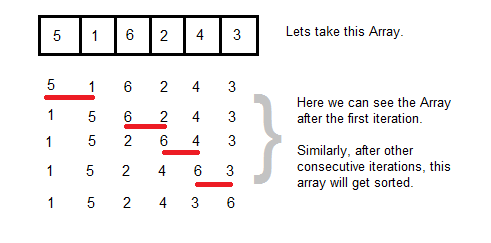
**Types of Sort:**

**Bubble Sorting**

**Bubble Sort** is an algorithm which is used to sort **N** elements that are given in a memory for eg: an Array with **N** number of elements. Bubble Sort compares all the element one by one and sort them based on their values.

It is called Bubble sort, because with each iteration the smaller element in the list bubbles up towards the first place, just like a water bubble rises up to the water surface.

Sorting takes place by stepping through all the data items one-by-one in pairs and comparing adjacent data items and swapping each pair that is out of order.



**Sorting using Bubble Sort Algorithm**

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;

}

}

}

//now you can print the sorted array after this

Above is the algorithm, to sort an array using Bubble Sort. Although the above logic will sort and unsorted array, still the above algorithm isn't efficient and can be enhanced further. Because as per the above logic, the for loop will keep going 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. If no swapping is taking place that means the array is sorted and wew can jump out of the for loop.

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++)

{

int flag = 0; //taking a flag variable

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

}

}

if(!flag) //breaking out of for loop if no swapping takes place

{

break;

}

}

In the above code, if in a complete single cycle of j iteration(inner for loop), no swapping takes place, and flag remains 0, then we will break out of the for loops, because the array has already been sorted.

**Complexity Analysis of Bubble Sorting**

In Bubble Sort, n-1 comparisons will be done in 1st pass, n-2 in 2nd pass, n-3 in 3rd pass and so on. So the total number of comparisons will be

(n-1)+(n-2)+(n-3)+.....+3+2+1

Sum = n(n-1)/2

i.e O(n2)

Hence the complexity of Bubble Sort is **O(n2)**.

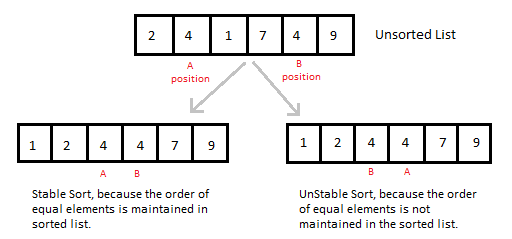
The main advantage of Bubble Sort is the simplicity of the algorithm.Space complexity for Bubble Sort is **O(1)**, because only single additional memory space is required for **temp** variable

**Best-case** Time Complexity will be **O(n)**, it is when the list is already sorted.

**Insertion Sorting**

It is a simple Sorting algorithm which sorts the array by shifting elements one by one. Following are some of the important characteristics of Insertion Sort.

1. It has one of the simplest implementation
2. It is efficient for smaller data sets, but very inefficient for larger lists.
3. Insertion Sort is adaptive, that means it reduces its total number of steps if given a partially sorted list, hence it increases its efficiency.
4. It is better than Selection Sort and Bubble Sort algorithms.
5. Its space complexity is less, like Bubble Sorting, inerstion sort also requires a single additional memory space.
6. It is **Stable**, as it does not change the relative order of elements with equal keys



**How Insertion Sorting Works**



**Sorting using Insertion Sort Algorithm**

int a[6] = {5, 1, 6, 2, 4, 3};

int i, j, key;

for(i=1; i<6; i++)

{

key = a[i];

j = i-1;

while(j>=0 && key < a[j])

{

a[j+1] = a[j];

j--;

}

a[j+1] = key;

}

Now lets, understand the above simple insertion sort algorithm. We took an array with 6 integers. We took a variable **key**, in which we put each element of the array, in each pass, starting from the second element, that is **a[1]**.

Then using the while loop, we iterate, until **j** becomes equal to zero or we find an element which is greater than **key**, and then we insert the key at that position.

