Sorting الترتيب

• **Sorting** is the process of arranging a group of items into a defined order based on particular criteria, e.g. arranging Person information in ascending order of age.
Sorting algorithms are often classified by:
1-stability
2-time
3-space (consider memory usage).
4-adaptability
5-Computational complexity theory: (See Big O notation.)

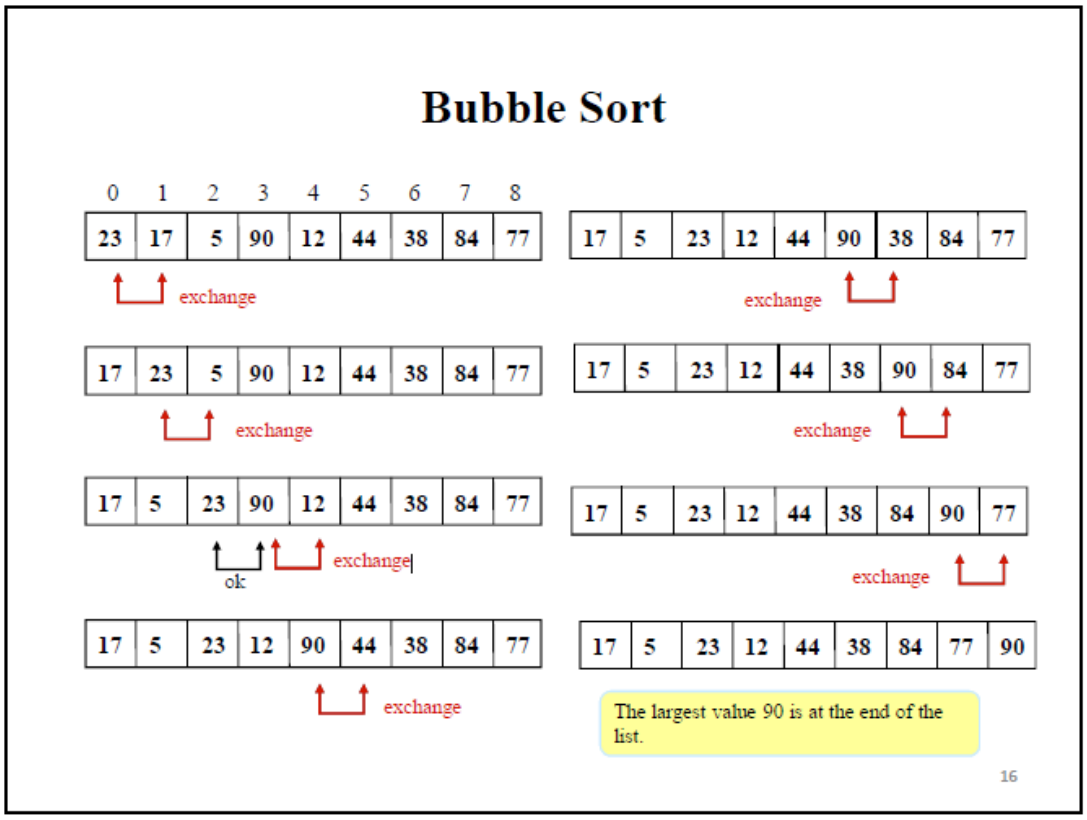
هذه النظرية تقيس عدد التبديلات وعددالمقارنات في كل دورة وتعتمد على ثلاث حالات:

1- worst ,
2- average ,
3- best behavior.
in terms of the size of the list (n). For typical serial sorting algorithms good behavior is O(n log n), with parallel sort in O($\log^2 n$), and bad behavior is O(n^2). Ideal behavior for a serial sort is O(n).

• There are different types of the sorting algorithm such as:
Bubble Sort, Selection Sort, Insertion Sort, Quick Sort and Merge Sort.

1-Bubble Sort:

• The bubble sort improves the performance by making more than one exchange during its pass.
• By making multiple exchanges, we will be able to move more elements toward their correct positions using the same number of comparisons as the selection sort makes.
• The key idea of the bubble sort is to make pair wise comparisons and exchange the positions of the pair if they are out of order.



