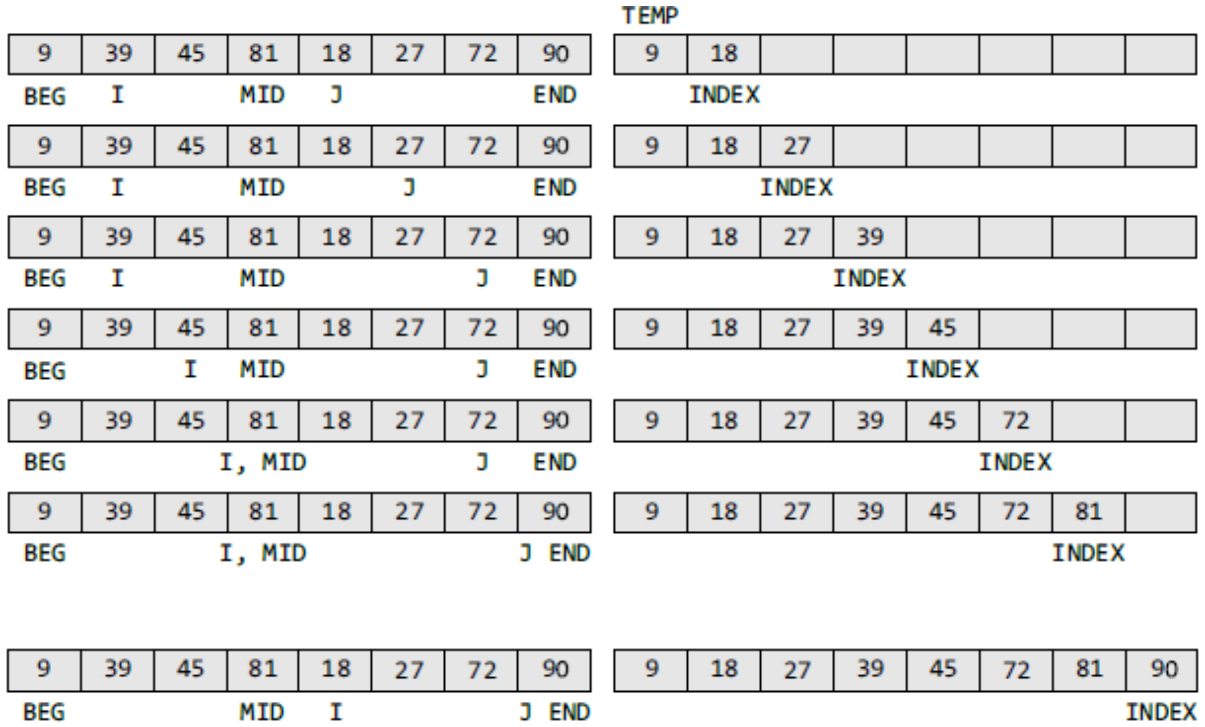
```
MERGE (ARR, BEG, MID, END)

Step 1: [INITIALIZE] SET I = BEG, J = MID + 1, INDEX = 0
Step 2: Repeat while (I <= MID) AND (J<=END)
            IF ARR[I] < ARR[J]
                  SET TEMP[INDEX] = ARR[I]
                  SET I = I + 1
            ELSE
                  SET TEMP[INDEX] = ARR[J]
                  SET J = J + 1
            [END OF IF]
            SET INDEX = INDEX + 1
      [END OF LOOP]
Step 3: [Copy the remaining elements of right sub-array, if any]
            IF I > MID
               Repeat while J <= END
                  SET TEMP[INDEX] = ARR[J]
                  SET INDEX = INDEX + 1, SET J = J + 1
               [END OF LOOP]
      [Copy the remaining elements of left sub-array, if any]
            ELSE
               Repeat while I <= MID
                  SET TEMP[INDEX] = ARR[I]
                  SET INDEX = INDEX + 1, SET I = I + 1
               [END OF LOOP]
            [END OF IF]
Step 4: [Copy the contents of TEMP back to ARR] SET K=0
Step 5: Repeat while K < INDEX
               SET ARR[K] = TEMP[K]
               SET K = K + 1
      [END OF LOOP]
Step 6: END
```

# Birleştirme Sıralaması

- Birleştirme algoritmasını anlamak için, iki listeyi nasıl birleştirip tek bir liste oluşturduğumuzu gösteren aşağıdaki şekli inceleyin.
- Anlaşılmasının kolay olması için her biri dört elemandan oluşan iki alt liste aldık.
- Aynı kavram, iki eleman içeren dört alt listeyi veya her biri bir eleman içeren sekiz alt listeyi birleştirmek için de kullanılabilir.

