# S3 Program

- Reminders
- Chapter 1 : Algorithmic complexity
- **Chapter 2 : Sorting Algorithms**
- Chapter 3 : Trees
- Chapter 4 : Graphs

# Chapter II –
# SORTING Algorithms

# Introduction of the problem

WHAT IS THE PROBLEM TO BE SOLVED?
Given a list of numbers: (5 2 14 1 6)
We want to sort it: (1 2 5 6 14) or (14 6 5 2 1)

Sorting is arguably the fundamental problem in algorithms.
1. More than 25% of CPU cycles are spent sorting.
2. Sorting is fundamental to many other problems, such as binary search.

Thus, after sorting, many problems become easy to solve. For example:
1. Uniqueness of elements: after sorting, test adjacent elements.
2. Once sorted, the $k^{th}$ largest element can be determined in O(1).

The sorting problems discussed in this chapter are those where all the data to be sorted is located in main memory. Sorting problems where the data is located in secondary memory are not discussed in this chapter.

**How to sort? There are several solutions:**

**1) Bubble sort**

**2) Selection sort**

**3) Insert sort**

**4) Merge sort**

**5) Quick sort**

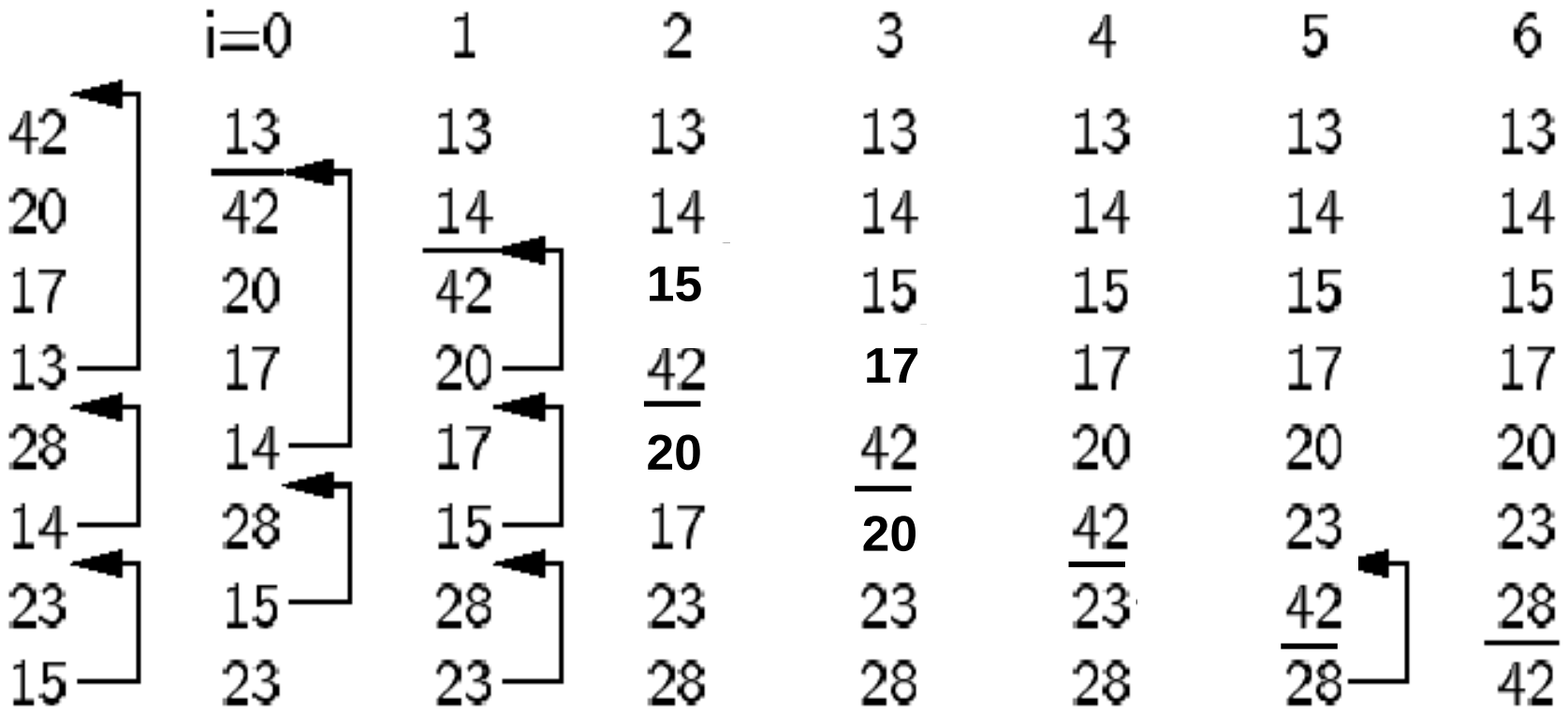
## 1) Bubble Sort

The strategy for this algorithm is as follows:

1. Scan the array, comparing successive elements in pairs, swapping them if they are out of order.

2. Repeat until all swaps are completed.

# Example :


i=0
1
2
3
4
5
6
42 20 17 13 28 14 23 15
13 42 20 17 14 28 15 23
13 14 42 20 17 15 28 23
13 14 15 42 17 20 23 28
13 14 15 17 42 20 23 28
13 14 15 17 20 42 23 28
13 14 15 17 20 23 42 28
13 14 15 17 20 23 28 42

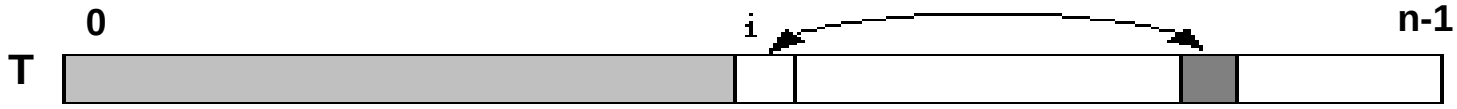
**Exercise 01:** Program the bubble sort algorithm for ascending order.

**Exercise 02:** Study the complexity of the above bubble sort algorithm.

## 2) Selection sort

Repeat

1. Find the largest (smallest) element
2. Place it at the end (beginning)




**The i-1 smallest sorted elements**

**the smallest element of the remainder**

| | i=0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|---|
| 42 | 13 | 13 | 13 | 13 | 13 | 13 | 13 |
| 20 | 20 | 14 | 14 | 14 | 14 | 14 | 14 |
| 17 | 17 | 17 | 15 | 15 | 15 | 15 | 15 |
| 13 | 42 | 42 | 42 | 17 | 17 | 17 | 17 |
| 28 | 28 | 28 | 28 | 28 | 20 | 20 | 20 |
| 14 | 14 | 20 | 20 | 20 | 28 | 23 | 23 |
| 23 | 23 | 23 | 23 | 23 | 23 | 28 | 28 |
| 15 | 15 | 15 | 17 | 42 | 42 | 42 | 42 |

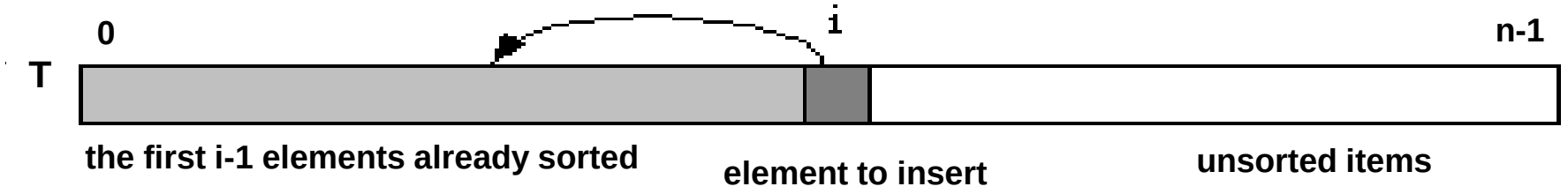
**Exercise 03:** Program the selection sort algorithm for ascending order.

