Sorting Algorithms

Sorting algorithms are algorithms used to rearrange elements in a certain order which can be numerical that is increasing or decreasing order or alphabetical order. Sorting algorithms helps understand program accuracy and speed. There are many sorting algorithms that are widely used for example bubble sort, quick sort, insertion sort, merge sort etc. Here we will be discussing a few other sorting algorithms.

Heap Sort

This is a sorting technique based on Binary Heap data structure. In a complete binary tree, it is expected to fill both the left and the right nodes before moving on to the next node. A Binary Heap is a Complete Binary Tree where values are stored such that the parent node is greater than the two children nodes.

Algorithm:

To construct the Binary Heap tree:

- Step 1: Consider the 1st element as the parent node.
- Step 2: The next element would be the child node. If the child node is greater than the parent node, swap the items in the tree.
- Step 3: Make sure to fill both the left and right node of the parent node before moving to a different level.

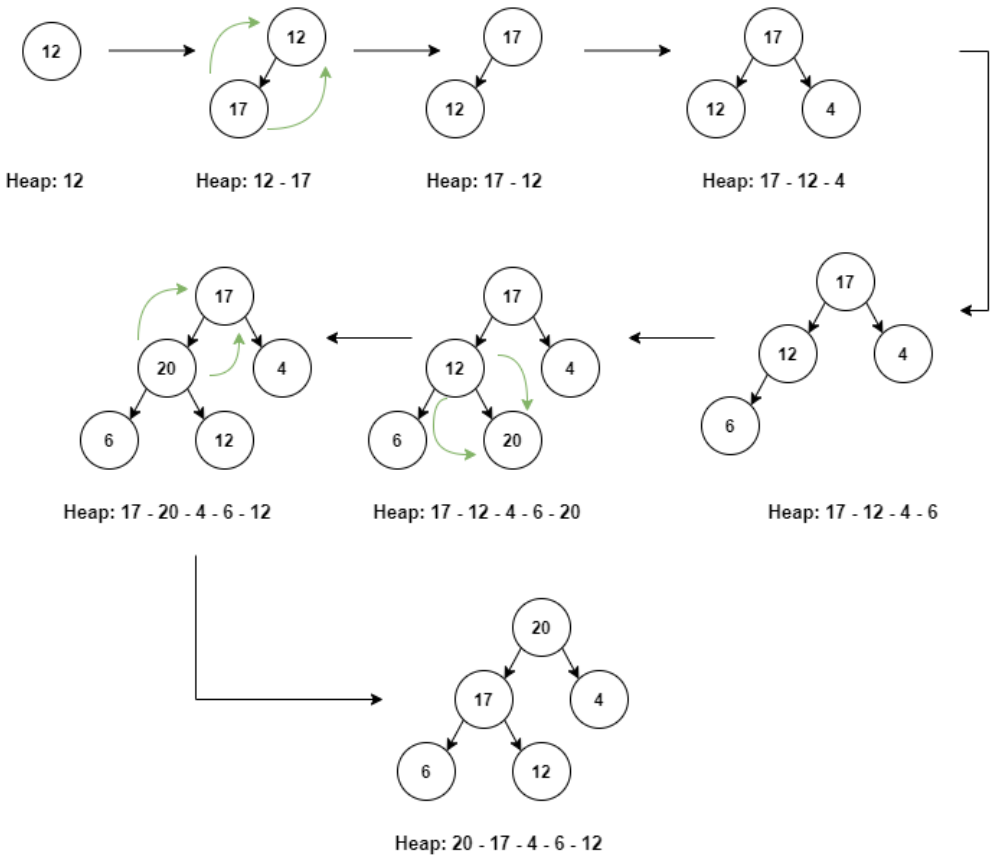
To perform the Heap Sort:

- Step 1: First construct the Binary Heap tree for the given elements.
- Step 2: Now replace the root of the heap tree with the last item and reduce the size of heap by 1.
- Step 3: Now check if the heap tree condition is satisfied, that is make sure if the parent node is less than the child node at every point. If not then swap the contents of the parent and the child node.
- Step 4: Repeat this until the size of the heap is 0.