- ARR[I] ve 

elirtilen yere yerleştirilir ve ardından I veya J değeri artırılır.




- I, MID'der .

# Birleştirme Sıralaması

- ***Birleştirme Sıralamasının Karmaşıklığı***
- Birleştirme sıralamasının ortalama durumda ve en kötü durumda çalışma süresi O(n log n) olarak verilebilir. Birleştirme sıralamasının optimal bir zaman karmaşıklığı olmasına rağmen, geçici dizi TEMP için O(n) ek bir alana ihtiyaç duyar.

# Birleştirme Sıralaması

**PROGRAMMING EXAMPLE**

**8.** Write a program to implement merge sort.

```
#include <stdio.h>
#include <conio.h>
#define size 100

void merge(int a[], int, int, int);
void merge_sort(int a[],int, int);
void main()
{
        int arr[size], i, n;
        printf("\n Enter the number of elements in the array : ");
        scanf("%d", &n);
        printf("\n Enter the elements of the array: ");
        for(i=0;i<n;i++)
        {
                scanf("%d", &arr[i]);
        }
        merge_sort(arr, 0, n-1);
        printf("\n The sorted array is: \n");
        for(i=0;i<n;i++)
        printf(" %d\t", arr[i]);
        getch();
}
void merge(int arr[], int beg, int mid, int end)
{
        int i=beg, j=mid+1, index=beg, temp[size], k;
        while((i<=mid) && (j<=end))
        {
                if(arr[i] < arr[j])
                {
                        temp[index] = arr[i];
                        i++;
                }
                else
                {
                        temp[index] = arr[j];
                        j++;
                }
                index++;
        }
        if(i>mid)
        {
                while(j<=end)
                {
                        temp[index] = arr[j];
                        j++;
                        index++;
                }
        }
        else
        {
                while(i<=mid)
                {
                        temp[index] = arr[i];
                        i++;
                        index++;
                }
        }
        for(k=beg;k<index;k++)
        arr[k] = temp[k];
}
void merge_sort(int arr[], int beg, int end)
{
        int mid;
        if(beg<end)
        {
                mid = (beg+end)/2;
                merge_sort(arr, beg, mid);
                merge_sort(arr, mid+1, end);
                merge(arr, beg, mid, end);
        }
}
```

# Hızlı Sıralama

- Birleştirme sıralamasında olduğu gibi, yinelemenin temel durumu, dizinin sıfır veya bir elemanı olduğu durumdur çünkü bu durumda dizi zaten sıralanmıştır.
- Her yinelemeden sonra bir eleman (pivot) her zaman son pozisyonundadır.
- Dolayısıyla her yinelemede dizide sıralanacak bir eleman daha az olur.
- Dolayısıyla asıl görev, diziyi iki eşit parçaya bölecek pivot elemanını bulmaktır.