**Exercise 04:** Study the complexity of the above selection sort algorithm.

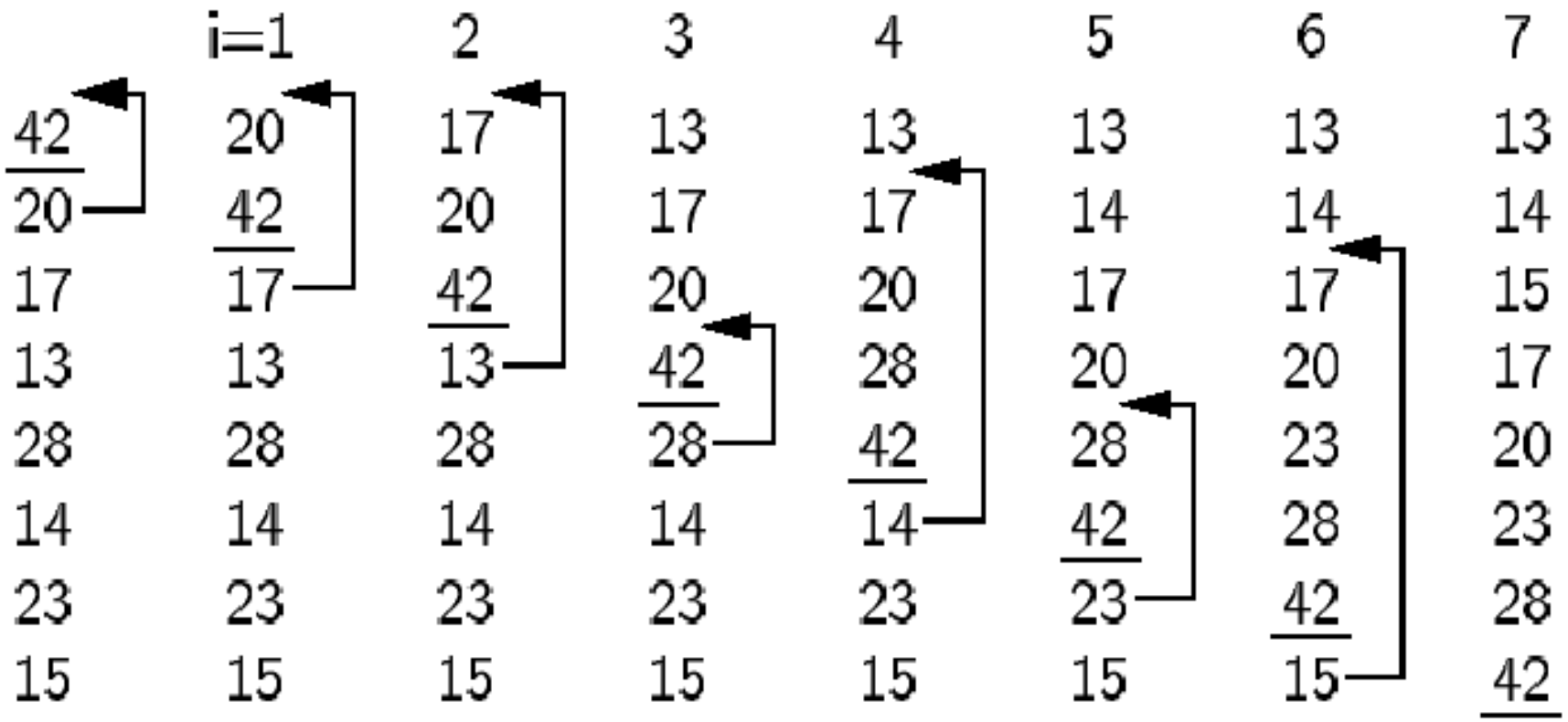
## 3) Insertion Sort

In this case, we iteratively insert the next element into the previously sorted part. The starting part that is sorted is the first element.

By inserting an element into the sorted part, we may have to move several others.



**Example**

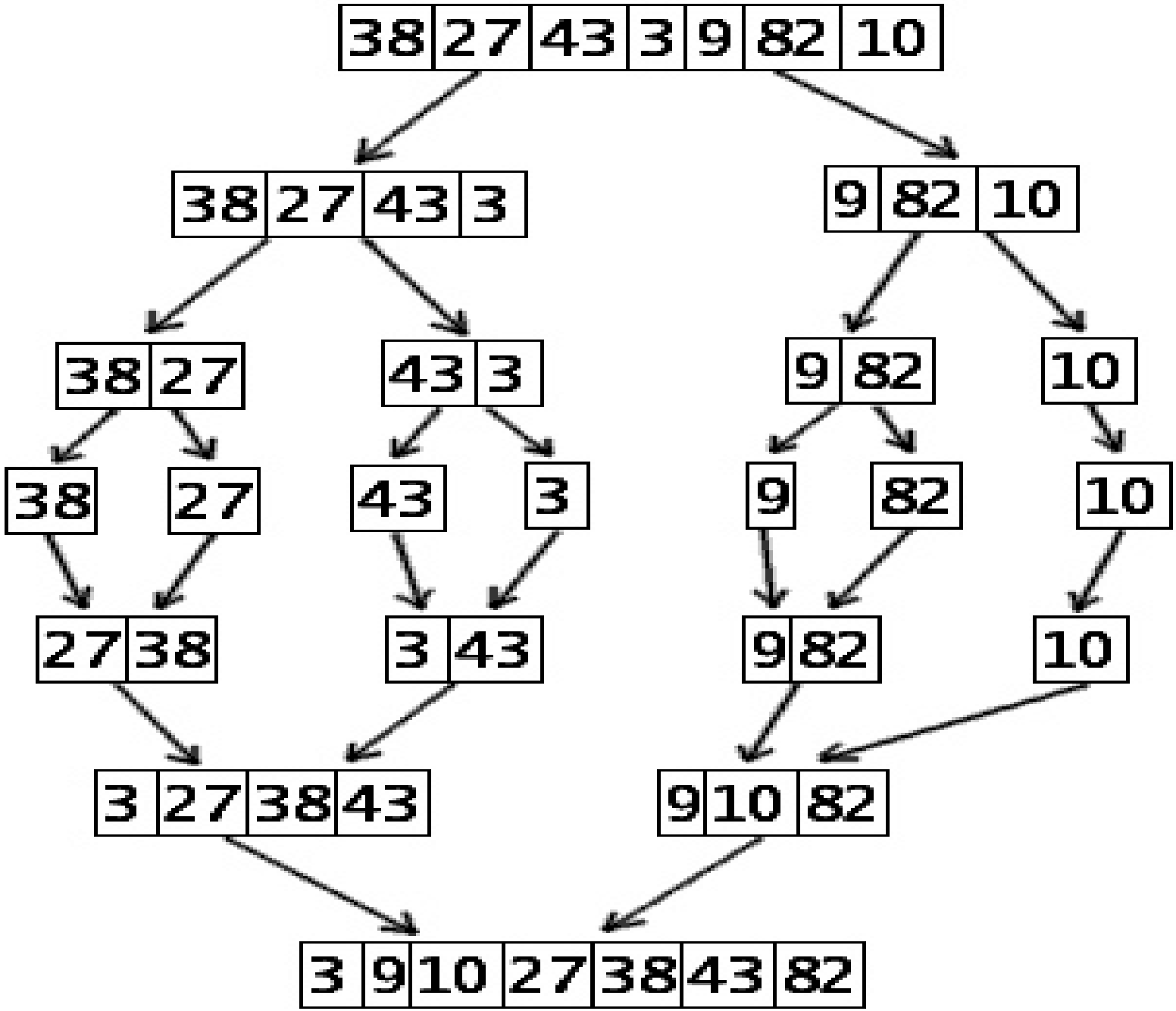

i=1
2
3
4
5
6
7
42
20
17
13
28
14
23
15
20
42
17
13
28
14
23
15
17
20
42
13
28
14
23
15
13
17
20
42
28
14
23
15
13
17
20
28
42
14
23
15
13
14
17
20
28
42
23
15
13
14
15
17
20
23
28
42
13
14
15
17
20
23
28
42