Bubble sort Algorithm:

We assume **list** is an array of **n** elements. We further assume that **swap** function swaps the values of the given array elements.

```
begin BubbleSort(list)
  for all elements of list
    if list[i] > list[i+1]
      swap(list[i], list[i+1])
    end if
  end for
   return list
end BubbleSort
```

Sorting using Bubble Sort program:

Let's consider an array with values {5, 1, 6, 2, 4, 3}

```
int a[6] = {5, 1, 6, 2, 4, 3};
int i, j, temp;
for(i=0; i<6; i++)
{
  for(j=0; j<6-i-1; j++)
  {
    if( a[j] > a[j+1])
    {
      temp = a[j];
      a[j] = a[j+1];
      a[j+1] = temp;
    }
  }
}
```

Above is the algorithm, to sort an array using Bubble Sort. Although the above logic will sort and unsorted array, still the above algorithm is not efficient because as per the above logic, the for-loop will keep executing for six iterations even if the array gets sorted after the second iteration.

Hence we can insert a flag and can keep checking whether swapping of elements is taking place or not in the following iteration.

```
int a[6] = {5, 1, 6, 2, 4, 3};
int i, j, temp;
for(i=0; i<6; i++)
{
  int flag = 0;        //taking a flag variable
  for(j=0; j<6-i-1; j++)
  {
    if( a[j] > a[j+1])
    {
      temp = a[j];
```

```
    a[j] = a[j+1];
    a[j+1] = temp;
    flag = 1;        //setting flag as 1, if swapping occurs
   }
 }
for(i=0;i<6;i++)
 cout<<a[i]<<"  ";
```

H.W:

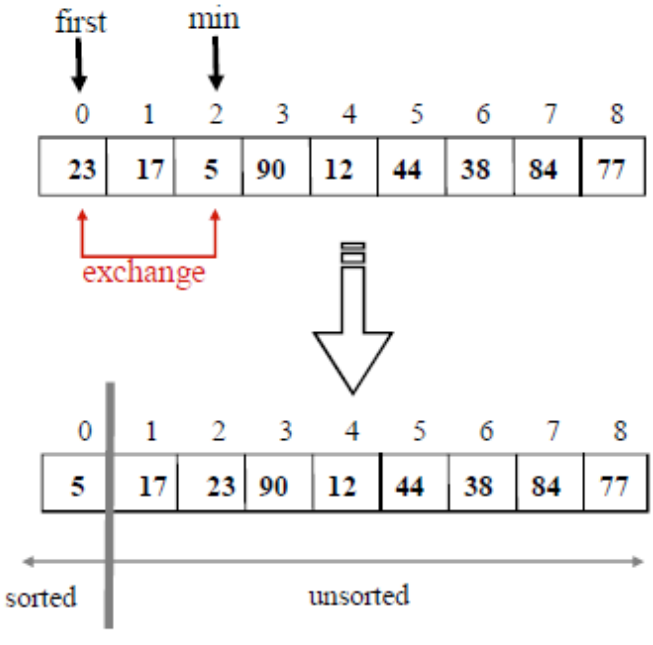
1-write program to sort the following number by use bubble sort function: 8,6,4,2

2-suppose that bubble sort is applied to the following list of numbers, show what the list will look after each phase in the sort: 73, 21, 15, 83, 66, 7, 19, 18