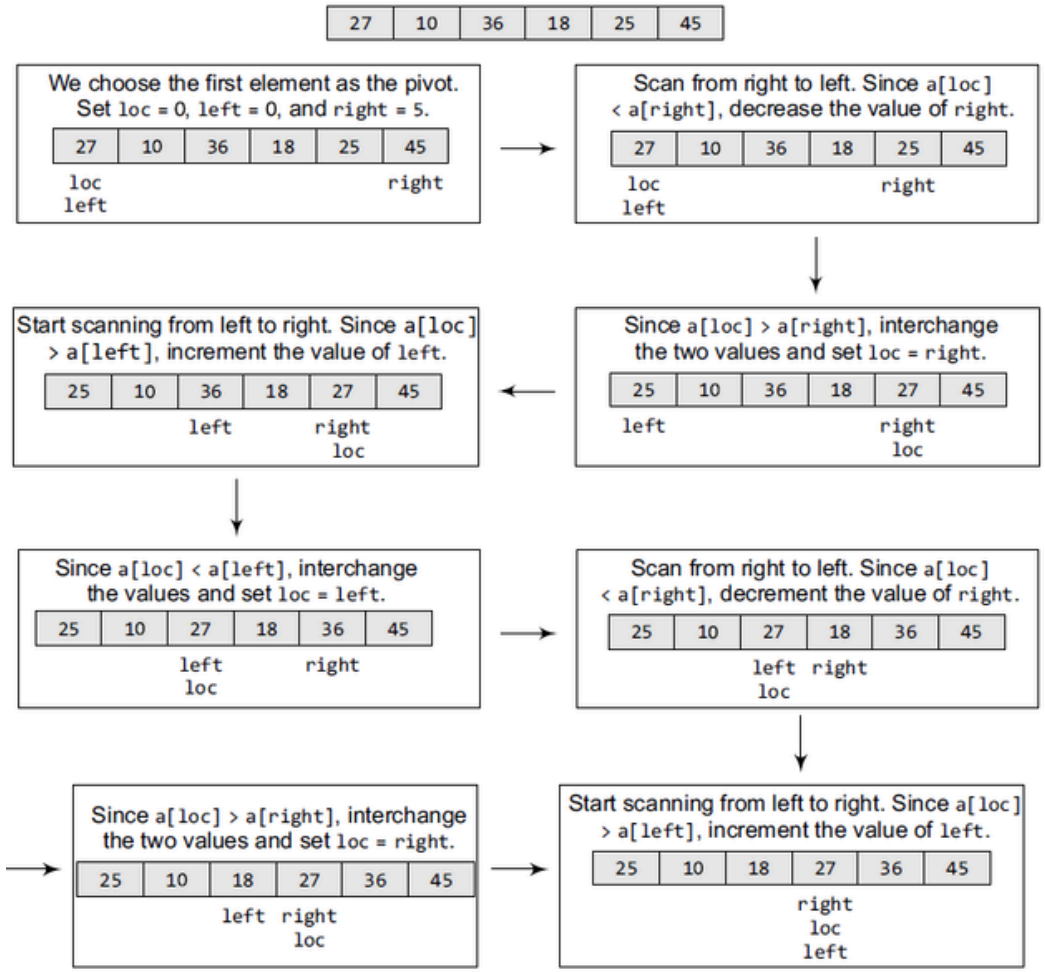
# Hızlı Sıralama

- ***Teknik***
- Hızlı sıralama şu şekilde çalışır:

1. Dizideki ilk öğenin dizinini loc ve left değişkenlerine ayarlayın. Ayrıca, dizinin son öğesinin dizinini right değişkenine ayarlayın. Yani, loc = 0, left = 0 ve right = n–1 (burada n, dizideki öğe sayısındadır)
2. right tarafından işaret edilen öğeden başlayın ve diziyi sağdan sola tarayın, yol üzerindeki her öğeyi loc değişkeni tarafından işaret edilen öğeyle karşılaştırın. Yani, a[loc] a[right]'dan küçük olmalıdır.
    a. Eğer durum buysa, o zaman sağ loc'a eşit olana kadar karşılaştırmaya devam edin. Sağ = loc olduğunda, pivotun doğru pozisyonuna yerleştirildiği anlamına gelir.
    b. Ancak, herhangi bir noktada a[loc] > a[right] varsa, o zaman iki değeri birbiriyle değiştirin ve Adım 3'e geçin.
    c. loc = sağa ayarla
3. left ile işaret edilen öğeden başlayın ve diziyi soldan sağa tarayın, yol üzerindeki her öğeyi loc ile işaret edilen öğeyle karşılaştırın. Yani, a[loc] a[left]'den büyük olmalıdır.
    a. Eğer durum buysa, o zaman sol loc'a eşit olana kadar karşılaştırmaya devam edin. Sol = loc olduğunda, pivotun doğru pozisyonuna yerleştirildiği anlamına gelir.
    b. Ancak, herhangi bir noktada a[loc] < a[left] değerine sahipsek, iki değeri birbiriyle değiştirin ve 2. Adıma geçin.
    c. Loc = left olarak ayarla.

# Hızlı Sıralama

**Example 14.6** Sort the elements given in the following array using quick sort algorithm




- Şimdi left = loc, yani prosedür pivot elemanı (dizinin ilk elemanı, yani 27) doğru pozisyonuna yerleştirildiğinde sonlanır. 27'den küçük tüm elemanlar ondan önce, 27'den büyük olanlar ondan sonra yerleştirilir. 25, 10, 18'i içeren sol alt dizi ve 36 ve 45'i içeren sağ alt dizi aynı şekilde sıralanır.