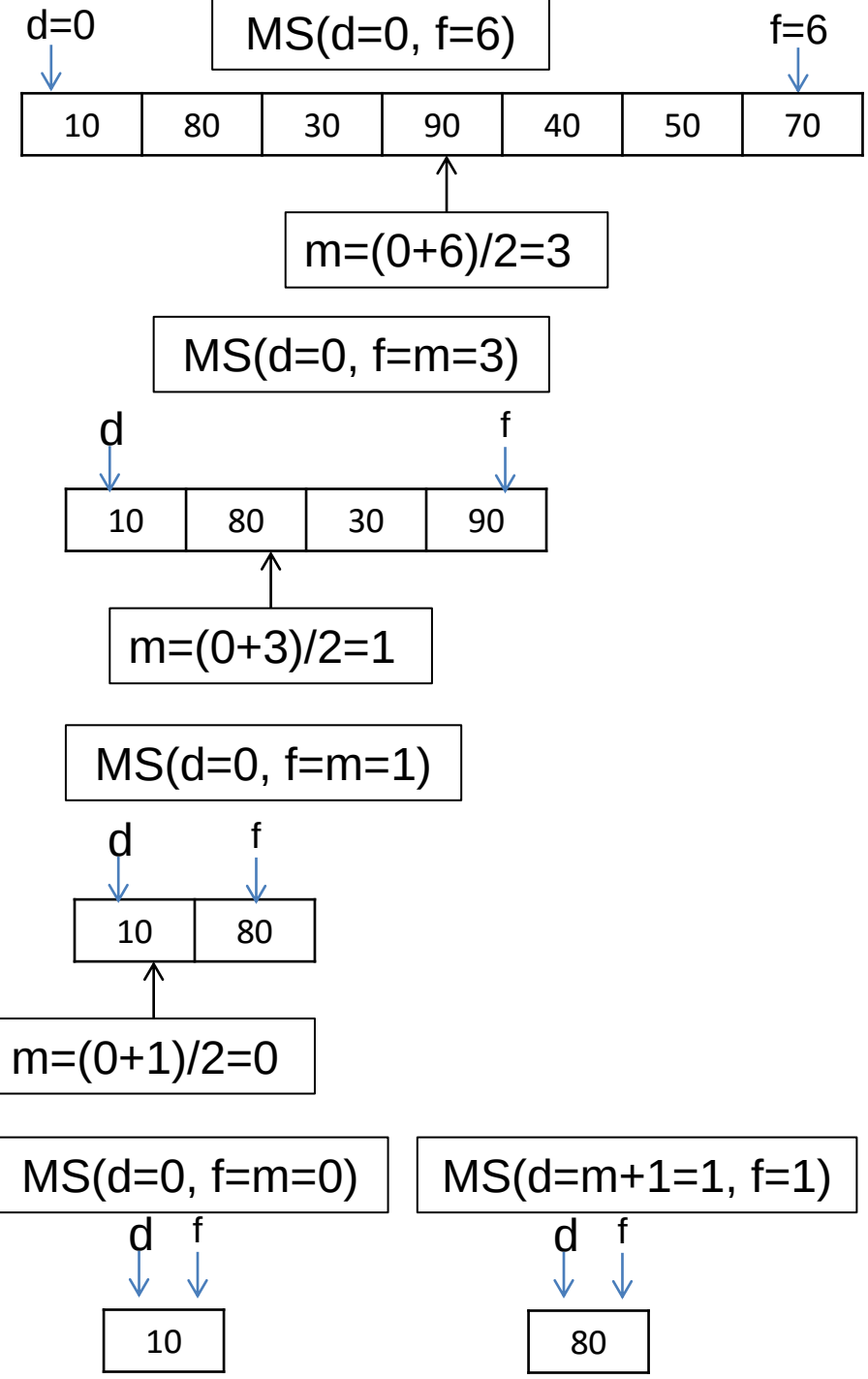
**Exercise 05:** Program the insertion sort algorithm for ascending order.

**Exercise 06:** Study the complexity of the above insertion sort algorithm.

## 4) Merge Sort

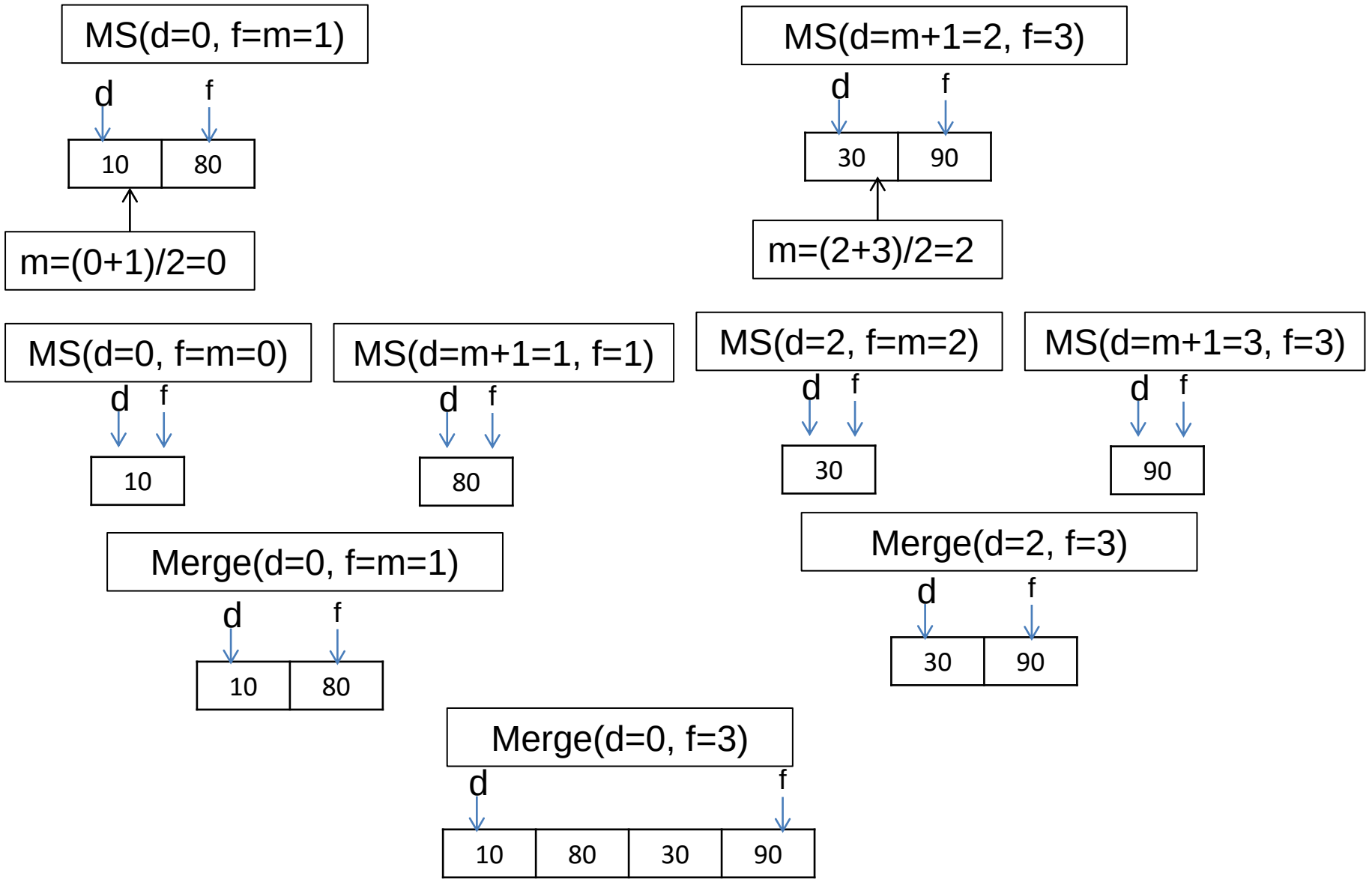
- This algorithm divides the data array in question into two equal parts.
- After these two parts are recursively sorted, they are merged to sort the entire data set. Note that this merging must take into account the fact that these parts are already sorted.

