## Basic sorting techniques:

**Bubble sort:**

Suppose, there are n elements in the array (Or list or file, whatever!) which are not in order (order means either ascending or descending). In order to make them in order, we have to sort it.

Bubble sort is the basic sorting technique.

But it is least efficient among all sorting techniques.

The basic idea of Bubble Sort technique is:

Bubble sort is to pass through the file sequentially several times. Each pass consists of comparing each element in the file with its successor (x[i] and x[i+1]) and interchanging two elements if they are not in proper order. Consider the following file:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 25 | 57 | 48 | 37 | 12 | 92 | 86 | 33 |
| **j** |  |  |  |  |  |  |  |

At first pass:

x[0] will be compared with x[1] Since X[0] is not greater than x[1], no interchange will take place

Now, j is increased by 1

x[1] with x[2] Interchange

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 25 | 48 | 57 | 37 | 12 | 92 | 86 | 33 |
|  | **j** |  |  |  |  |  |  |

Now, j is increased by 1 (i.e. j is now pointing to the 2nd index (index is started from 0)

x[2] (57) with x[3] (37) **interchange**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 25 | 48 | 37 | 57 | 12 | 92 | 86 | 33 |
|  |  | **j** |  |  |  |  |  |

Now, j is increased by 1 (at the start of the iteration of the inner loop)

x[3] with x[4] **Interchange**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 25 | 48 | 37 | 12 | 57 | 92 | 86 | 33 |
|  |  |  | j |  |  |  |  |

Again, j is increased by 1(at the start of the iteration of the inner loop)

x[4] with x[5] **no interchange**

Again, j is increased by 1(at the start of the iteration of the inner loop)

x[5] with x[6] **Interchange**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 25 | 48 | 37 | 12 | 57 | 86 | 92 | 33 |
|  |  |  |  |  | **j** |  |  |

Again, j is increased by 1(at the start of the iteration of the inner loop)

x[6] with x[7] **interchange**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 25 | 48 | 37 | 12 | 57 | 86 | 33 | 92 |
|  |  |  |  |  |  | **j** |  |

Now, look, we don’t even increase the j…because, the last element is already properly placed.

Now, as you can see, after the first pass, the largest element is properly placed.

**Second pass:**

x[0] is compared with x[1] **no interchange**

x[1] is compared with x[2] **interchange**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 25 | 37 | 48 | 12 | 57 | 86 | 33 | 92 |
|  | **j** |  |  |  |  |  |  |

Again, j is increased by 1(at the start of the iteration of the inner loop)

x[2] is compared with x[3] **interchange**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 25 | 37 | 12 | 48 | 57 | 86 | 33 | 92 |
|  |  | **j** |  |  |  |  |  |

Again, j is increased by 1(at the start of the iteration of the inner loop)

x[3] is compared with x[4] **no interchange**

Again, j is increased by 1(at the start of the iteration of the inner loop)

x[4] is compared with x[5] **no interchange**

Again, j is increased by 1(at the start of the iteration of the inner loop)

x[5] is compared with the x[6] **interchange**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 25 | 37 | 12 | 48 | 57 | 33 | 86 | 92 |
|  |  |  |  |  | **j** |  |  |

No further iteration of j is needed for this pass value.

Now, as you can see, after first pass 1st largest element gets properly placed.

after second pass, 2nd largest element gets properly placed.

after third pass, 3rd largest element gets properly placed.

after ith pass, ith largest element gets properly placed.

So, after ith pass, we need to sort remaining n-i elements…(so, comparison will take place from 0 to (n-i-1)th index…so, j’s value can be 0 to (n-i-1)-1 because, the element of jth index is compared to (j+1)th index.

Another thing to be noticed, that after n-2 passes, only 2 elements remain to sort.