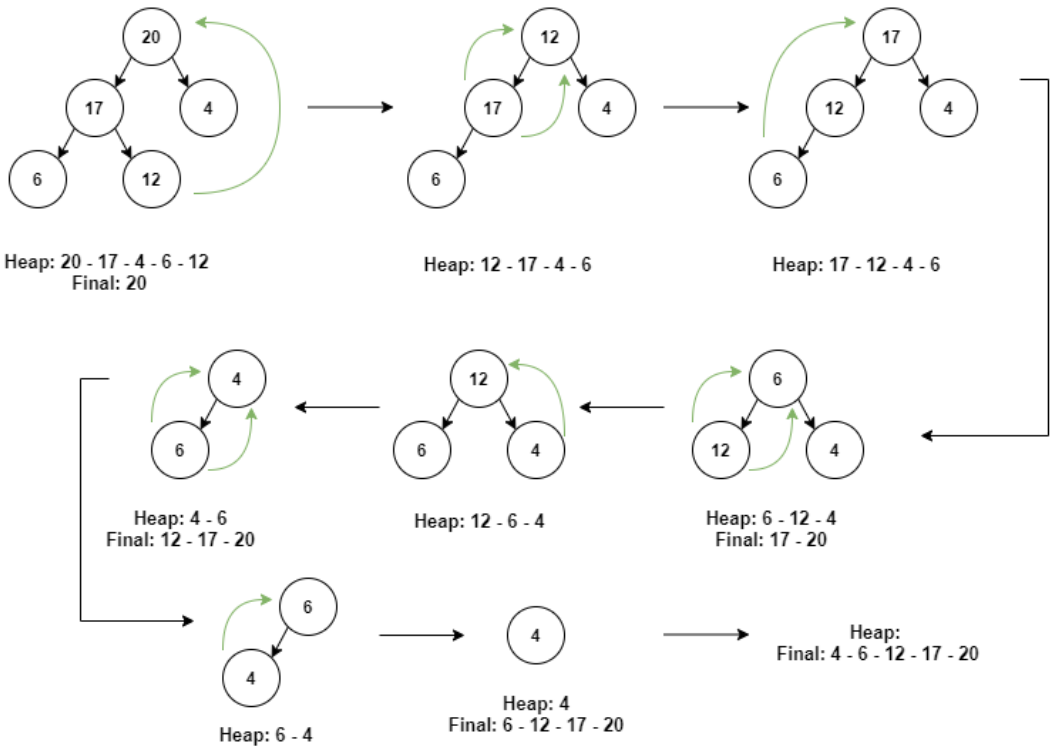
Example:

Consider the elements,

12 17 4 6 20 8



Binary Heap Tree after inserting the elements



Binary Heap Tree after deleting the elements

Pseudocode:

```
Maxheap (A, n, i){
    int largest = i;
    int left = (2*i);
    int right = (2*i) +1;
    while (left≤ n && A[left]>A[largest]) {
        largest = left;
    }
    while (right≤ n && A[right]>A[largest]) {
        largest = right;
    }
    if (largest != i) {
        swap(A[largest], A[i]);
        heap (A, n, largest);
    }
}
```

Radix Sort/ Bucket Sort

Radix sort is a non-comparative sorting algorithm. The main difference of this algorithm with many other sorting algorithms is that this technique can be used for both numerical and alphabetical sorting.

Algorithm:

- Step 1: For numerical values sorting, consider 10 buckets numbered from 0 to 9 respectively.
- Step 2: Find the number of digits in the largest number and make all the numbers of the same size as the largest number by adding 0 s to the left of the number.
- Step 3: Now based on the Least Significant Bit of all the numbers, sort the numbers into the 10 buckets.
- Step 4: Write back the numbers in order in which the buckets are placed.
- Step 5: Now consider the next Least Significant Bit and perform the same operation.
- Step 6: This process is repeated based on the number of digits in the largest number. That is suppose the largest digit is 14785 then this procedure must be repeated 5 times.

If we want to sort numerical values, we consider 10 buckets and start sorting based on the least significant bit. The number of passes is based on the number of digits that are present in the greatest number. If we want to sort alphabetical elements then we consider 26 buckets and we start sorting based on the Most Significant Bit.