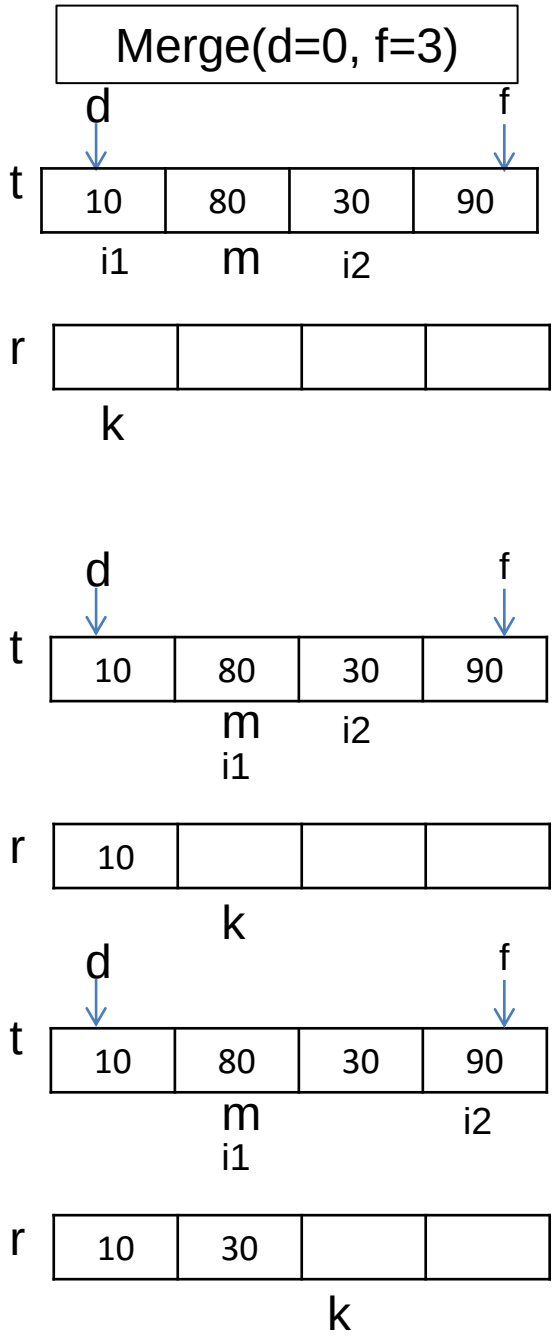

38 27 43 3 9 82 10
38 27 43 3
9 82 10
38 27
43 3
9 82
10
38
27
43
3
9
82
10
27 38
3 43
9 82
10
3 27 38 43
9 10 82
3 9 10 27 38 43 82




**Implementation of the merge sort algorithm 1**

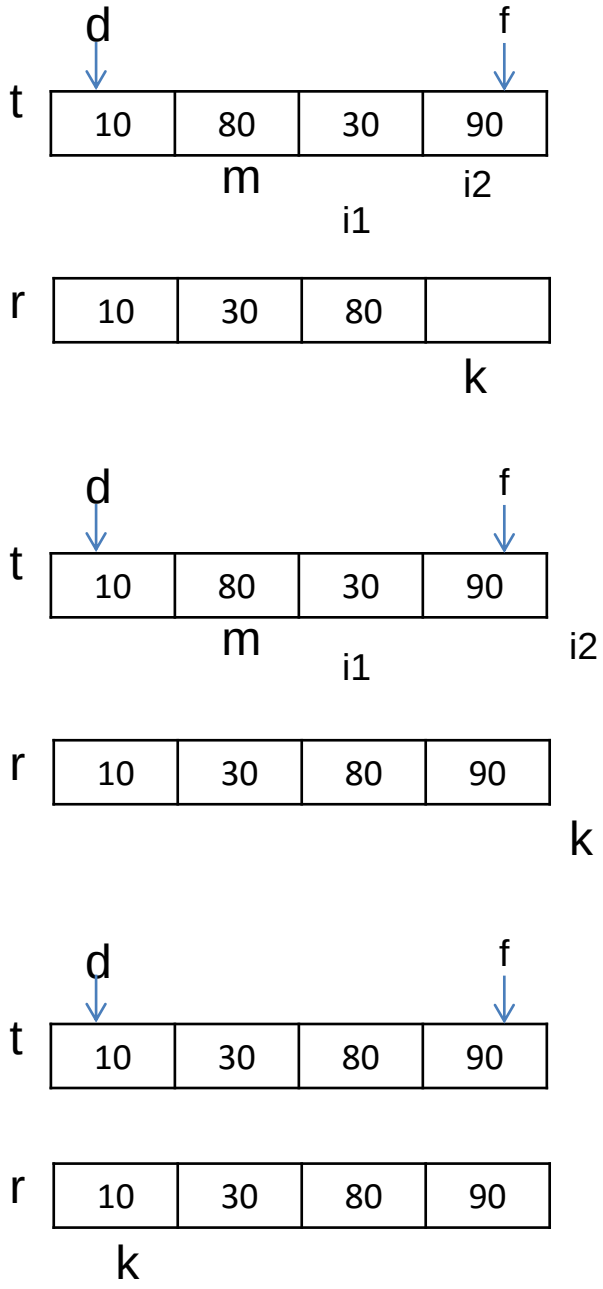
```
void MergeSort (int t[],int n, int d, int f) {
   if (d<f) {  int m=(d+f)/2;
                  MergeSort(t,n,d,m);
                  MergeSort(t,n,m+1,f);
                  Merge(t,n,d,f); }       }

void Merge(int t[], int  n, int d, int f) {
int r[f-d+1];
int m=(d+f)/2;
int i1=d, i2=m+1, k=0;
while (i1<=m && i2 <=f)  {
            if (t[i1] < t[i2])  {  r[k] = t[i1];  i1++;}
                     else   {  r[k] = t[i2];  i2++; }
            k++;                          }
while (i1 <=m)  {  r[k] = t[i1]; i1++; k++; }
while (i2 <=f)    { r[k] = t[i2]; i2++; k++; }

for(k=0; k<=f-d; k++)  t[d+k]=r[k];     }
```


MS(d=0, f=m=1)
d
f
10
80
m=(0+1)/2=0
MS(d=m+1=2, f=3)
d
f
30
90
m=(2+3)/2=2
MS(d=0, f=m=0)
d
f
10
MS(d=m+1=1, f=1)
d
f
80
MS(d=2, f=m=2)
d
f
30
MS(d=m+1=3, f=3)
d
f
90
Merge(d=0, f=m=1)
d
f
10
80
Merge(d=2, f=3)
d
f
30
90
Merge(d=0, f=3)
d
f
10
80
30
90