# Hızlı Sıralama

- Hızlı sıralama algoritması (Şekil 14.10), diziyi iki alt diziye bölmek için Partition fonksiyonunu kullanır.

```
PARTITION (ARR, BEG, END, LOC)

Step 1: [INITIALIZE] SET LEFT = BEG, RIGHT = END, LOC = BEG, FLAG = 0
Step 2: Repeat Steps 3 to 6 while FLAG = 0
Step 3: Repeat while ARR[LOC] <= ARR[RIGHT] AND LOC != RIGHT
              SET RIGHT = RIGHT - 1
        [END OF LOOP]
Step 4: IF LOC = RIGHT
              SET FLAG = 1
        ELSE IF ARR[LOC] > ARR[RIGHT]
              SWAP ARR[LOC] with  ARR[RIGHT]
              SET LOC = RIGHT
        [END OF IF]
Step 5: IF FLAG = 0
              Repeat while ARR[LOC] >= ARR[LEFT] AND LOC != LEFT
              SET LEFT = LEFT + 1
              [END OF LOOP]
Step 6:       IF LOC = LEFT
                      SET FLAG = 1
              ELSE IF ARR[LOC] < ARR[LEFT]
                      SWAP ARR[LOC] with  ARR[LEFT]
                      SET LOC = LEFT
              [END OF IF]
        [END OF IF]
Step 7: [END OF LOOP]
Step 8: END

QUICK_SORT (ARR, BEG, END)

Step 1: IF (BEG < END)
            CALL PARTITION (ARR, BEG, END, LOC)
            CALL QUICKSORT(ARR, BEG, LOC - 1)
            CALL QUICKSORT(ARR, LOC + 1, END)
        [END OF IF]
Step 2: END
```

**Figure 14.10** Algorithm for quick sort

# Hızlı Sıralama

- ***Hızlı Sıralamanın Karmaşıklığı***
- Ortalama durumda, hızlı sıralama işleminin çalışma süresi O(nlog n) olarak verilebilir.
- Dizinin bölümlenmesi, dizi elemanları üzerinde bir kez döngü oluşturarak O(n) zaman kullanır.
- En iyi durumda, diziyi her böldüğümüzde, listeyi neredeyse eşit iki parçaya böleriz. Yani, yinelemeli çağrı, boyutu yarı olan alt diziyi işler. En fazla, boyutu 1 olan bir alt diziye ulaşmadan önce yalnızca log n iç içe çağrı yapılabilir. Bu, çağrı ağacının derinliğinin O(log n) olduğu anlamına gelir. Ve her seviyede yalnızca O(n) olabileceğinden, sonuç süresi O(n log n) süre olarak verilir.
- Hızlı sıralama işleminin pratikte verimliliği pivot olarak seçilen elemana bağlıdır.
- En kötü durum verimliliği $O(n^2)$ olarak verilir. En kötü durum, dizi zaten sıralanmış olduğunda (artan veya azalan düzende) ve en soldaki eleman pivot olarak seçildiğinde meydana gelir.
- Ancak, birçok uygulama pivot elemanını rastgele seçer. Hızlı sıralama algoritmasının rastgele versiyonu her zaman O(n log n) algoritmik karmaşıklığına sahiptir.

# Hızlı Sıralama

**PROGRAMMING EXAMPLE**

**9.** Write a program to implement quick sort algorithm.

```
#include <stdio.h>
#include <conio.h>
#define size 100
int partition(int a[], int beg, int end);
void quick_sort(int a[], int beg, int end);
void main()
{
        int arr[size], i, n;
        printf("\n Enter the number of elements in the array: ");
        scanf("%d", &n);
        printf("\n Enter the elements of the array: ");
        for(i=0;i<n;i++)
        {
                scanf("%d", &arr[i]);
        }
        quick_sort(arr, 0, n-1);
        printf("\n The sorted array is: \n");
        for(i=0;i<n;i++)
        printf(" %d\t", arr[i]);
        getch();
}
int partition(int a[], int beg, int end)
{
        int left, right, temp, loc, flag;
        loc = left = beg;
        right = end;
        flag = 0;
        while(flag != 1)
        {
                while((a[loc] <= a[right]) && (loc!=right))
                right--;
```