In the above array, first we pick 1 as key, we compare it with 5(element before 1), 1 is smaller than 5, we shift 1 before 5. Then we pick 6, and compare it with 5 and 1, no shifting this time. Then 2 becomes the key and is compared with, 6 and 5, and then 2 is placed after 1. And this goes on, until complete array gets sorted.

**Complexity Analysis of Insertion Sorting**

**Worst Case Time Complexity :** O(n2)

**Best Case Time Complexity :** O(n)

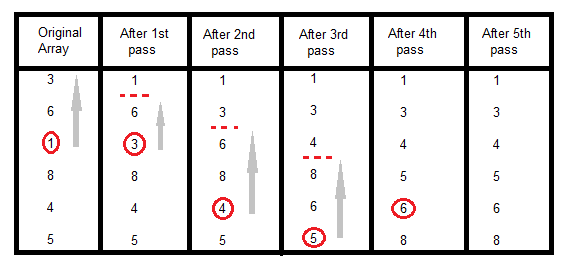
**Average Time Complexity :** O(n2)

**Space Complexity :** O(1)

**Selection Sorting**

Selection sorting is conceptually the most simplest sorting algorithm. This algorithm first finds the smallest element in the array and exchanges it with the element in the first position, then find the second smallest element and exchange it with the element in the second position, and continues in this way until the entire array is sorted.

**How Selection Sorting Works**



In the first pass, the smallest element found is 1, so it is placed at the first position, then leaving first element, smallest element is searched from the rest of the elements, 3 is the smallest, so it is then placed at the second position. Then we leave 1 nad 3, from the rest of the elements, we search for the smallest and put it at third position and keep doing this, until array is sorted.

**Sorting using Selection Sort Algorithm**

void selectionSort(int a[], int size)

{

int i, j, min, temp;

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

{

min = i; //setting min as i

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

{

if(a[j] < a[min]) //if element at j is less than element at min position

{

min = j; //then set min as j

}

}

temp = a[i];

a[i] = a[min];

a[min] = temp;

}

}

**Complexity Analysis of Selection Sorting**

**Worst Case Time Complexity :** O(n2)

**Best Case Time Complexity :** O(n2)

**Average Time Complexity :** O(n2)

**Space Complexity :** O(1)

**Quick Sort Algorithm**

Quick Sort, as the name suggests, sorts any list very quickly. Quick sort is not stable search, but it is very fast and requires very less aditional space. It is based on the rule of **Divide and Conquer**(also called *partition-exchange sort*). This algorithm divides the list into three main parts :

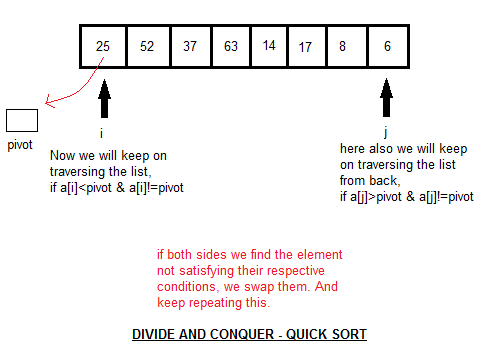
1. Elements less than the Pivot element
2. Pivot element
3. Elements greater than the pivot element

In the list of elements, mentioned in below example, we have taken **25** as pivot. So after the first pass, the list will be changed like this.

6 8 17 14 **25** 63 37 52

Hnece after the first pass, pivot will be set at its position, with all the elements smaller to it on its left and all the elements larger than it on the right. Now 6 8 17 14 and 63 37 52 are considered as two separate lists, and same logic is applied on them, and we keep doing this until the complete list is sorted.