Example:

Consider the example

20 6 15 221 704 9 145 80 151 11

We can see that the largest number here is 704 which has 3 digits. Now make all the elements of the same size. We get,

020 006 015 221 704 009 145 080 151 011

Bucket No.	Items
0	020 - 080
1	221 - 151 - 011
2	
3	
4	704
5	015 - 145
6	006
7	
8	
9	009

Pass 1:

The new sequence of values would be,

020 - 080 – 221 – 151 – 011 – 704 – 015 – 145 – 006 - 009

Bucket No.	Items
0	704 - 006 - 009
1	011 - 015
2	020 - 221
3	
4	145
5	151
6	
7	
8	080
9	

Pass 2:

The new sequence of values would be,

704 – 006 – 009 – 011 – 015 – 020 – 221 – 145 – 151 - 080

Bucket No.	Items
0	006 - 009 - 011 - 015 - 020 - 080
1	145 - 151
2	221
3	
4	
5	
6	
7	704
8	
9	

Pass 3:

The new sequence of values would be,

006 – 009 – 011 – 015 – 020 – 080 – 145 – 151 – 221 – 704

After the final pass we get the sorted elements as

6 9 11 15 20 80 145 151 221 704

Pseudocode:

```
radixSort (input A, d) {
    for (j = 1 j≤ d j++) {
        int count [10] = {0};
        for (i = 0 i≤ A i++) {
            count [key of(A[i]) in pass j] ++;
        }
        for (k = 1 k≤10 k++) {
            count[k] = count[k] + count[k-1]
        }
        for (i = n-1 i>0 i--) {
            result [ count [key of(A[i])]] = A[j]
            count [key of(A[i])]—
        }
        for (i=0 i≤ n i++) {
            A[i] = result[i]
        }
    }
}
```

Shell Sort

A shell sort algorithm is similar to the insertion sort algorithm except that in this algorithm, we compare and swap the 2 numbers which are apart from each other at a fixed gap. This gap is then reduced after every pass. This algorithm is mainly used if the smaller elements are present on the right side of the sequence and the larger elements are present on the left of the sequence.

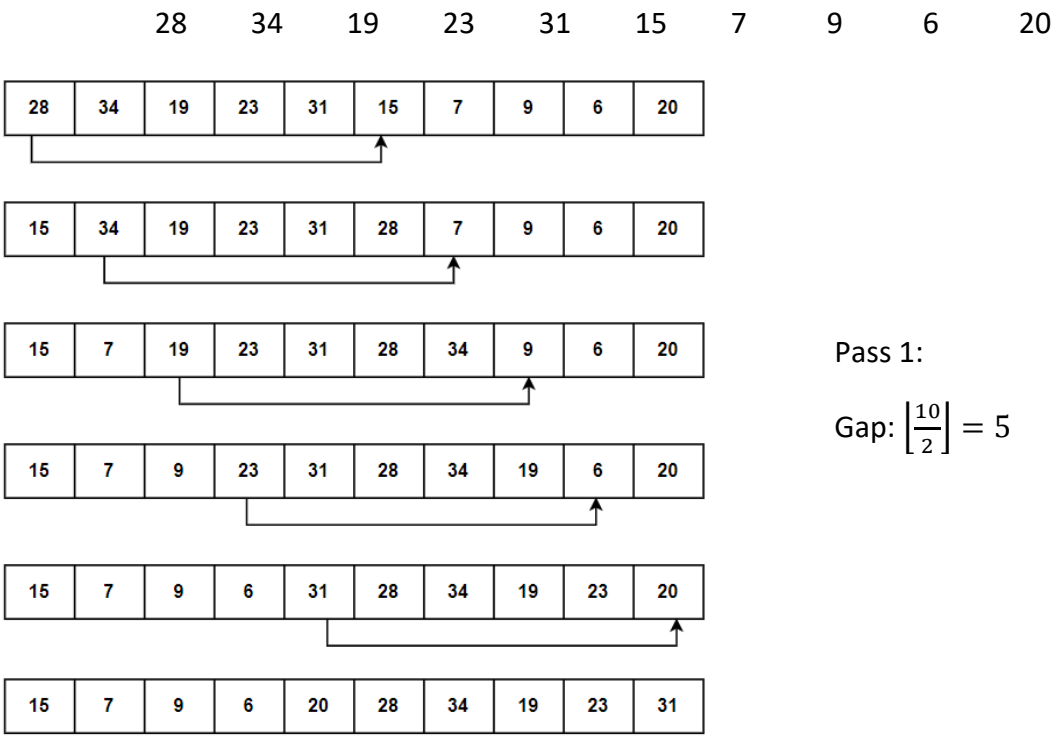
Algorithm:

There are several methods for choosing the gap for comparing, but the most commonly used is mentioned below.