# Hızlı Sıralama

```
                if(loc==right)
                flag =1;
                else if(a[loc]>a[right])
                {
                        temp = a[loc];
                        a[loc] = a[right];
                        a[right] = temp;
                        loc = right;
                }
                if(flag!=1)
                {
                        while((a[loc] >= a[left]) && (loc!=left))
                        left++;
                        if(loc==left)
                        flag =1;
                        else if(a[loc] <a[left])
                        {
                                temp = a[loc];
                                a[loc] = a[left];
                                a[left] = temp;
                                loc = left;
                        }
                }
        }
        return loc;
}
void quick_sort(int a[], int beg, int end)
{
        int loc;
        if(beg<end)
        {
                loc = partition(a, beg, end);
                quick_sort(a, beg, loc-1);
                quick_sort(a, loc+1, end);
        }
}
```

# Radix'i sırala

- Radix sort, tam sayılar için doğrusal bir sıralama algoritmasıdır ve isimleri alfabetik sıraya göre sıralama kavramını kullanır.
  - Sıralanmış isimlerden oluşan bir listemiz olduğunda, taban 26'dır (veya 26 kova) çünkü İngilizce alfabesinde 26 harf vardır.
  - Kelimelerin önce ismin ilk harfine göre sıralandığına dikkat edin.
  - İkinci geçişte, isimler ikinci harfe göre gruplandırılır. İkinci geçişten sonra, isimler ilk iki harfe göre sıralanır. Bu işlem, n'inci geçişe kadar devam eder, burada n, en fazla harfe sahip ismin uzunluğudur.
- Tam sayılarda radix sort kullanıldığında, sıralama sayıdaki her bir basamakta yapılır. Sıralama prosedürü en önemsizden en önemli basamağa doğru sıralanarak ilerler. Sayıları sıralarken, her biri bir basamak (0, 1, 2, ..., 9) için on kovamız olur ve geçiş sayısı, en fazla basamağa sahip sayının uzunluğuna bağlı olacaktır.

# Radix'i sırala

```
Algorithm for RadixSort (ARR, N)

Step 1: Find the largest number in ARR as LARGE
Step 2: [INITIALIZE] SET NOP = Number of digits in LARGE
Step 3: SET PASS = 0
Step 4: Repeat Step 5 while PASS <= NOP-1
Step 5:            SET I = 0 and INITIALIZE buckets
Step 6:            Repeat Steps 7 to 9 while I<N-1
Step 7:                  SET DIGIT  = digit at PASSth place in A[I]
Step 8:                  Add A[I] to the bucket numbered DIGIT
Step 9:                  INCEREMENT bucket count for bucket numbered DIGIT
                   [END OF LOOP]
Step 10:           Collect the numbers in the bucket
        [END OF LOOP]
Step 11: END
```

**Figure 14.11** Algorithm for radix sort

# Radix'i sırala

**Example 14.7** Sort the numbers given below using radix sort.

345, 654, 924, 123, 567, 472, 555, 808, 911

In the first pass, the numbers are sorted according to the digit at ones place. The buckets are pictured upside down as shown below.

| Number | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| 345 | | | | | | 345 | | | | |
| 654 | | | | | 654 | | | | | |
| 924 | | | | | 924 | | | | | |
| 123 | | | | 123 | | | | | | |
| 567 | | | | | | | | 567 | | |
| 472 | | | 472 | | | | | | | |
| 555 | | | | | | 555 | | | | |
| 808 | | | | | | | | | 808 | |
| 911 | | 911 | | | | | | | | |

After this pass, the numbers are collected bucket by bucket. The new list thus formed is used as an input for the next pass. In the second pass, the numbers are sorted according to the digit at the tens place. The buckets are pictured upside down.

| Number | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| 911 | | 911 | | | | | | | | |
| 472 | | | | | | | | 472 | | |
| 123 | | | 123 | | | | | | | |
| 654 | | | | | | 654 | | | | |
| 924 | | | 924 | | | | | | | |
| 345 | | | | | 345 | | | | | |
| 555 | | | | | | 555 | | | | |
| 567 | | | | | | | 567 | | | |
| 808 | 808 | | | | | | | | | |

# Radix'i sırala

In the third pass, the numbers are sorted according to the digit at the hundreds place. The buckets are pictured upside down.

| Number | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| 808 | | | | | | | | | 808 | |
| 911 | | | | | | | | | | 911 |
| 123 | | 123 | | | | | | | | |
| 924 | | | | | | | | | | 924 |
| 345 | | | | 345 | | | | | | |
| 654 | | | | | | | 654 | | | |
| 555 | | | | | | 555 | | | | |
| 567 | | | | | | 567 | | | | |
| 472 | | | | | 472 | | | | | |

The numbers are collected bucket by bucket. The new list thus formed is the final sorted result. After the third pass, the list can be given as

123, 345, 472, 555, 567, 654, 808, 911, 924.