If one of them can be properly placed, then the rest of element will also be properly placed. So, we don’t need total n passes to sort all the elements we only need (n-1) passes, because after n-1 passes, n-1 elements is in their proper position. So, the left element is automatically proper placed.

for(pass=0;pass<n-1;pass++)

{

for(j=0;j<n-1-pass;j++)

{

if(x[j]>x[j+1])

{

temp=x[j];

x[j]=x[j+1];

x[j+1]=temp;

}

}

}

Whole program of the corresponding algorithm is:

**import java.io.\*;**

**class Bubblesort1**

**{**

**static void Bubble(int arr[],int n)**

**{**

**int pass,j,temp;**

**for(pass=0;pass<n-1;pass++)**

**{**

**for(j=0;j<n-pass-1;j++)**

**{**

**if(arr[j]>arr[j+1])**

**{**

**temp=arr[j];**

**arr[j]=arr[j+1];**

**arr[j+1]=temp;**

**}**

**}**

**}**

**}**

**//then the driver function or main**

**public static void main(String args[])throws IOException**

**{**

**System.out.println("\u000c");**

**int n;**

**BufferedReader z=new BufferedReader(new InputStreamReader(System.in));**

**System.out.print("Enter the no. of elements of the array:");**

**n=Integer.parseInt(z.readLine());**

**System.out.println();**

**int arr[]=new int[n];**

**//read operation**

**for(int i=0;i<n;i++)**

**{**

**arr[i]=Integer.parseInt(z.readLine());**

**}**

**Bubble(arr,n);**

**//printing elements**

**System.out.print("After sorting the elements are:");**

**for(int i=0;i<n;i++)**

**{**

**System.out.print(arr[i]+" ");**

**}**

**System.out.println();**

**}**

**}**

However, learners may find it difficult to understand this particular algorithm of bubble sort. So, there’s an easy algorithm of bubble sort.

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

{

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

{

if(x[i]>x[j])

{

temp=x[i];

x[i]=x[j];

x[j]=temp;

}

}

}

This is also the same algorithm, written in a diff. manner. Unlike the first algorithm (where, after ith iteration of the outer loop, ith largest element is properly placed), here, after ith iteration of the outer loop, the ith smallest element of the array, gets properly placed.

It will work like:

**First iteration of the outer loop:**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 25 | 57 | 48 | 37 | 12 | 92 | 86 | 33 |
| **i** | **j** |  |  |  |  |  |  |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 25 | 57 | 48 | 37 | 12 | 92 | 86 | 33 |
| **i** |  | **j** |  |  |  |  |  |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 25 | 57 | 48 | 37 | 12 | 92 | 86 | 33 |
| **i** |  |  | **j** |  |  |  |  |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 25 | 57 | 48 | 37 | 12 | 92 | 86 | 33 |
| **i** |  |  |  | **j** |  |  |  |

Now, interchange

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 12 | 57 | 48 | 37 | 25 | 92 | 86 | 33 |
| **i** |  |  |  | **j** |  |  |  |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 12 | 57 | 48 | 37 | 25 | 92 | 86 | 33 |
| **i** |  |  |  |  | **j** |  |  |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 12 | 57 | 48 | 37 | 25 | 92 | 86 | 33 |
| **i** |  |  |  |  |  | **j** |  |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 12 | 57 | 48 | 37 | 25 | 92 | 86 | 33 |
| **i** |  |  |  |  |  |  | **j** |

At second pass (second iteration of the outer loop)

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 12 | 57 | 48 | 37 | 25 | 92 | 86 | 33 |
|  | **i** | **j** |  |  |  |  |  |

Interchange

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 12 | 48 | 57 | 37 | 25 | 92 | 86 | 33 |
|  | **i** | **j** |  |  |  |  |  |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 12 | 48 | 57 | 37 | 25 | 92 | 86 | 33 |
|  | **i** |  | **j** |  |  |  |  |

interchange

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 12 | 37 | 57 | 48 | 25 | 92 | 86 | 33 |
|  | **i** |  | **j** |  |  |  |  |

then,

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 12 | 37 | 57 | 48 | 25 | 92 | 86 | 33 |
|  | **i** |  |  | **j** |  |  |  |

interchange

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 12 | 25 | 57 | 48 | 37 | 92 | 86 | 33 |
|  | **i** |  |  | **j** |  |  |  |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 12 | 25 | 57 | 48 | 37 | 92 | 86 | 33 |
|  | **i** |  |  |  | **j** |  |  |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 12 | 25 | 57 | 48 | 37 | 92 | 86 | 33 |
|  | **i** |  |  |  |  | **j** |  |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 12 | 25 | 57 | 48 | 37 | 92 | 86 | 33 |
|  | **i** |  |  |  |  |  | **j** |

Now, third iteration, clearly after second iteration of the outer loop, 2nd smallest element gets proper placed.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 12 | 25 | 57 | 48 | 37 | 92 | 86 | 33 |
|  |  | **i** | **j** |  |  |  |  |