- Step 1: First consider the gap say gap1 as N/2 where N is the number of elements in the sequence.
- Step 2: Now compare the 2 elements of the sequence which are at a distance of the gap obtained above.
- Step 3: Suppose there is a swap performed, then also compare this element with the element at the mentioned gap even on the left side of the sequence. For example, if the gap is considered as 4 and the element in consideration is 15 then compare the element with the element in 11^{th} position too.
- Step 4: After every pass, the gap value changes as $\left\lfloor \frac{Gap1}{2} \right\rfloor$.
- Step 5: Repeat this procedure till the gap value reaches 1. At this point, the shell algorithm works exactly like the insertion sort algorithm.

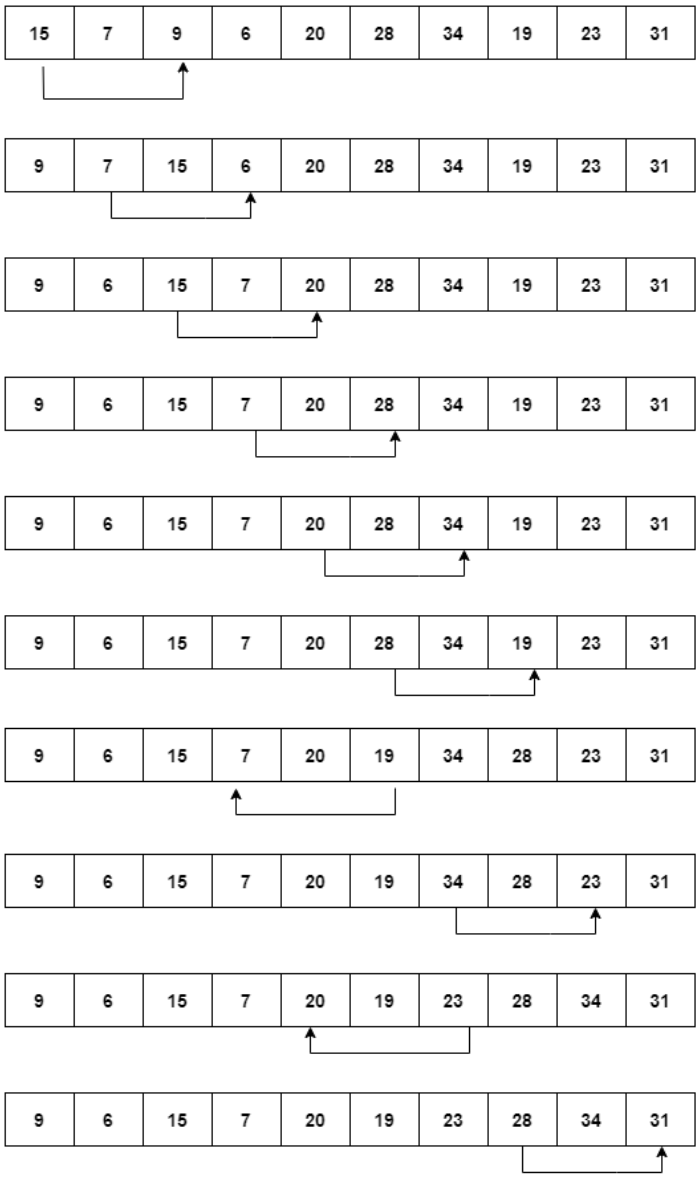
Example:

Consider the example,



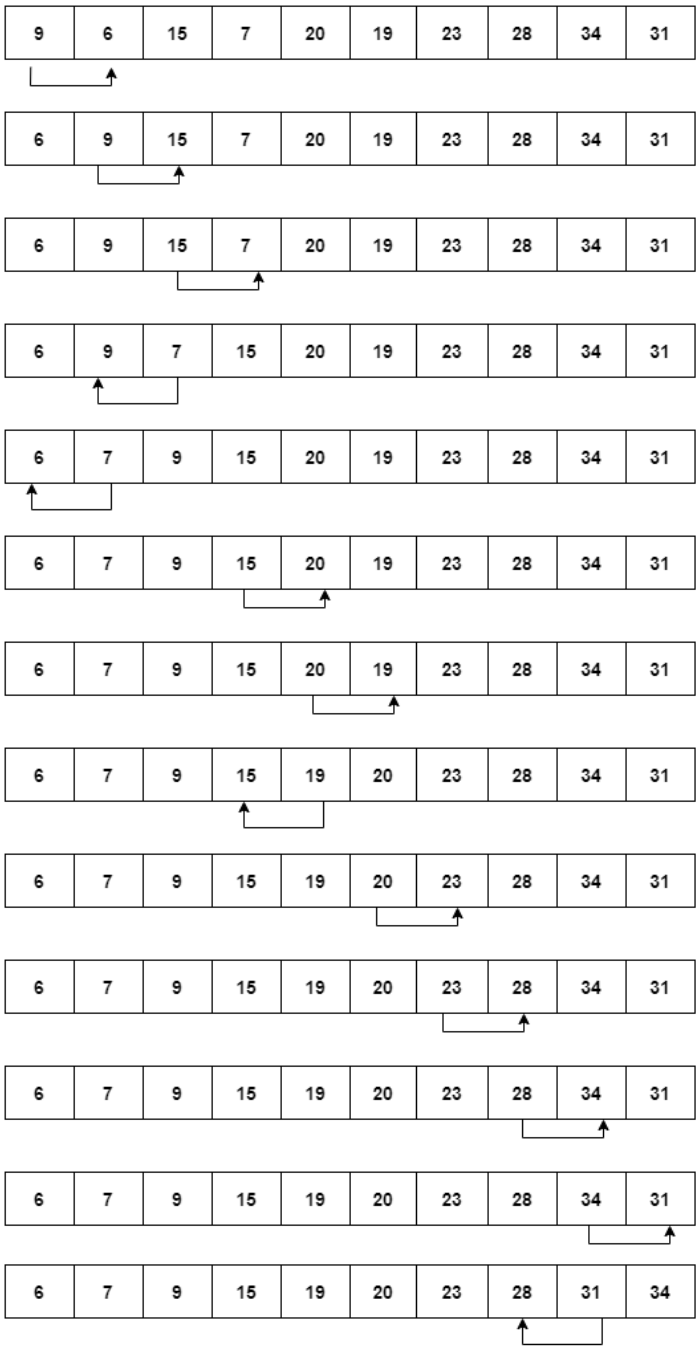
Pass 1:

Gap: $\left\lfloor \frac{10}{2} \right\rfloor = 5$



Pass 2:

Gap: $\left\lfloor \frac{5}{2} \right\rfloor = 2$



Pass 3:

Gap: $\left\lfloor \frac{2}{2} \right\rfloor = 1$

Pseudocode:

```
for (gap=n/2; gap≥ 1; gap/2) {
    for (j=gap; j<n; j++) {
        for(i=j-gap;i≥ 0;i-gap){
            if (a[i+gap]>a[i]){
                break;
            }
```

```
                else {
                        swap(a[i+gap],a[i]);
                }
            }
        }
    }
```

Conclusion:

Here we have seen 3 types of sorting algorithms Heap Sort, Radix Sort and Shell Sort. The time complexity to insert into the heap tree would be $O(\log n)$ since we insert at the end and compare only with the parent element and not with all the elements at any point. Consider n elements to be inserted, the time complexity would be $O(n \log n)$.

The time complexity to delete an element from the Max-Heap would also be $O(n \log n)$. Thus, we can conclude that the time complexity of the entire Heap Sort algorithm in best case, worse case and average case would be,

$$Time\ Complexity\ of\ Heap\ Sort: O(n \log n)$$

In radix sort algorithm, the time complexity is given $O(d * (n + b))$ where b is the base for representing the numbers. It can also be considered as the number of buckets. In case of numbers $b = 10$, in case of strings $b = 26$. Suppose k is the maximum possible value, the d is $O(\log_b k)$. Thus, the overall complexity of this sorting algorithm is

$$Time\ Complexity\ of\ Radix\ Sort: O(\log_b k * (n + b))$$

In shell sort, the best-case is when the array is already sorted. Now since we compare with just the right element and not all elements, the complexity would be $O(\log n)$. Since there are n elements, we consider the efficiency to be $O(n \log n)$. As sad earlier, there are multiple ways to choose the gap sequences and the worst-case complexity changes each time. In the method used above to determine the complexity, the worst-case complexity would be $O(n^2)$.

$$Time\ complexity\ Shell\ Sort\ Best\ Case: O(n \log n)$$

$$Time\ complexity\ Shell\ Sort\ Worst\ Case: O(n^2)$$