- ***Radix Sort'un Karmaşıklığı***
- Taban sıralaması algoritmasının karmaşıklığını hesaplamak için, sıralanması gereken n sayı olduğunu ve k'nin en büyük sayıdaki basamak sayısı olduğunu varsayalım.
- Bu durumda k kere toplamda radix sort algoritması çağrılır.
- İç döngü n kez yürütülür.
- Bu nedenle, tüm taban sıralama algoritmasının yürütülmesi O(kn) zaman alır.
- Sonlu büyüklükteki bir veri kümesine (çok küçük sayı kümesi) taban sıralaması uygulandığında, algoritma O(n) asimptotik sürede çalışır.

# Radix'i sırala

**PROGRAMMING EXAMPLE**

**10.** Write a program to implement radix sort algorithm.

```
##include <stdio.h>
#include <conio.h>
#define size 10
int largest(int arr[], int n);
void radix_sort(int arr[], int n);
void main()
{
        int arr[size], i, n;
        printf("\n Enter the number of elements in the array: ");
        scanf("%d", &n);
        printf("\n Enter the elements of the array: ");
        for(i=0;i<n;i++)
        {
                scanf("%d", &arr[i]);
        }
        radix_sort(arr, n);
        printf("\n The sorted array is: \n");
        for(i=0;i<n;i++)
        printf(" %d\t", arr[i]);
        getch();
}
int largest(int arr[], int n)
{
        int large=arr[0], i;
        for(i=1;i<n;i++)
        {
                if(arr[i]>large)
                large = arr[i];
        }
        return large;
}
```

# Radix'i sırala

```
void radix_sort(int arr[], int n)
{
        int bucket[size][size], bucket_count[size];
        int i, j, k, remainder, NOP=0, divisor=1, large, pass;
        large = largest(arr, n);
        while(large>0)
        {
                NOP++;
                large/=size;
        }
        for(pass=0;pass<NOP;pass++) // Initialize the buckets
        {
                for(i=0;i<size;i++)
                bucket_count[i]=0;
                for(i=0;i<n;i++)
                {
                        // sort the numbers according to the digit at passth place
                        remainder = (arr[i]/divisor)%size;
                        bucket[remainder][bucket_count[remainder]] = arr[i];
                        bucket_count[remainder] += 1;
                }
                // collect the numbers after PASS pass
                i=0;
                for(k=0;k<size;k++)
                {
                        for(j=0;j<bucket_count[k];j++)
                        {
                                arr[i] = bucket[k][j];
                                i++;
                        }
                }
                divisor *= size;
        }
}

}
```

# Kabuk Sıralaması

- Donald Shell tarafından 1959'da icat edilen Shell sort, ekleme sıralamasının genelleştirilmiş hali olan bir sıralama algoritmasıdır. Ekleme sıralamasını tartışırken iki şey gözlemledik:
  - Öncelikle, eklemeli sıralama, giriş verileri 'neredeyse sıralı' olduğunda iyi çalışır.
  - İkincisi, eklemeli sıralama kullanımı oldukça verimsizdir çünkü değerleri her seferinde yalnızca bir konum hareket ettirir.
- Kabuk sıralaması, birkaç pozisyonluk bir boşlukla ayrılmış öğeleri karşılaştırdığı için ekleme sıralamasına göre bir iyileştirme olarak kabul edilir. Bu, öğenin beklenen konumuna doğru daha büyük adımlar atmasını sağlar. Kabuk sıralamasında, öğeler birden fazla geçişte sıralanır ve her geçişte, veriler daha küçük ve daha küçük boşluk boyutlarıyla alınır. Ancak, kabuk sıralamasının son adımı düz bir ekleme sıralamasıdır. Ancak son adıma ulaştığımızda, öğeler zaten 'neredeyse sıralanmış' olur ve bu nedenle iyi bir performans sağlar.