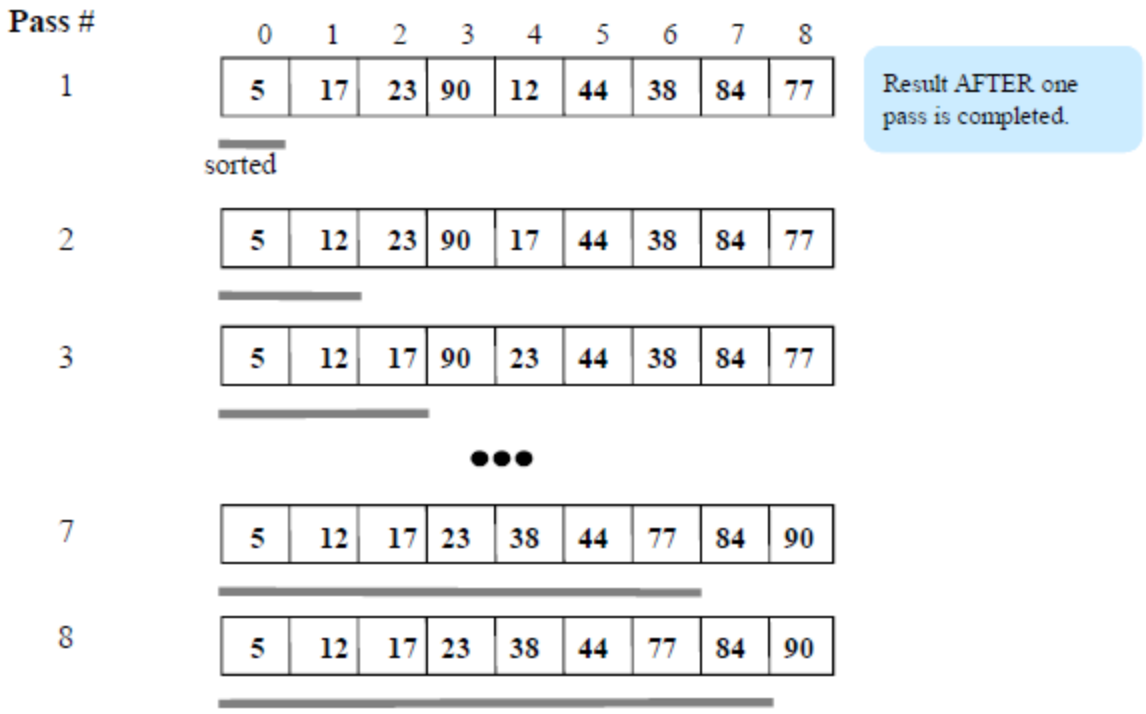
2-Selection sort:

خوارزمية الاختيار تعتمد على اختيار اصغر عنصر وابداله مع العنصر الاول (اذا كان الترتيب تصاعدي) وتكرار هذه الطريقه مع باقي العناصر , ويمكن تلخيصها بالشكل التالي:

1. Find the smallest element in the list.
2. Exchange the element in the first position and the smallest element. Now the smallest element is in the first position.
3. Repeat Step 1 and 2 with the list having one less element (i.e., the smallest element is discarded from further processing).



This is the result of one pass.



Selection sorting algorithm:

```
void selectSort(int arr[], int n)
{
       int min,temp;

       for (int i=0; i < n-1; i++)
       {
           min = i;

               for (int j=i+1; j < n; j++)
               {

                 if (arr[j] < arr[min])
                    min=j;

               }

                    if (min != i)
             {
                  temp = arr[i];
                  arr[i] = arr[min];
                  arr[min] = temp;
             }
       }
}
```

Time Complexity of Selection Sort:

The time complexity of selection sort is $O(n^2)$, for best, average, and worst case scenarios. Because of this selection sort is a very ineffecient sorting algorithm for large amounts of data, it's sometimes preffered for very small amounts of data such as the example above. The complexity is $O(n^2)$ for all cases because of the way selection sort is designed to traverse the data. The outer loops first iteration has n comparisons (where n is the number of elements in the data) the second iteration would have n-1 comparisons followed by n-2, n-3, n-4...thus resulting in $O(n^2)$ time complexity.

3-Insertion sort:

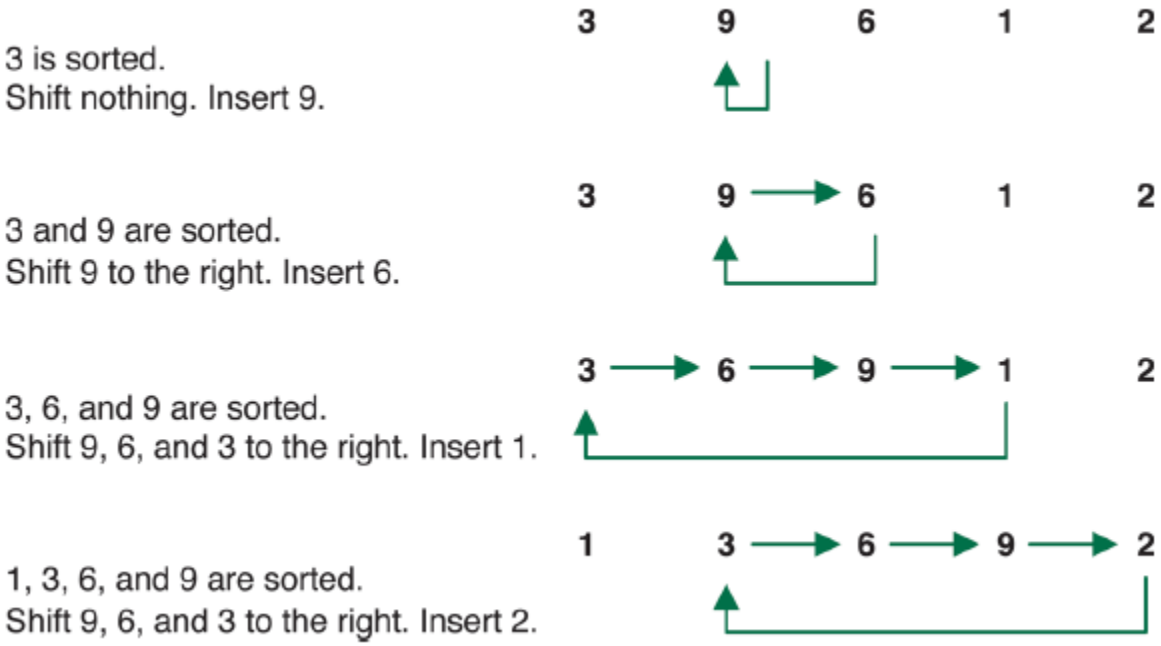
Insertion sort is a faster and more improved sorting algorithm than selection sort

How Insertion Sort Works?

- Insertion sort orders a list of values by repetitively inserting a particular value into a sorted subset of the list
- More specifically:
  - consider the first item to be a sorted sublist of length **1**
  - insert the second item into the sorted sublist, shifting the first item if needed
  - insert the third item into the sorted sublist, shifting the other items as needed
  - repeat until all values have been inserted into their proper positions

We take an unsorted array for our example.

3 9 6 1 2



Insertion sort algorithm:

```
void insertion_sort (int arr[], int length){
              int j, temp;

       for (int i = 0; i < length; i++){
              j = i;

              while (j > 0 && arr[j] < arr[j-1]){
                        temp = arr[j];
                        arr[j] = arr[j-1];
                        arr[j-1] = temp;
                        j--;
                        }
              }
}
```

4-Quick sort :