And so on…

(Now, what can be said about the efficiency of the bubble sort? In the case of these two algorithms,

for placing the 1st smallest/largest element in the order, we need (n-1) iterations of the inner loop. For placing the 2nd smallest/largest element in the order (in its proper place), we need (n-2) iterations of the inner loop. So, if totally k iterations are needed (k is always (n-1). Because, if we can place (n-1) largest/smallest elements according to order, the left element is automatically properly placed. However, let’s take it as k) of the outer loop, to sort the file, then the total no. of iterations of the inner loop is (n-1)+(n-2)+(n-3)+…………+(n-k)=(nk)-(1+2+3+…k)

=nk-(k(k+1))/2

=(2kn-k2-k)/2 )Not needed

The program corresponding to this algorithm is:  
import java.io.\*;

**class Bubblesort2**

**{**

**static void Bubble(int arr[],int n)**

**{**

**int i,j,temp;**

**for(i=0;i<n-1;i++)**

**{**

**for(j=i+1;j<n;j++)**

**{**

**if(arr[i]>arr[j])**

**{**

**temp=arr[i];**

**arr[i]=arr[j];**

**arr[j]=temp;**

**}**

**}**

**}**

**}**

**public static void main(String args[])throws IOException**

**{**

**System.out.println("\u000c");**

**int n;**

**BufferedReader z=new BufferedReader(new InputStreamReader(System.in));**

**System.out.print("Enter the no. of elements of the array:");**

**n=Integer.parseInt(z.readLine());**

**System.out.println();**

**int arr[]=new int[n];**

**//read operation**

**for(int i=0;i<n;i++)**

**{**

**arr[i]=Integer.parseInt(z.readLine());**

**}**

**Bubble(arr,n);**

**//printing elements**

**System.out.print("After sorting the elements are:");**

**for(int i=0;i<n;i++)**

**{**

**System.out.print(arr[i]+" ");**

**}**

**System.out.println();**

**}**

**}**

Insertion sort

Insertion sort is a simple sorting algorithm that builds the final sorted array (or list) one item at a time. It is much less efficient on large lists than more advanced algorithms such as quicksort, heapsort, or merge sort. However, insertion sort provides several advantages:

static void insertion\_sort(int Arr[],int n)

{

int p,j;

int temp;

for(p=1;p<n;p++)

{

temp=Arr[p];

for(j=p;j>0&&Arr[j-1]>temp;j--)

{

Arr[j]=Arr[j-1];

}

Arr[j]=temp;

}

}

**Unsorted data**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 25 | 57 | 48 | 37 | 12 | 86 | 92 | 33 |

Start of the outer loop: (starting of the first pass)

Start of the inner loop:

temp is set as 57

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 25 | 57 | 48 | 37 | 12 | 86 | 92 | 33 |
|  | **p,j** |  |  |  |  |  |  |

Now, A[j-1] is not greater than temp. So, no change will take place

p increases (starting of 2nd pass)

At the starting of the 2nd pass: temp is set as 48

starting of inner loop:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 25 | 57 | 48 | 37 | 12 | 86 | 92 | 33 |
|  |  | **p,j** |  |  |  |  |  |

Now, A[j-1] is greater than temp. So, A[j]=A[j-1] and after4 this j will be decreased by 1

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 25 | 57 | 57 | 37 | 12 | 86 | 92 | 33 |
|  | **j** | **p** |  |  |  |  |  |

Now, A[j-1] (25) is not greater than A[p] (57)

Iteration of inner loop ends.

Now, A[j]=temp.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 25 | 48 | 57 | 37 | 12 | 86 | 92 | 33 |
|  | **j** | **p** |  |  |  |  |  |

Now, second pass ends.