# Kabuk Sıralaması

**Example 14.8** Sort the elements given below using shell sort.

63, 19, 7, 90, 81, 36, 54, 45, 72, 27, 22, 9, 41, 59, 33

***Solution***

Arrange the elements of the array in the form of a table and sort the columns.

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 63 | 19 | 7 | 90 | 81 | 36 | 54 | 45 |
| 72 | 27 | 22 | 9 | 41 | 59 | 33 | |

*Result:*

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 63 | 19 | 7 | 9 | 41 | 36 | 33 | 45 |
| 72 | 27 | 22 | 90 | 81 | 59 | 54 | |

The elements of the array can be given as:

63, 19, 7, 9, 41, 36, 33, 45, 72, 27, 22, 90, 81, 59, 54

Repeat Step 1 with smaller number of long columns.

| | | | | |
|---|---|---|---|---|
| 63 | 19 | 7 | 9 | 41 |
| 36 | 33 | 45 | 72 | 27 |
| 22 | 90 | 81 | 59 | 54 |

*Result:*

| | | | | |
|---|---|---|---|---|
| 22 | 19 | 7 | 9 | 27 |
| 36 | 33 | 45 | 59 | 41 |
| 63 | 90 | 81 | 72 | 54 |

The elements of the array can be given as:

22, 19, 7, 9, 27, 36, 33, 45, 59, 41, 63, 90, 81, 72, 54

Repeat Step 1 with smaller number of long columns.

| | | |
|---|---|---|
| 22 | 19 | 7 |
| 9 | 27 | 36 |
| 33 | 45 | 59 |
| 41 | 63 | 90 |
| 81 | 72 | 54 |

*Result:*

| | | |
|---|---|---|
| 9 | 19 | 7 |
| 22 | 27 | 36 |
| 33 | 45 | 54 |
| 41 | 63 | 59 |
| 81 | 72 | 90 |

# Kabuk Sıralaması

The elements of the array can be given as:

9, 19, 7, 22, 27, 36, 33, 45, 54, 41, 63, 59, 81, 72, 90

Finally, arrange the elements of the array in a single column and sort the column.

*Result:*

| | |
|---|---|
| 9 | 7 |
| 19 | 9 |
| 7 | 19 |
| 22 | 22 |
| 27 | 27 |
| 36 | 33 |
| 33 | 36 |
| 45 | 41 |
| 54 | 45 |
| 41 | 54 |
| 63 | 59 |
| 59 | 63 |
| 81 | 72 |
| 72 | 81 |
| 90 | 90 |

Finally, the elements of the array can be given as:

7, 9, 19, 22, 27, 33, 36, 41, 45, 54, 59, 63, 72, 81, 90

# Kabuk Sıralaması

- ***Teknik***
- Kabuk sıralamasının çalışma şeklini görselleştirmek için aşağıdaki adımları gerçekleştirin:
  - *Adım 1: Dizi elemanlarını bir tablo şeklinde düzenleyin ve sütunları sıralayın (eklemeli sıralama kullanarak).*
  - *Adım 2: Adım 1'i her seferinde daha az sayıda ve daha uzun sütunlarla tekrarlayın; böylece sonunda sıralanacak yalnızca bir sütun veri kalır.*
- Burada sadece elemanların tabloda düzenlenmesini görselleştirdiğimizi, algoritmanın sıralama işlemini yerinde yaptığını unutmayın.

# Kabuk Sıralaması

- Eleman dizisini kabuk sıralaması kullanarak sıralama algoritması Şekil 14.13'te gösterilmiştir.
- Algoritmada Arr dizisinin elemanlarını birden fazla geçişte sıralıyoruz.
- Her geçişte, 4. Adımda yapıldığı gibi gap_size'ı (sütun sayısı olarak görselleştirin) yarı yarıya azaltıyoruz.
- Adım 5'teki for döngüsünün her yinelemesinde, dizinin değerlerini karşılaştırırız ve daha küçük değerden önce gelen daha büyük bir değer varsa bunları değiştiririz.

```
Shell_Sort(Arr, n)

Step 1: SET FLAG = 1, GAP_SIZE = N
Step 2: Repeat Steps 3 to 6 while FLAG = 1 OR GAP_SIZE > 1
Step 3:     SET FLAG = 0
Step 4:     SET GAP_SIZE = (GAP_SIZE + 1) / 2
Step 5:     Repeat Step 6 for I = 0 to I < (N - GAP_SIZE)
Step 6:           IF Arr[I + GAP_SIZE] > Arr[I]
                        SWAP Arr[I + GAP_SIZE], Arr[I]
                        SET FLAG = 0
Step 7: END
```