**Implementation of the merge sort algorithm 1**

```
void MergeSort (int t[],int n, int d, int f) {
   if (d<f) {  int m=(d+f)/2;
                MergeSort(t,n,d,m);
                MergeSort(t,n,m+1,f);
                Merge(t,n,d,f); }       }

void Merge(int t[], int  n, int d, int f) {
int r[f-d+1];
int m=(d+f)/2;
int i1=d, i2=m+1, k=0;
while (i1<=m && i2 <=f)  {
            if (t[i1] < t[i2])  {  r[k] = t[i1];  i1++;}
                    else   {  r[k] = t[i2];  i2++; }
            k++;                          }
while (i1 <=m)  {  r[k] = t[i1]; i1++; k++; }
while (i2 <=f)    { r[k] = t[i2]; i2++; k++; }

for(k=0; k<=f-d; k++)  t[d+k]=r[k];     }
```




**Implementation of the merge sort algorithm 1**

```
void MergeSort (int t[],int n, int d, int f) {
   if (d<f) {  int m=(d+f)/2;
                MergeSort(t,n,d,m);
                MergeSort(t,n,m+1,f);
                Merge(t,n,d,f); }       }

void Merge(int t[], int  n, int d, int f) {
int r[f-d+1];
int m=(d+f)/2;
int i1=d, i2=m+1, k=0;
while (i1<=m && i2 <=f)  {
            if (t[i1] < t[i2])  {  r[k] = t[i1];  i1++;}
                    else   {  r[k] = t[i2];  i2++; }
            k++;                        }
while (i1 <=m)  {  r[k] = t[i1]; i1++; k++; }
while (i2 <=f)    { r[k] = t[i2]; i2++; k++; }

for(k=0; k<=f-d; k++)  t[d+k]=r[k];     }
```

**Exercise 07:** Program the merge sort algorithm for ascending order.

**Exercise 08:** Study the complexity of the above merge sort algorithm.

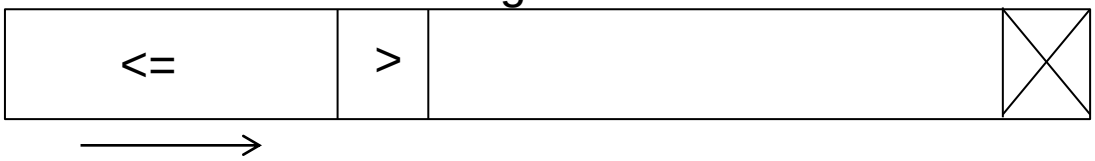
## 5) QuicSort

The strategy of the quicksort algorithm consists, firstly, in dividing the table into two parts separated by an element (called pivot) in such a way that the elements of the left part are all less than or equal to this element and those of the right part are all greater than this pivot (the part which performs this task is called partition). Then, in a recursive manner, this process is iterated on the two parts thus created.

**Choosing the pivot:** The ideal choice would be to cut the array exactly into two equal parts. But this is not always possible. You can take the first or last element. But there are several other strategies!

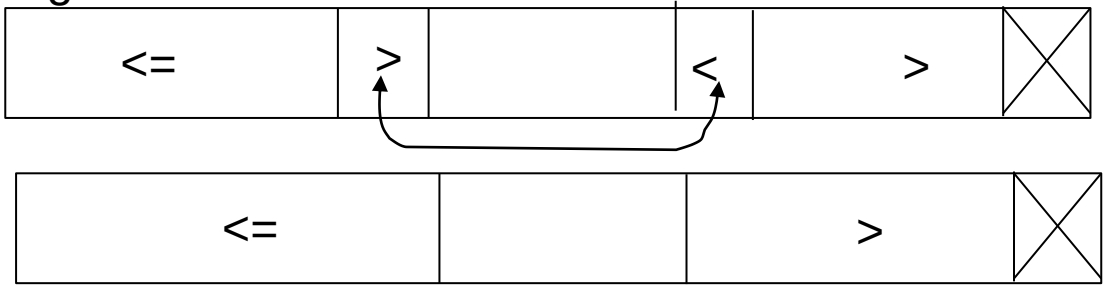
**Partitioning :**

- We traverse the table from left to right until we encounter an element greater than the pivot



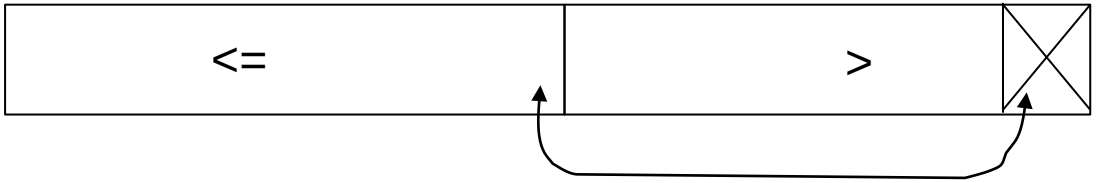

- We traverse the table from right to left until we encounter an element less than the pivot

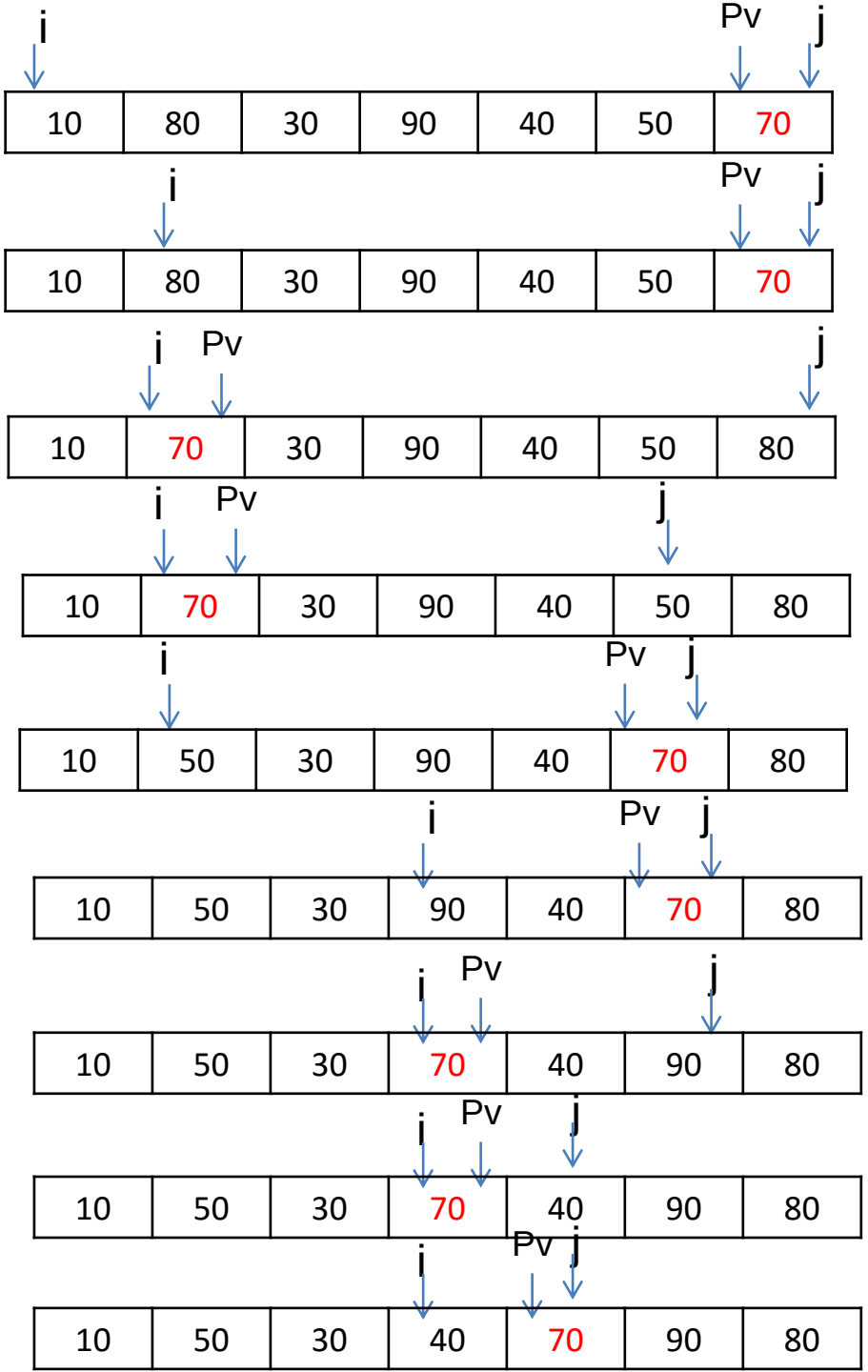

- We exchange these two elements




and we repeat the left-right and right-left paths until we have:


Then it is enough to put the pivot at the border (by an exchange)

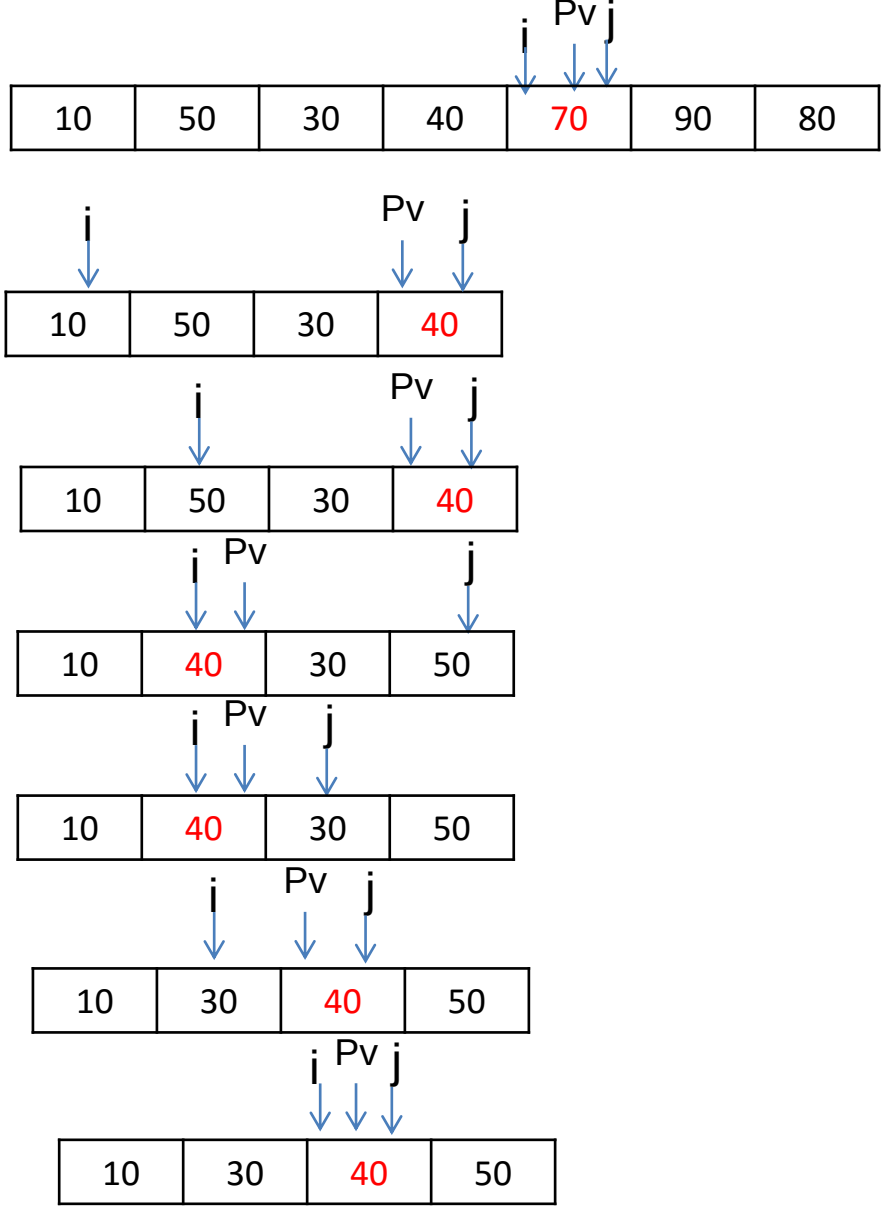





**QuickSort Implementation**

```
void tri_rapide_bis(int tab[], int debut, int fin) {
if (debut<fin)                                    {
            int piv=partition(tab, debut, fin);
            tri_rapide_bis(tab,debut,piv-1);
            tri_rapide_bis(tab,piv+1,fin); }    }

void tri_rapide(int tab[], int n)
{  tri_rapide_bis(tab,0,n-1); }

void echanger(int tab[], int i, int j)  {
                        int memoire=tab[i];
                        tab[i]=tab[j];
                        tab[j]=memoire; }
```

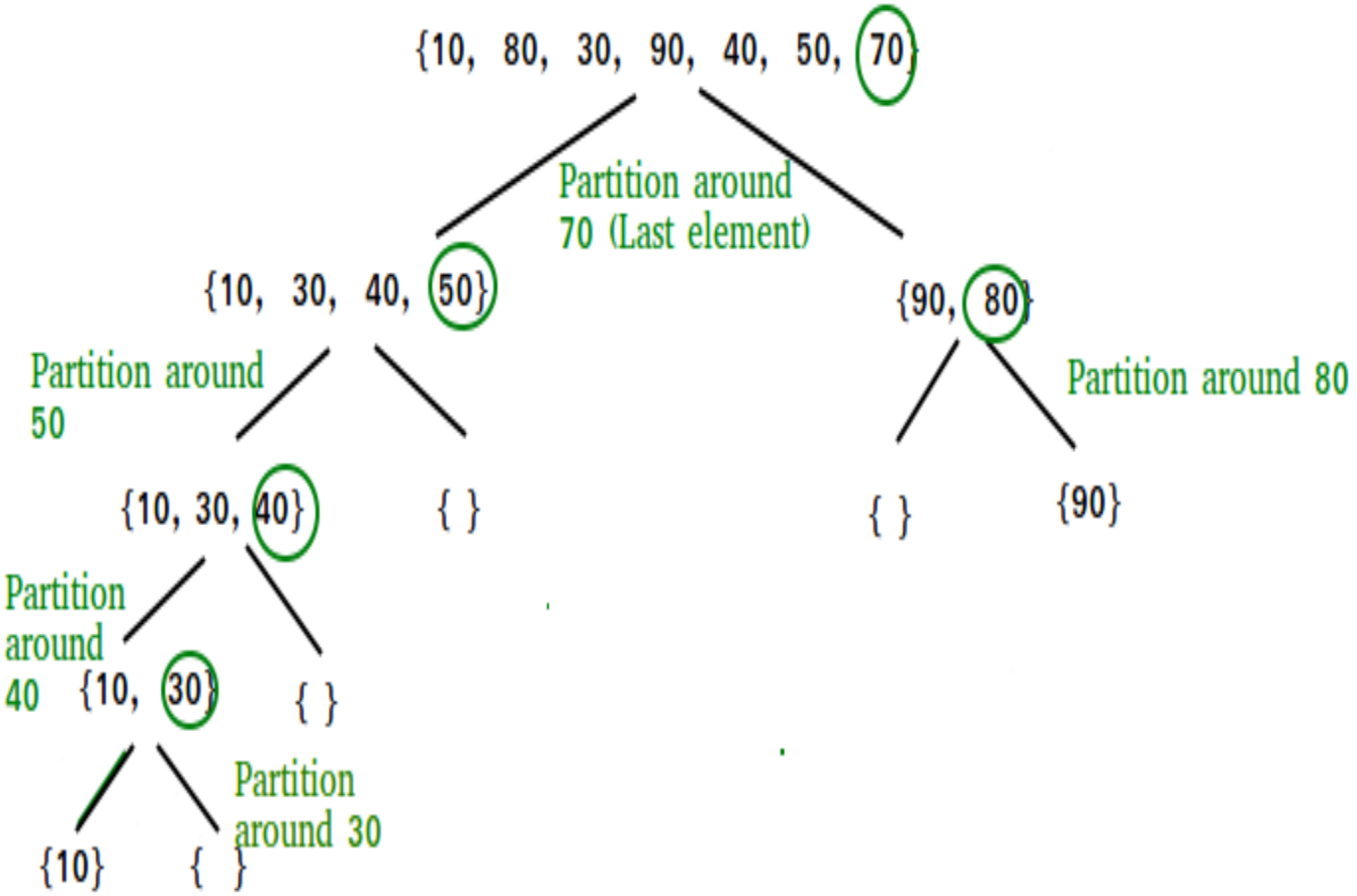
```
int partition(int tab [], int deb, int fin) {
int pivot = tab [fin];
int i = debut ; j = fin;
do {        while (tab [i] < pivot) i++;
            while (tab [j] > pivot) j--;
            if (i < j) echanger (tab, i, j);
     }  while (i < j);
return(j); }
```