At the beginning of third pass:

p=3 (index)

temp is set as 37

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 25 | 48 | 57 | 37 | 12 | 86 | 92 | 33 |
|  |  |  | **p, j** |  |  |  |  |

A[j-1]>temp ? Yes

So, A[j]=A[j-1]

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 25 | 48 | 57 | 57 | 12 | 86 | 92 | 33 |
|  |  | **j** | **p** |  |  |  |  |

Now, A[j-1] is greater than temp

j decreases

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 25 | 48 | 48 | 57 | 12 | 86 | 92 | 33 |
|  | **j** |  | **p** |  |  |  |  |

Now, A[j-1] is not greater than temp. Now, loop breaks

A[j]=37

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 25 | 37 | 48 | 57 | 12 | 86 | 92 | 33 |
|  | **j** |  | **p** |  |  |  |  |

Now, third pass ends

Now, At the beginning of 4th pass

p is set as 4 (index)

and temp is set as 12

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 25 | 37 | 48 | 57 | 12 | 86 | 92 | 33 |
|  |  |  |  | **p,j** |  |  |  |

Now,

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 25 | 37 | 48 | 57 | 57 | 86 | 92 | 33 |
|  |  |  | **j** | **p** |  |  |  |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 25 | 37 | 48 | 48 | 57 | 86 | 92 | 33 |
|  |  | **j** |  | **p** |  |  |  |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 25 | 37 | 37 | 48 | 57 | 86 | 92 | 33 |
|  | **j** |  |  | **p** |  |  |  |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 25 | 25 | 37 | 48 | 57 | 86 | 92 | 33 |
| **j** |  |  |  | **p** |  |  |  |

inner loop breaks

Now, a[j] (j’s value is 0. that’s why the inner loop breaks) gets the value of temp.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 12 | 25 | 37 | 48 | 57 | 86 | 92 | 33 |
| **j** |  |  |  | **p** |  |  |  |

Now,5th pass starts and p=5 temp=86

After, 5th pass, same thing remains

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 12 | 25 | 37 | 48 | 57 | 86 | 92 | 33 |
| **j** |  |  |  |  | **p** |  |  |

This is the situation after 5th pass

Now, at the beginning of the next pass:

p=6 and temp=92

same situation remains after 6th pass

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 12 | 25 | 37 | 48 | 57 | 86 | 92 | 33 |
| **j** |  |  |  |  |  | **p** |  |

and the beginning of the new pass (7th pass) p is set as 7 and temp=33

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 12 | 25 | 37 | 48 | 57 | 86 | 92 | 33 |
|  |  |  |  |  |  |  | **p, j** |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 12 | 25 | 37 | 48 | 57 | 86 | 92 | 92 |
|  |  |  |  |  |  | **j** | **p** |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 12 | 25 | 37 | 48 | 57 | 86 | 86 | 92 |
|  |  |  |  |  | **j** |  | **p** |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 12 | 25 | 37 | 48 | 57 | 57 | 86 | 92 |
|  |  |  |  | **j** |  |  | **p** |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 12 | 25 | 37 | 48 | 48 | 57 | 86 | 92 |
|  |  |  | **j** |  |  |  | **p** |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 12 | 25 | 37 | 37 | 48 | 57 | 86 | 92 |
|  |  | **j** |  |  |  |  | **p** |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 12 | 25 | 37 | 37 | 48 | 57 | 86 | 92 |
|  |  | **j** |  |  |  |  | **p** |

loop breaks (because, A[j-1] is not greater than temp)

A[j]=33

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 12 | 25 | 33 | 37 | 48 | 57 | 86 | 92 |
|  |  | **j** |  |  |  |  | **p** |

And then outer loop ends…

Now, I just drew the step by step thing. How does it really work? Why is this sorting technique called insertion sort ?

P starts from second element. The process continues by inserting the element denoted by temp (of pth index) in its relative position in the unsorted array.

25 is the first element. Now, the new element is 57. We have to insert it in its relative position.

25 57

Now, we have to insert 48 in its relative position in the array 25 57

25 48 57 (48 should be inserted in the 1st index)

Then 37 needs to be inserted in its relative position.

25 37 48 57

12 25 37 48 57 (12 is inserted in its relative position)

12 25 37 48 57 86 (86 is inserted in its relative position)

12 25 37 48 57 86 92 (92 is inserted in its relative position)

12 25 33 37 48 57 86 92 (33 is inserted in its proper position)

And that’s the end of insertion\_sort.

Insertion sort iterates, consuming one input element each repetition, and growing a sorted output list. Each iteration, insertion sort removes one element from the input data, finds the location it belongs within the sorted list, and inserts it there. It repeats until no input elements remain.