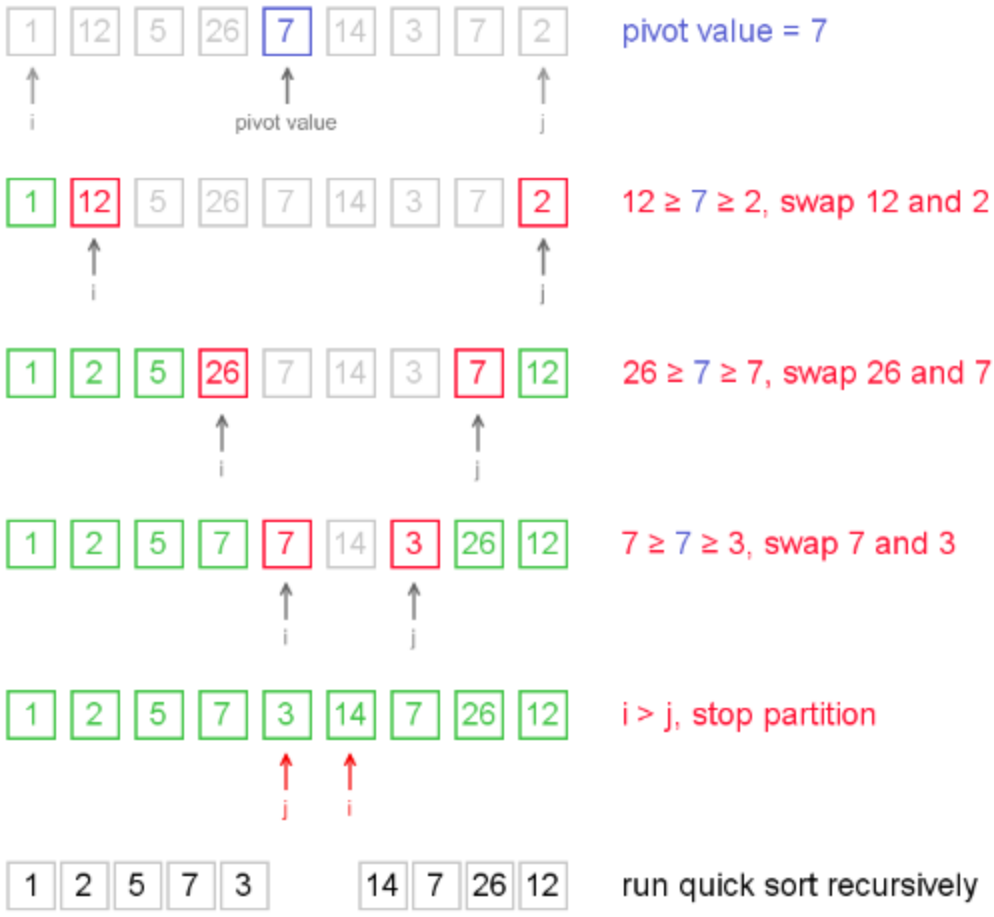
- **Quick sort** orders a list of values by partitioning the list around one element, then sorting each partition
- More specifically:
  - choose one element in the list to be the partition element
  - organize the elements so that all elements less than the partition element are to the left and all greater are to the right
  - apply the quick sort algorithm (recursively) to both partitions
- The choice of the partition element is arbitrary
- For efficiency, it would be nice if the partition element divided the list roughly in half

Partition algorithm in detail

There are two indices **i** and **j** and at the very beginning of the partition algorithm **i** points to the first element in the array and **j** points to the last one. Then algorithm moves **i** forward, until an element with value greater or equal to the pivot is found. Index **j** is moved backward, until an element with value lesser or equal to the pivot is found. If **i ≤ j** then they are swapped and i steps to the next position (**i + 1**), j steps to the previous one **(j - 1)**. Algorithm stops, when **i** becomes greater than **j**.

After partition, all values before **i-th** element are less or equal than the pivot and all values after **j-th** element are greater or equal to the pivot.

Example. Sort {1, 12, 5, 26, 7, 14, 3, 7, 2} using quicksort.



...

Quick sort algorithm:

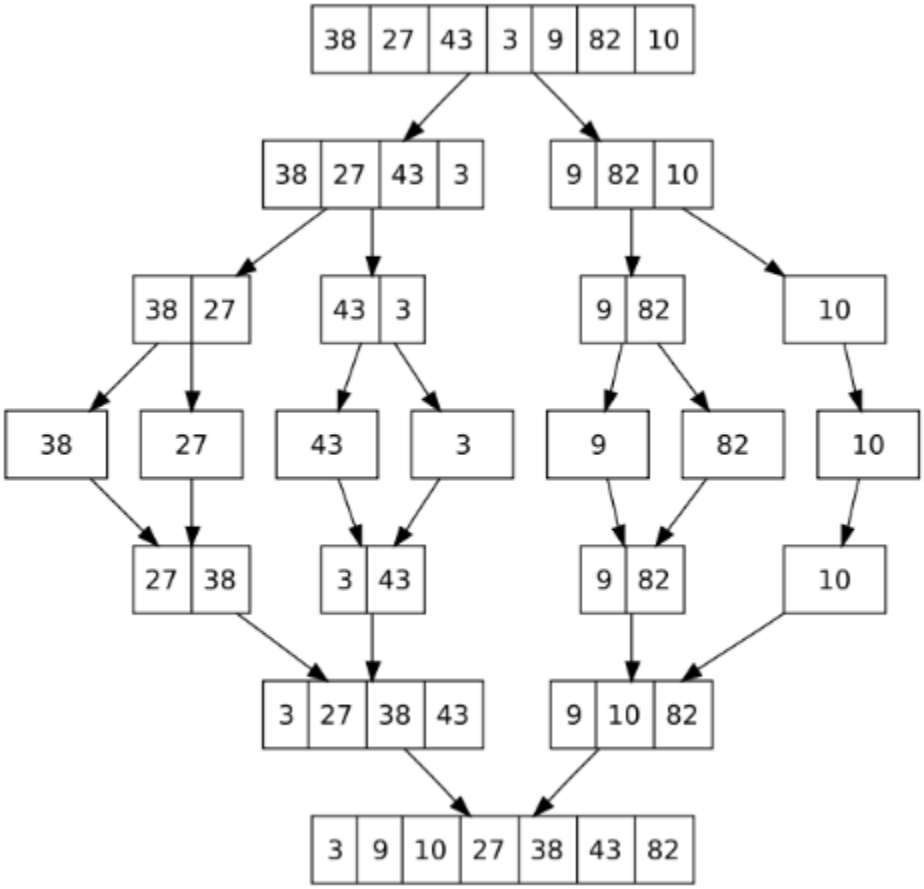
```
void quickSort(int arr[], int left, int right) {
      int i = left, j = right;
      int tmp;
      int pivot = arr[(left + right) / 2];

      /* partition */
      while (i <= j) {
            while (arr[i] < pivot)
                  i++;
            while (arr[j] > pivot)
                  j--;
            if (i <= j) {
                  tmp = arr[i];
                  arr[i] = arr[j];
                  arr[j] = tmp;
                  i++;
                  j--;
            }
      }
```

```
        /* recursion */
        if (left < j)
              quickSort(arr, left, j);
        if (i < right)
              quickSort(arr, i, right);
}
```

5-Merge sort:

- Merge sort orders a list of values by recursively dividing the list in half until each sub-list has one element, then recombining
- More specifically:
  - divide the list into two roughly equal parts
  - recursively divide each part in half, continuing until a part contains only one element
  - merge the two parts into one sorted list
  - continue to merge parts as the recursion unfolds

Example:



Merge sort algorithm:

```
#include <iostream>


void merge(int *, int *, int, int, int);
void mergesort(int *a, int *b, int low, int high)
{
   int pivot;
   if (low < high)
   {
      pivot = (low + high)/2;
      mergesort(a, b, low, pivot);
      mergesort(a, b, pivot + 1, high);
      merge(a, b, low, pivot, high);
   }
}
void merge(int *a, int *b, int low, int pivot, int high)
{
   int h, i, j, k;
   h = low;
   i = low;
```

```
   j = pivot + 1;

   while ((h <= pivot) && (j <= high)) // Traverse both halves of the array
   {
      if (a[h] <= a[j]) // if an element of left half is less than element of right half
      {
         b[i] = a[h]; // store element of left half in the temporary array
         h++; // shift the index of the array from which the element was copied to temporary
      }
      else // otherwise store the element of the right half in the temporary array
      {
         b[i] = a[j];
         j++; // shift the index of the array from which the element was copied to temporary
      }
      i++;
   }
   if (h > pivot) // If traversal of left half is done,
                  // copy remaining elements from right half to temporary
{
      for (k = j; k <= high; k++)
      {
         b[i] = a[k];
         i++;
      }
   }
   else // otherwise copy remaining elements from left half to temporary
   {
      for (k = h; k <= pivot; k++)
      {
         b[i] = a[k];
         i++;
      }
   }
   for (k = low; k <= high; k++) a[k] = b[k]; // recopy the values from temporary to original array.
}

int main()
{
   int a[] = {12, 10, 43, 23, -78, 45, 123, 56, 98, 41, 90, 24};
   int num;

   num = sizeof(a)/sizeof(int);

   int b[num]; // temporary array to be used for merging

   mergesort(a, b, 0, num-1);

   for (int i = 0; i < num; i++)
      cout << a[i] << " ";
   cout << endl;
}
```