**How Quick Sorting Works**



**Sorting using Quick Sort Algorithm**

/\* a[] is the array, p is starting index, that is 0,

and r is the last index of array. \*/

void **quicksort**(int a[], int p, int r)

{

if(p < r)

{

int q;

q = partition(a, p, r);

quicksort(a, p, q);

quicksort(a, q+1, r);

}

}

int **partition**(int a[], int p, int r)

{

int i, j, pivot, temp;

pivot = a[p];

i = p;

j = r;

while(1)

{

while(a[i] < pivot && a[i] != pivot)

i++;

while(a[j] > pivot && a[j] != pivot)

j--;

if(i < j)

{

temp = a[i];

a[i] = a[j];

a[j] = temp;

}

else

{

return j;

}

}

}

**Complexity Analysis of Quick Sort**

**Worst Case Time Complexity :** O(n2)

**Best Case Time Complexity :** O(n log n)

**Average Time Complexity :** O(n log n)

**Space Complexity :** O(n log n)

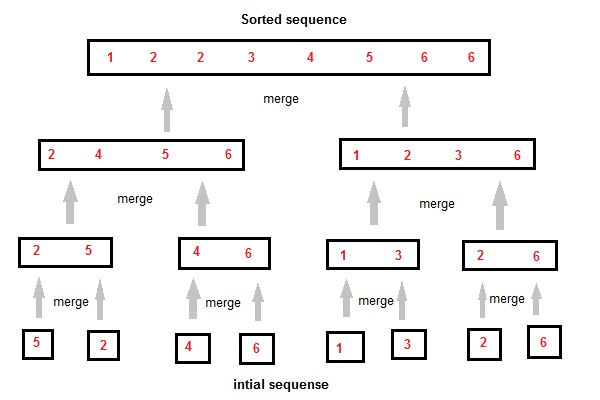
* Space required by quick sort is very less, only O(n log n) additional space is required.
* Quick sort is not a stable sorting technique, so it might change the occurence of two similar elements in the list while sorting.

**Merge Sort Algorithm**

Merge Sort follows the rule of **Divide and Conquer**. But it doesn't divides the list into two halves. In merge sort the unsorted list is divided into N sublists, each having one element, because a list of one element is considered sorted. Then, it repeatedly merge these sublists, to produce new sorted sublists, and at lasts one sorted list is produced.

Merge Sort is quite fast, and has a time complexity of **O(n log n)**. It is also a stable sort, which means the "equal" elements are ordered in the same order in the sorted list.

**How Merge Sort Works**



Like we can see in the above example, merge sort first breaks the unsorted list into sorted sublists, and then keep merging these sublists, to finlly get the complete sorted list.

**Sorting using Merge Sort Algorithm**

/\* a[] is the array, p is starting index, that is 0,

and r is the last index of array. \*/

Lets take a[5] = {32, 45, 67, 2, 7} as the array to be sorted.

void **mergesort**(int a[], int p, int r)

{

int q;

if(p < r)

{

q = floor( (p+r) / 2);

mergesort(a, p, q);

mergesort(a, q+1, r);

merge(a, p, q, r);

}

}

void **merge**(int a[], int p, int q, int r)

{

int b[5]; //same size of a[]

int i, j, k;

k = 0;

i = p;

j = q+1;

while(i <= q && j <= r)

{

if(a[i] < a[j])

{

b[k++] = a[i++]; // same as b[k]=a[i]; k++; i++;

}

else

{

b[k++] = a[j++];

}

}

while(i <= q)

{

b[k++] = a[i++];

}

while(j <= r)

{

b[k++] = a[j++];

}

for(i=r; i >= p; i--)

{

a[i] = b[--k]; // copying back the sorted list to a[]

}

}

**Complexity Analysis of Merge Sort**

**Worst Case Time Complexity :** O(n log n)

**Best Case Time Complexity :** O(n log n)

**Average Time Complexity :** O(n log n)

**Space Complexity :** O(n)

* Time complexity of Merge Sort is O(n Log n) in all 3 cases (worst, average and best) as merge sort always divides the array in two halves and take linear time to merge two halves.
* It requires equal amount of additional space as the unsorted list. Hence its not at all recommended for searching large unsorted lists.
* It is the best Sorting technique for sorting **Linked Lists**.