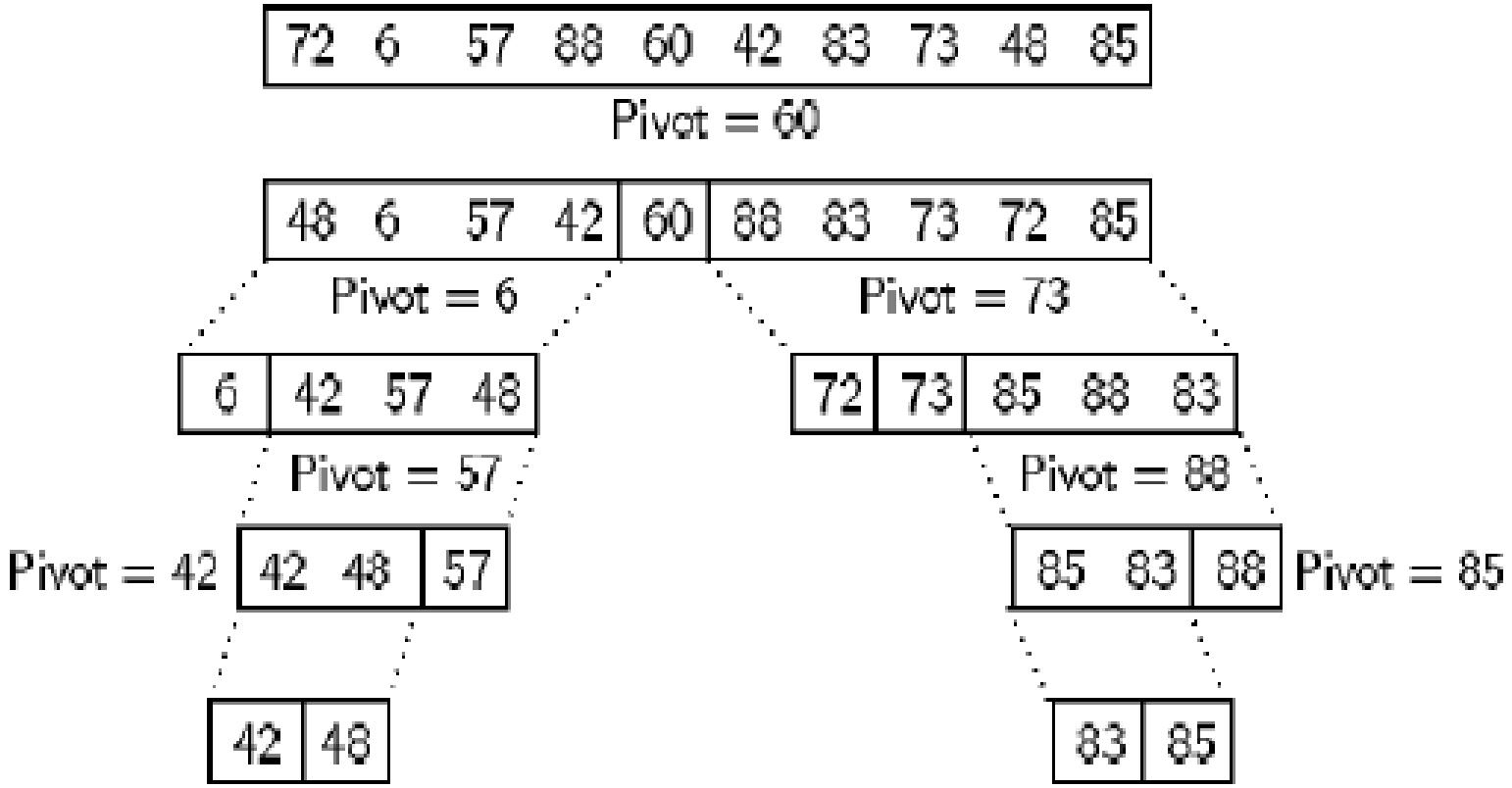

**QuickSort Implementation**

```
void tri_rapide_bis(int tab[], int debut, int fin) {
if (debut<fin)                                    {
            int pivot=partition(tab, debut, fin);
            tri_rapide_bis(tab,debut,pivot-1);
            tri_rapide_bis(tab,pivot+1,fin); }    }

void tri_rapide(int tab[], int n)
{  tri_rapide_bis(tab,0,n-1); }

void echanger(int tab[], int i, int j)  {
                        int memoire=tab[i];
                        tab[i]=tab[j];
                        tab[j]=memoire; }
```

```
int partition(int tab [], int deb, int fin) {
int pivot = tab [fin];
int i = debut ; j = fin;
do {        while (tab [i] < pivot) i++;
            while (tab [j] > pivot) j--;
            if (i < j) echanger (tab, i, j);
     }  while (i < j);
return(j); }
```


{10, 80, 30, 90, 40, 50, 70}
Partition around
70 (Last element)
{10, 30, 40, 50}
{90, 80}
Partition around
50
Partition around 80
{10, 30, 40}
{ }
{ }
{90}
Partition
around
40
{10, 30}
{ }
Partition
around 30
{10}
{ }

**Example**


72 6 57 88 60 42 83 73 48 85
Pivot = 60
48 6 57 42 60 88 83 73 72 85
Pivot = 6
Pivot = 73
6 42 57 48
72 73 85 88 83
Pivot = 57
Pivot = 88
Pivot = 42
42 48 57
85 83 88
Pivot = 85
42 48
83 85

**Exercise 09:** Program the quick sort algorithm for ascending order.

**Exercise 10:** Study the complexity of the above quick sort algorithm.

# Algos and complexities

```
void bubsort(Elem* tab, int n) { // Bubble Sort
for (int i=0; i<n-1; i++) // échanger
for (int j=n-1; j>i; j--)
        if ( tab[j] < tab [j-1] )
                        swap(tab, j, j-1);
}
```

## ***Complexity of bubble sort***

- The number of comparisons "if Tab[ j-1 ] > Tab[ j ] then" is a value that depends only on the size n of the list; this number is equal to the number of iterations.
- The count shows that the loop "for i from n to 1 do" executes n times (thus a sum of n terms) and that each time the loop "for j from 2 to i do" executes (i-2) + 1 times the comparison "if Tab[ j-1 ] > Tab[ j ] then".
- The complexity in number of comparisons is equal to the sum of the following n terms (i = n, i = n-1,....)

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

(it is the sum of the first n-1 integers).