Sorting is typically done in-place, by iterating up the array, growing the sorted list behind it. At each array-position, it checks the value there against the largest value in the sorted list (which happens to be next to it, in the previous array-position checked). If larger, it leaves the element in place and moves to the next. If smaller, it finds the correct position within the sorted list, shifts all the larger values up to make a space, and inserts into that correct position.  
Now, the program of insertion sort:  
**import java.io.\*;**

**class Insertionsort**

**{**

**static void insertion\_sort(int Arr[],int n)**

**{**

**int p,j;**

**int temp;**

**for(p=1;p<n;p++)**

**{**

**temp=Arr[p];**

**for(j=p;j>0&&Arr[j-1]>temp;j--)**

**{**

**Arr[j]=Arr[j-1];**

**}**

**Arr[j]=temp;**

**}**

**}**

**public static void main(String args[])throws IOException**

**{**

**System.out.println("\u000c");**

**int n;**

**BufferedReader z=new BufferedReader(new InputStreamReader(System.in));**

**System.out.print("Enter the no. of elements of the array:");**

**n=Integer.parseInt(z.readLine());**

**System.out.println();**

**int arr[]=new int[n];**

**//read operation**

**for(int i=0;i<n;i++)**

**{**

**arr[i]=Integer.parseInt(z.readLine());**

**}**

**insertion\_sort(arr,n);**

**//printfing elements**

**System.out.print("After sorting the elements are:");**

**for(int i=0;i<n;i++)**

**{**

**System.out.print(arr[i]+" ");**

**}**

**System.out.println();**

**}**

**}**

**Selection Sort**The algorithm divides the input list into two parts: the sub list of items already sorted, which is built up from left to right at the front (left) of the list, and the sub list of items remaining to be sorted that occupy the rest of the list. Initially, the sorted sub list is empty and the unsorted sub list is the entire input list. The algorithm proceeds by finding the smallest (or largest, depending on sorting order) element in the unsorted sub list, exchanging it with the leftmost unsorted element (putting it in sorted order), and moving the sub list boundaries one element to the right.   
Selection sort is easy to implement. But, it is inefficient for large list of unsorted numbers.  
  
**The algorithm of selection sort is:**

**static void selectionsort(int x[],int n)**

**{**

**int i,j,index,largest;**

**for(i=n-1;i>=1;i--)**

**{**

**index=0;**

**largest=x[0];**

**for(j=1;j<=i;j++)**

**{**

**if(x[j]>largest)**

**{**

**largest=x[j];**

**index=j;**

**}**

**}**

**x[index]=x[i];**

**x[i]=largest;**

**}  
How does this algorithm work?**

**Initial unsorted array:**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **25** | **57** | **48** | **37** | **12** | **86** | **32** | **96** |
|  |  |  |  |  |  |  |  |

**Now,** before first pass of the inner loop: 25 is set the largest element

then, we choose the largest element from 1 to (n-1) (index) and make it placed in the proper position i.e. in the (n-1) index

After first pass

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 25 | 57 | 48 | 37 | 12 | 86 | 32 | 96 |

The largest element is in its proper position

Then, again we find the 2nd largest element from 0 to (n-2) After 2nd pass:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 25 | 57 | 48 | 37 | 12 | 32 | 86 | 96 |

After 3rd pass:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 25 | 32 | 48 | 37 | 12 | 57 | 86 | 96 |

After 4th pass:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 25 | 32 | 12 | 37 | 48 | 57 | 86 | 96 |

After 5th pass:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 25 | 32 | 12 | 37 | 48 | 57 | 86 | 96 |

After 6th pass:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 25 | 12 | 32 | 37 | 48 | 57 | 86 | 96 |

After 7th pass:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 12 | 25 | 32 | 37 | 48 | 57 | 86 | 96 |

**Total no. of comparisons:**

**(n-1)+(n-2)+(n-3)+….+3+2+1**

Understanding of this thing:

In case of finding the largest element in the first case, (n-1) comparisons are needed (x[0] is compared with x[1],x[2]…..,x[n-1]) and after this, the largest element is placed in its proper place (i.e. in the last index)

Now, in the second case total (n-2) comparisons are needed (x[0] is compared with x[1].x[2],x[3]……x[n-2]) (it is not compared with the last index. Because, we are trying to find the second largest element. And largest element is already placed at index (n-1))