**Heap Sort Algorithm**

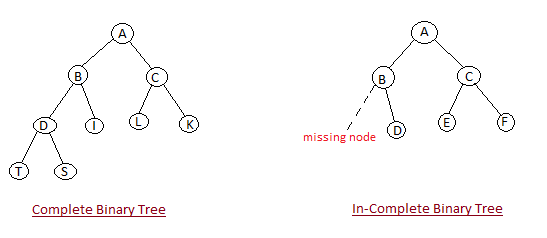
Heap Sort is one of the best sorting methods being in-place and with no quadratic worst-case scenarios. Heap sort algorithm is divided into two basic parts :

* Creating a Heap of the unsorted list.
* Then a sorted array is created by repeatedly removing the largest/smallest element from the heap, and inserting it into the array. The heap is reconstructed after each removal.

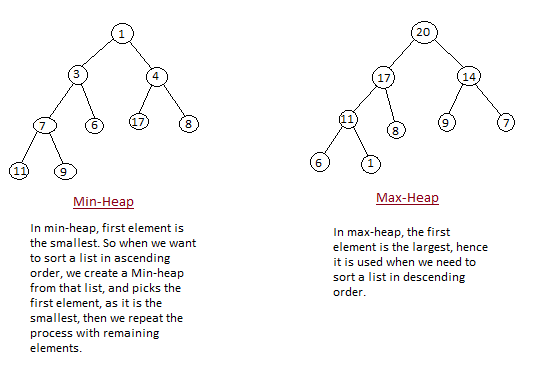
**What is a Heap ?**

Heap is a special tree-based data structure, that satisfies the following special heap properties :

1. **Shape Property :** Heap data structure is always a Complete Binary Tree, which means all levels of the tree are fully filled.



1. **Heap Property :** All nodes are either *[greater than or equal to]* or *[less than or equal to]* each of its children. If the parent nodes are greater than their children, heap is called a **Max-Heap**, and if the parent nodes are smalled than their child nodes, heap is called **Min-Heap**.



**How Heap Sort Works**

Initially on receiving an unsorted list, the first step in heap sort is to create a Heap data structure(Max-Heap or Min-Heap). Once heap is built, the first element of the Heap is either largest or smallest(depending upon Max-Heap or Min-Heap), so we put the first element of the heap in our array. Then we again make heap using the remaining elements, to again pick the first element of the heap and put it into the array. We keep on doing the same repeatedly untill we have the complete sorted list in our array.

In the below algorithm, initially **heapsort()** function is called, which calls **buildheap()** to build heap, which inturn uses **satisfyheap()** to build the heap.

**Sorting using Merge Sort Algorithm**

/\* Below program is written in C++ language \*/

void heapsort(int[], int);

void buildheap(int [], int);

void satisfyheap(int [], int, int);

void main()

{

int a[10], i, size;

cout << "Enter size of list"; // less than 10, because max size of array is 10

cin >> size;

cout << "Enter" << size << "elements";

for( i=0; i < size; i++)

{

cin >> a[i];

}

heapsort(a, size);

getch();

}

void **heapsort**(int a[], int length)

{

buildheap(a, length);

int heapsize, i, temp;

heapsize = length - 1;

for( i=heapsize; i >= 0; i--)

{

temp = a[0];

a[0] = a[heapsize];

a[heapsize] = temp;

heapsize--;

satisfyheap(a, 0, heapsize);

}

for( i=0; i < length; i++)

{

cout << "\t" << a[i];

}

}

void **buildheap**(int a[], int length)

{

int i, heapsize;

heapsize = length - 1;

for( i=(length/2); i >= 0; i--)

{

satisfyheap(a, i, heapsize);

}

}

void **satisfyheap**(int a[], int i, int heapsize)

{

int l, r, largest, temp;

l = 2\*i;

r = 2\*i + 1;

if(l <= heapsize && a[l] > a[i])

{

largest = l;