**Selection sort Implementation**

```
void selsort(Elem* array, int n)
{
for (int i=0; i<n-1; i++) {       // Select the ith element
   int lowindex = i;              // memorize this index
   for (int j=n-1; j>i; j--)      // find the smallest value
   if (key(array[j]) < key(array[lowindex])) lowindex = j; // update the index
   swap(array, i, lowindex); // exchange values
}                          }
```

**Complexity :** The worst case, best case and average case are the same (why?)
To find the smallest element, (n-1) iterations are required, for the 2nd smallest element, (n-2) iterations are performed, …. To find the last smallest element, 0 iterations are performed. Therefore, the number of iterations the algorithm performs is :

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

(it is the sum of the first n-1 integers).
If, on the other hand, we take the number of data movements as the evaluation measure, then the algorithm performs n-1, because there is exactly one exchange per iteration.

## Insertion sort algorithm

```
void inssort(Elem* array, int n)
{
for (int i=1; i<n; i++) // insérer le ième element
for (int j=i; (j>0) && (key(array[j])<key(array[j-1])); j--)
        swap(array, j, j-1);
}
```

**Complexity :** Since we don't necessarily have to scan the entire sorted section, the worst case, best case, and average case may differ.

**Best case**: Each element is inserted at the end of the sorted section. In this case, we don't have to move any elements. Since we have to insert (n-1) elements, each generating only one comparison, the complexity is O(n).

**Worst case:** Each element is inserted at the beginning of the sorted section. In this case, all elements in the sorted section must be moved at each iteration. The ith iteration generates (i-1) comparisons and value swaps:

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

**Note:** This is the same number of comparisons as selection sort, but performs more value swaps. If the values to be swapped are large, this number can significantly slow down this algorithm.

**Average case:** Given a random permutation of numbers, the probability of inserting the element at the $k^{th}$ position among the positions (0, 1, 2, …, i-1) is 1/i. Therefore, the average number of comparisons at the $i^{th}$ iteration is:

$$\sum_{k=1}^{i}\frac{(k-1)}{i} = \frac{1}{i}\sum_{k=1}^{i-1}k = \frac{1}{i}\times\frac{i(i-1)}{2} = \frac{i-1}{2}$$

Summing over i, we obtain:

$$\sum_{i=1}^{n}\frac{i-1}{2} = \frac{1}{2}\left(\frac{n(n-1)}{2}+\frac{n}{2}\right) = \frac{n^2}{4}+O(n)$$

**Implementation of the merge sort algorithm 1**

```
void MergeSort (int t[],int n, int d, int f) {
   if (d<f) {  int m=(d+f)/2;
               MergeSort(t,n,d,m);
               MergeSort(t,n,m+1,f);
               Merge(t,n,d,f);          }       }

void Merge(int t[], int  n, int d, int f) {
int r[f-d+1];
int m=(d+f)/2;
int i1=d, i2=m+1, k=0;
while (i1<=m && i2 <=f)  {
            if (t[i1] < t[i2])  {  r[k] = t[i1];  i1++;}
                        else   {  r[k] = t[i2];  i2++; }
            k++;                    }
while (i1 <=m)  {  r[k] = t[i1]; i1++; k++; }
while (i2 <=f)    { r[k] = t[i2]; i2++; k++; }

for(k=0; k<=f-d; k++)  t[d+k]=r[k];     }
```

**Implementation of the merge sort algorithm 2**

```
void mergesort(Elem* array, Elem* temp, int left, int right)
{
int i, j, k, mid = (left+right)/2;       // mid=left+(right-left)/2  ;  other possibility
if (left == right) return;
mergesort(array, temp, left, mid); // the first half
mergesort(array, temp, mid+1, right);// Sort 2nd half
// the merge operation. First, copy both halves into temp.
for (i=left; i<=mid; i++)
temp[i] = array[i];
for (j=1; j<=right-mid; j++)
temp[right-j+1] = array[j+mid];
// merge the two halves in array
for (i=left,j=right,k=left; k<=right; k++)
if (key(temp[i]) < key(temp[j]))
array[k] = temp[i++];
else array[k] = temp[j--];
}
```

**Complexity:**

The complexity of this algorithm is given by the following relation:

$$T(n) = \begin{cases} O(1) & n = 1, \\ T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + O(n) & n > 1, \end{cases}$$

n being the number of elements in the array.

To simplify the solution of the previous equation, we assume that $n=2^k$ for an integer $k \geq 0$. Replacing O(n) with n, we obtain (in principle, it should be replaced with cn):

$$T(n) = \begin{cases} 1 & n = 1, \\ 2T(n/2) + n & n > 1, \end{cases}$$

$$\begin{aligned} T(n) &= 2T(n/2) + n \\ &= 4T(n/4) + 2n \\ &= 8T(n/8) + 3n \\ &\vdots \\ &= 2^k T(n/2^k) + kn \\ &\vdots \\ &= nT(1) + n\log_2 n \\ &= n + n\log_2 n \end{aligned}$$

The time complexity of merge sort is therefore O(n log n).

**QuickSort Implementation**

```
void tri_rapide_bis(int tableau[], int debut, int fin) {
if (debut<fin)                                  {
            int pivot=partition(tableau, debut, fin);
            tri_rapide_bis(tableau,debut,pivot-1);
            tri_rapide_bis(tableau,pivot+1,fin); }     }

void tri_rapide(int tableau[], int n)
{  tri_rapide_bis(tableau,0,n-1); }

void echanger(int tab[], int i, int j)  {  int memoire;
                                           memoire=tab[i];
                                           tab[i]=tab[j];
                                           tab[j]=memoire; }

int partition(int tableau[], int deb, int fin) {
int pivot = tableau[fin];
int i = debut ; j = fin;
do {        while (tableau[i] < pivot) i++;
            while (tableau[j] > pivot) j--;
            if (i < j) echanger (tableau, i, j);
     }  while (i < j);
return(j); }
```

**Complexity**

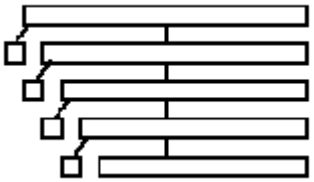
When QSORT(1,n) is called, the pivot is placed at position i. This leaves us with a problem of sorting two subparts of size i-1 and n-i. The partition algorithm clearly has a complexity of at most cn for a constant c.

$T(n)= T(i-1 ) + T(n-i)+cn$

$T(1)=1$

Let's look at the three possible complexity cases:

**Worst case (maximum complexity):**



The pivot is always the smallest element. The recurrence relation becomes :

$T(n) = T(n – 1) + cn$ (pourquoi ?)

$T(n -1) = T(n - 2) + c(n - 1)$

$T(n - 2) = T(n - 3) + c(n - 2)$

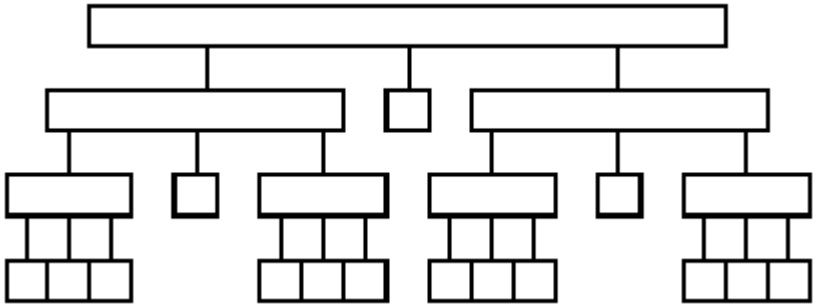
... …….

$T(2) = T(1) + c(2)$

Adding member to member, we get:

$T(n) = O( n^2 )$

**Best case (min complexity):**



In the best case, the pivot is, each time, located in the middle of the part to be sorted. $T(n) = 2T(n/2) + cn$

This development stops as soon as we reach T(1). In other words, as soon as $n / 2k = 1$

$n = 2k \Rightarrow k = \log n$

Final Solution : $T(n) = O(n \log n)$