**Figure 14.13** Algorithm for shell sort

# Kabuk Sıralaması

**PROGRAMMING EXAMPLE**

**12.** Write a program to implement shell sort algorithm.

```
#include<stdio.h>
void main()
{
        int arr[10]={-1};
        int i, j, n, flag = 1, gap_size, temp;
        printf("\n Enter the number of elements in the array: ");
        scanf("%d", &n);
        printf("\n Enter %d numbers: ",n); // n was added
        for(i=0;i<n;i++)
        scanf("%d", &arr[i]);
        gap_size = n;
        while(flag == 1 || gap_size > 1)
        {
                flag = 0;
                gap_size = (gap_size + 1) / 2;
                for(i=0; i< (n - gap_size); i++)
                {
                        if( arr[i+gap_size] < arr[i])
                        {
                                temp = arr[i+gap_size];
                                arr[i+gap_size] = arr[i];
                                arr[i] = temp;
                                flag = 0;
                        }
                }
        }
        printf("\n The sorted array is: \n");
        for(i=0;i<n;i++){
                printf(" %d\t", arr[i]);
        }
}
```

# Ağaç Sıralaması

- Ağaç sıralaması, ikili arama ağacının özelliklerini kullanarak sayıları sıralayan bir sıralama algoritmasıdır.
- Algoritma, önce sıralanacak sayıları kullanarak ikili bir arama ağacı oluşturur ve ardından sayıların sıralı bir düzende alınması için sıralı bir gezinme gerçekleştirir.

# Ağaç Sıralaması

**PROGRAMMING EXAMPLE**

13. Write a program to implement tree sort algorithm.

```
#include <stdio.h>
#include <conio.h>
#include <alloc.h>
struct tree
{
        struct tree *left;
        int num;
        struct tree *right;
} ;
void insert (struct tree **, int);
void inorder (struct tree *);
void main( )
{
        struct tree *t ;
        int arr[10];
        int i ;
        clrscr( ) ;
        printf("\n Enter 10 elements : ");
        for(i=0;i<10;i++)
                scanf("%d", &arr[i]);
        t = NULL ;
        printf ("\n The elements of the array are : \n" ) ;
        for (i = 0 ; i <10 ; i++)
                printf ("%d\t", arr[i]) ;
        for (i = 0 ; i <10 ; i++)
                insert (&t, arr[i]) ;
        printf ("\n The sorted array is : \n") ;
        inorder (t ) ;
        getche( ) ;
}
```

# Ağaç Sıralaması

```
void insert (struct tree **tree_node, int num)
{
        if ( *tree_node == NULL )
        {
                *tree_node = malloc (sizeof ( struct tree )) ;
                ( *tree_node ) -> left = NULL ;
                ( *tree_node ) -> num = num ;
                ( *tree_node ) -> right = NULL ;
        }
        else
        {
                if ( num < ( *tree_node ) -> num )
                        insert ( &( ( *tree_node ) -> left ), num ) ;
                else
                        insert ( &( ( *tree_node ) -> right ), num ) ;
        }
}
void inorder (struct tree *tree_node )
{
        if ( tree_node != NULL )
        {
                inorder ( tree_node -> left ) ;
                printf ( "%d\t", tree_node -> num ) ;
                inorder ( tree_node -> right ) ;
        }
}
```

# Sıralama Algoritmalarının Karşılaştırılması

- Tablo 14.1, şu ana kadar tartışılan farklı sıralama algoritmalarının ortalama ve en kötü durum zaman karmaşıklıklarını karşılaştırmaktadır.

**Table 14.1** Comparison of algorithms

| Algorithm | Average Case | Worst Case |
|---|---|---|
| Bubble sort | $O(n^2)$ | $O(n^2)$ |
| Bucket sort | $O(n.k)$ | $O(n^2.k)$ |
| Selection sort | $O(n^2)$ | $O(n^2)$ |
| Insertion sort | $O(n^2)$ | $O(n^2)$ |
| Shell sort | – | $O(n \log^2 n)$ |
| Merge sort | $O(n \log n)$ | $O(n \log n)$ |
| Heap sort | $O(n \log n)$ | $O(n \log n)$ |
| Quick sort | $O(n \log n)$ | $O(n^2)$ |