}

else

{

largest = i;

}

if( r <= heapsize && a[r] > a[largest])

{

largest = r;

}

if(largest != i)

{

temp = a[i];

a[i] = a[largest];

a[largest] = temp;

satisfyheap(a, largest, heapsize);

}

}

**Complexity Analysis of Heap Sort**

**Worst Case Time Complexity :** O(n log n)

**Best Case Time Complexity :** O(n log n)

**Average Time Complexity :** O(n log n)

**Space Complexity :** O(n)

* Heap sort is not a Stable sort, and requires a constant space for sorting a list.
* Heap Sort is very fast and is widely used for sorting.

**HASHING**

**Hashing** is the process of mapping large amount of data item to a smaller table with the help of a **hashing function**. The essence of hashing is to facilitate the next level searching method when compared with the linear or binary search. The advantage of this searching method is its efficiency to hand vast amount of data items in a given collection (i.e. collection size).  
Due to this hashing process, the result is a **Hash data structure** that can store or retrieve data items in an average time disregard to the collection size.  
  
**Hash Table** is the result of storing the hash data structure in a smaller table which incorporates the hash function within itself. The **Hash Function** primarily is responsible to map between the original data item and the smaller table itself. Here the mapping takes place with the help of **an output integer in a consistent range** produced when a given data item (any data type) is provided for storageand this **output integer range** determines the location in the smaller table for the data item. In terms of implementation, the hash table is constructed with the help of an array and the indices of this array are associated to the output integer range.  
  
Hash Table Example :  
Here, we construct a hash table for storing and retrieving data related to the citizens of a county and the social-security number of citizens are used as the indices of the array implementation (i.e. **key**). Let's assume that the table size is 12, therefore the hash function would be **Value modulus of 12**.  
  
Hence, the Hash Function would equate to:  
**(sum of numeric values of the characters in the data item) %12**   
Note! % is the modulus operator  
  
Let us consider the following social-security numbers and produce a hashcode:  
120388113D => 1+2+0+3+8+8+1+1+3+13=40  
Hence, (40)%12 => Hashcode=4  
  
310181312E => 3+1+0+1+8+1+3+1+2+14=34  
Hence, (34)%12 => Hashcode=10  
  
041176438A => 0+4+1+1+7+6+4+3+8+10=44  
Hence, (44)%12 => Hashcode=8  
  
Therefore, the Hashtable content would be as follows:  
-----------------------------------------------------  
0:empty  
1:empty  
2:empty  
3:empty  
4:occupied Name:Drew Smith SSN:120388113D  
5:empty  
6:empty  
7:empty  
8:occupied Name:Andy Conn SSN:041176438A  
9:empty  
10:occupied Name:Igor Barton SSN:310181312E  
11:empty  
-----------------------------------------------------

Graphs

A graph is a set of *vertices* and *edges* which connect them. We write:

**G = (V,E)**

where **V** is the set of vertices and the set of edges,

**E = { (vi,vj) }**

where **vi** and **vj** are in **V**.

#### Paths

A *path*, **p**, of length, **k**, through a graph is a sequence of connected vertices:

**p = <v0,v1,...,vk>**

where, for all **i** in (0,**k**-1:

**(vi,vi+1)** is in **E**.

#### Cycles

A graph contains no *cycles* if there is no path of non-zero length through the graph, **p = <v0,v1,...,vk>** such that **v0 = vk**.

#### Spanning Trees

A *spanning tree* of a graph, G, is a set of **|V|**-1 edges that connect all vertices of the graph.

#### Minimum Spanning Tree

In general, it is possible to construct multiple spanning trees for a graph, **G**. If a cost, **cij**, is associated with each edge, **eij = (vi,vj)**, then the minimum spanning tree is the set of edges, **Espan**, forming a spanning tree, such that:

**C = sum( cij** | all **eij** in **Espan** )

is a minimum.