And so on…

**The program of the selectionsort is:**

**import java.io.\*;**

**class Selectionsort**

**{**

**static void selectionsort(int x[],int n)**

**{**

**int i,j,index,largest;**

**for(i=n-1;i>=1;i--)**

**{**

**index=0;**

**largest=x[0];**

**for(j=1;j<=i;j++)**

**{**

**if(x[j]>largest)**

**{**

**largest=x[j];**

**index=j;**

**}**

**}**

**x[index]=x[i];**

**x[i]=largest;**

**}**

**}**

**public static void main(String args[])throws IOException**

**{**

**System.out.println("\u000c");**

**int n;**

**BufferedReader z=new BufferedReader(new InputStreamReader(System.in));**

**System.out.print("Enter the no. of elements of the array:");**

**n=Integer.parseInt(z.readLine());**

**System.out.println();**

**int arr[]=new int[n];**

**//read operation**

**for(int i=0;i<n;i++)**

**{**

**arr[i]=Integer.parseInt(z.readLine());**

**}**

**selectionsort(arr,n);**

**//printfing elements**

**System.out.print("After sorting the elements are:");**

**for(int i=0;i<n;i++)**

**{**

**System.out.print(arr[i]+" ");**

**}**

**System.out.println();**

**}**

**}**

**Quicksort (probably not needed in your syllabus)**

**Algorithm** quicksort (lb,ub: array indices )

//i.e. lb (lower bound) and ub (upper bound) are the array indices

if(lb<ub)

then j<-partition (lb,ub)

//partition algorithm is to place the pivot element

quicksort(lb,j-1);

quicksort(j+1,ub);

return;

Before, writing the partition algorithm, let’s learn how the partition algorithm works.

First, set ‘a’ as x[lb] (as pivot element)

(down will be initially set to the lb index, and it will search a bigger element than the ‘a’. When, it will find so, down index will stop there. Up index will be initially set to ub index. And it will search an element smaller than the a)

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| down-> |  |  |  |  |  |  | <-Up |
| 25 | 57 | 48 | 37 | 12 | 86 | 92 | 33 |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | down |  |  |  |  |  | <-Up |
| 25 | 57 | 48 | 37 | 12 | 86 | 92 | 33 |

Down index will not move now. since it finds the first element which is greater than the ‘a’

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | down |  |  |  |  | <-up |  |
| 25 | 57 | 48 | 37 | 12 | 86 | 92 | 33 |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | down |  |  |  | <-up |  |  |
| 25 | 57 | 48 | 37 | 12 | 86 | 92 | 33 |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | down |  |  | up |  |  |  |
| 25 | 57 | 48 | 37 | 12 | 86 | 92 | 33 |

Now, up index will stop moving. Since, it found the first element which is smaller than ‘a’

Now, swapping between x[up] and x[down] will take place

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | down-> |  |  | <-up |  |  |  |
| 25 | 12 | 48 | 37 | 57 | 86 | 92 | 33 |

Then,

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  |  | down |  | up |  |  |  |
| 25 | 12 | 48 | 37 | 57 | 86 | 92 | 33 |

Again, down found the element

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  |  | down | up |  |  |  |  |
| 25 | 12 | 48 | 37 | 57 | 86 | 92 | 33 |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  |  | down,<-up |  |  |  |  |  |
| 25 | 12 | 48 | 37 | 57 | 86 | 92 | 33 |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | up | down |  |  |  |  |  |
| 25 | 12 | 48 | 37 | 57 | 86 | 92 | 33 |

j<-up (because up becomes less than down)

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | up | down |  |  |  |  |  |
| 12 | 25 | 48 | 37 | 57 | 86 | 92 | 33 |

**Steps of partition**

Step1: Set a<-x[lb]

set up<-ub (upper bound)

set down<-lb (lower bound)

Step 2: Repeatedly increase ptr. down until x[down]>a

Step 3: Repeatedly decrease ptr. up until x[up]<=a

Step4:-if up>down

then exchange x[up] and x[down] and go to step 2

else if up<=down

then exchange x[up] with ‘a’

step 5:j<-up

**algorithm partition (lb,ub) //two indices again**

**{**

**a<-x[lb];//choosing x[lb] as the pivot element**

**while(up>down)**

**{**

**while(x[down]<=a) and (down<ub)**

**down<-down+1;**

**while(x[up]>a)**

**up<-up-1;**

**if(up>down)**

**{**

**temp<-x[down]**

**}**

**x[down]<=x[up]**

**x[up]<-temp;**

**}**

**}/\*end of while\*/**

**//exchange x[up] with x[lb]( pivot element]**

**//why we are not exchanging it with a**

**//because then the change with not take place in the array**

**temp<-x[lb];**

**x[lb]<-x[up];**

**x[up]<-temp;**

**j<-up;**

**return j;**

Now, understanding of a very crucial position of the algorithm:

Inside while(up>down)

we write this:

while(x[down]<=a) and (down<ub)

down<-down+1;

while(x[up]>a)

up<-up-1;

why we need such types of conditions:

Like: we do we need the additional condition (down<ub) in the first while

Because, down’s motive is to find such a x[down] which is greater than a (the pivot element) which is selected as the first element of the currently handled array

Now, it could be possible that already we enter a (decreasing sorted array i.e. the first element of the array is the greatest one…like in this case, if we enter 92 86 57 48 37 33 25 12 as our input array for sorting. Then, we can’t find any element greater than the pivot element. So, if we does not apply this additional condition (down<ub) then the index/ptr. down goes out of bound. in this case, the job of up is pretty simple (to find an element lesser or equal to than the pivot element which is the last element itself (12) in this case. Now, why don’t we need such additional condition for restricting ‘up’ to go out of bound. Because, ‘up’ can never go out of bound. Since, it’s job to find an element less than or equal to the pivot element (now, even if the array is sorted in the increasing order (which could be the critical condition for up), ‘up’ index does not go out of bound.

**(Time complexity of Quick sort: (Not needed for your course)**

Let, the no. of elements in an array: n=2m, m=log2n

n+2.n/2+4.n/4+….+n.n/n

=n+n+n+….n (m times)

=mn

=n log2n (but this is average case time complexity)

**But as we know, the efficiency of a sorting is decided based upon worst case time complexity.**

**Worst case time complexity of the quicksort:**

The worst case happens when the array is completely sorted and the condition that the proper position

of the pivot element turns out to be middle does not hold. (i.e. pivot element’s proper position is not in the middle of the array)

if a=x[lb] is in correct position of the original array then it is split into two sub arrays of sizes zero and n-1 if the process continues a total of (n-1) sub arrays are sorted, the first of size n, second of size (n-1), third of size (n-2) and so on. Therefore the total number of comparisons to sort the entire (already sorted) array is (n-1)+(n-2)+……+2+1

=(n\*(n-1))/2

\approx\,n2 (I did not get the actual symbol needed) )

**The program for the quicksort is:**

**import java.io.\*;**

**class Quicksort**

**{**

**public static void main(String args[])throws IOException**

**{**

**System.out.println("\u000c");**

**int n;**

**BufferedReader z=new BufferedReader(new InputStreamReader(System.in));**

**System.out.print("Enter the no. of elements of the array:");**

**n=Integer.parseInt(z.readLine());**

**System.out.println();**

**int arr[]=new int[n];**

**//read operation**

**for(int i=0;i<n;i++)**

**{**

**arr[i]=Integer.parseInt(z.readLine());**

**}**

**quicksort(arr,0,n-1);**

**//printfing elements**

**System.out.print("After sorting the elements are:");**

**for(int i=0;i<n;i++)**

**{**

**System.out.print(arr[i]+" ");**

**}**

**System.out.println();**

**}**

**static void quicksort(int arr[],int lb,int ub)**

**{**

**int actual\_pos;**

**if(lb<ub)**

**{**

**actual\_pos=partition(arr,lb,ub);**

**quicksort(arr,lb,actual\_pos-1);**

**quicksort(arr,actual\_pos+1,ub);**

**}**

**}**

**static int partition(int arr[],int lb,int ub)**

**{**

**int up=ub,down=lb,temp,pivot=arr[lb],pivot\_pos;**

**while(up>down)**

**{**

**while((arr[down]<=pivot)&&(down<ub))**

**down=down+1;**

**while(arr[up]>pivot)**

**up=up-1;**

**if(up>down)**

**{**

**temp=arr[down];**

**arr[down]=arr[up];**

**arr[up]=temp;**

**}**

**}**

**//exchange arr[lb] and arr[up]**

**temp=arr[lb];**

**arr[lb]=arr[up];**

**arr[up]=temp;**

**pivot\_pos=up;**

**return pivot\_pos;**

**}**

**}**