#### Kruskal's Algorithm

This algorithm creates a *forest* of trees. Initially the forest consists of **n** single node trees (and no edges). At each step, we add one (the cheapest one) edge so that it joins two trees together. If it were to form a cycle, it would simply link two nodes that were already part of a single connected tree, so that this edge would not be needed.

The basic algorithm looks like this:

Forest MinimumSpanningTree( Graph g, int n, double \*\*costs ) {

Forest T;

Queue q;

Edge e;

T = ConsForest( g );

q = ConsEdgeQueue( g, costs );

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

do {

e = ExtractCheapestEdge( q );

} while ( !Cycle( e, T ) );

AddEdge( T, e );

}

return T;

}

The steps are:

1. The forest is constructed - with each node in a separate tree.
2. The edges are placed in a priority queue.
3. Until we've added **n**-1 edges,
   1. Extract the cheapest edge from the queue,
   2. If it forms a cycle, reject it,
   3. Else add it to the forest. Adding it to the forest will join two trees together.

Every step will have joined two trees in the forest together, so that at the end, there will only be one tree in T.

We can use a heap for the priority queue. The trick here is to detect cycles. For this, we need a *union-find* structure.

#### Union-find structure

To understand the union-find structure, we need to look at a *partition* of a set.

#### Partitions

A partitions is a set of sets of elements of a set.

* Every element of the set belong to one of the sets in the partition.
* No element of the set belong to more than one of the sub-sets.

or

* Every element of a set belongs to one *and only one* of the sets of a partition.

The forest of trees is a partition of the original set of nodes. Initially all the sub-sets have exactly one node in them. As the algorithm progresses, we form a union of two of the trees (sub-sets), until eventually the partition has only one sub-set containing all the nodes.

A partition of a set may be thought of as a set of *equivalence classes*. Each sub-set of the partition contains a set of equivalent elements (the nodes connected into one of the trees of the forest). This notion is the key to the cycle detection algorithm. For each sub-set, we denote one element as the *representative* of that sub-set or equivalence class. Each element in the sub-set is, somehow, equivalent and represented by the nominated representative.

As we add elements to a tree, we arrange that all the elements point to their representative. As we form a union of two sets, we simply arrange that the representative of one of the sets now points to any one of the elements of the other set.

So the test for a cycle reduces to: for the two nodes at the ends of the candidate edge, find their representatives. If the two representatives are the same, the two nodes are already in a connected tree and adding this edge would form a cycle. The search for the representative simply follows a chain of links.

Each node will need a representative pointer. Initially, each node is its own representative, so the pointer is set to NULL. As the initial pairs of nodes are joined to form a tree, the representative pointer of one of the nodes is made to point to the other, which becomes the representative of the tree. As trees are joined, the representative pointer of the representative of one of them is set to point to any element of the other. (Obviously, representative searches will be somewhat faster if one of the representatives is made to point directly to the other.)

#### Greedy operation

At no stage did we try to look ahead more than one edge - we simply chose the best one at any stage. Naturally, in some situations, this myopic view would lead to disaster! The simplistic approach often makes it difficult to prove that a greedy algorithm leads to the *optimal* solution. proof by contradiction is a common proof technique used: we demonstrate that if we didn't make the greedy choice now, a non-optimal solution would result. [Proving the MST algorithm](https://www.cs.auckland.ac.nz/%7Ejmor159/PLDS210/greedy_proof.html)  
is, happily, one of the simpler proofs by contradiction!

#### Data structures for graphs

You should note that we have discussed graphs in an abstract way: specifying that they contain nodes and edges and using operations like AddEdge, Cycle, *etc*. This enables us to define an abstract data type *without* considering implementation details, such as how we will store the attributes of a graph! This means that a complete solution to, for example, the MST problem can be specified before we've even decided how to store the graph in the computer. However, representation issues can't be deferred forever, so we need to examine ways of [representing graphs](https://www.cs.auckland.ac.nz/%7Ejmor159/PLDS210/graph_rep.html) in a machine. As before, the performance of the algorithm will be determined by the data structure chosen.