Sorting

Introduction

Sorting is a fundamental operation in computer science. A good deal of effort in computing is related to making data available to users in some particular order. The concept of an ordered set of data is one that has considerable impact on our daily lives. **Sorting** arranges data in a sequence which makes searching easier

Sorting is the process of arranging the elements of a collection so that they are placed in some relevant order which may ascending or descending, such as increasing or decreasing, with numeric data or alphabetically, with character data. Sorting is important because it is often the first step in more complex algorithms.

For example, if we have an array that is declared and initialized as

```
int A[] = {21, 34, 11, 9, 1, 0, 22};
```

Then the sorted array (ascending order) can be given as:

```
A[] = {0, 1, 9, 11, 21, 22, 34};
```

There are many different sorting methods and they can be classified in various ways. One classification is based on where the data to be sorted is stored:

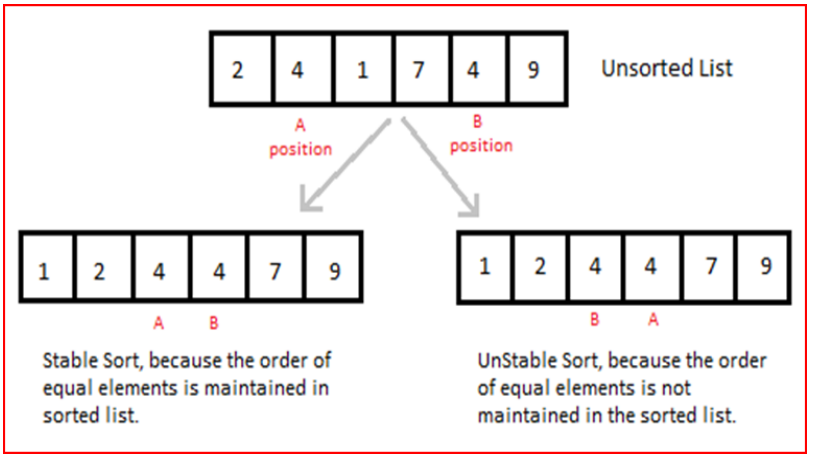
- ***Internal*** sorts are designed for data stored in main memory.
- ***External*** sorts are designed for data stored in secondary memory (e.g., on disk). External sorting is applied when there is voluminous data that cannot be stored in the memory.

Most of the sorting methods we consider here are internal sorts, and we will only consider internal sorting in this course.

A sorting algorithm is defined as an algorithm that puts the elements of a list in a certain order, which can be numerical order, lexicographical order, or any user-defined order. Efficient sorting algorithms are widely used to optimize the use of other algorithms like search and merge algorithms which require sorted lists to work correctly.

Sorting algorithms often have additional properties that are of interest, depending on the application. Here are two important properties.

- **In place sorting**
  Sorting of a data structure does not require any external data structure for storing the intermediate steps. It is possible to sort very large lists without the need to allocate additional working storage.
- **Stable sorting**
  If the same element is present multiple times, then they retain the original relative order of positions.



Sorting Efficiency

There are many techniques for sorting. Implementation of particular sorting technique depends upon situation. Sorting techniques mainly depends on two parameters; time and space.

The efficiency of an algorithm is always stated as a function of the input size.

Worst case scenario: maximum number of computation steps taken on any input size n.

Time Complexity is defined to be the time the computer takes to run a program (or algorithm in our case).
Space complexity is defined to be the amount of memory the computer needs to run a program.
We assume each operation in a program take one time unit.

Time Units to Compute

- 1 for the assignment.
- 1 assignment, $n+1$ tests, and n increments.
- n loops of 3 units for an assignment, an addition, and one multiplication.
- 1 for the return statement.

Total: $1+(1+n+1+n)+3n+1$
$= 5n+4 = O(n)$

```
int sum (int n)
{
  int partial_sum = 0;
  for (int i = 1; i <= n; i++)
    partial_sum = partial_sum + (i * i );
  return partial_sum;
}
```

Bubble Sort: (Exchange Sort)

Bubble sort is a very simple method that sorts the array elements by repeatedly moving the largest element to the highest index position of the array segment (in case of arranging elements in ascending order). In *bubble sorting*, consecutive adjacent pairs of elements in the array are compared with each other. If the element at the lower index is greater than the element at the higher index, the two elements are interchanged so that the element is placed before the bigger one (for example a[i] with a[i + 1]). This process will continue till the list of unsorted elements exhausts.

This procedure of sorting is called bubble sorting because smaller elements (in ascending order) 'bubble' to the top of the list. Note that at the end of the first pass, the largest element in the list will be placed at its proper position (i.e., at the end of the list).

Technique

The basic methodology of the working of bubble sort is given as follows:

a. In Pass 1, A[0] and A[1] are compared, then A[1] is compared with A[2], A[2] is compared with A[3], and so on. Finally, A[N–2] is compared with A[N–1]. Pass 1 involves n–1 comparisons and places the biggest element at the highest index of the array.

b. In Pass 2, A[0] and A[1] are compared, then A[1] is compared with A[2], A[2] is compared with A[3], and so on. Finally, A[N–3] is compared with A[N–2]. Pass 2 involves n–2 comparisons and places the second biggest element at the second highest index of the array.

c. In Pass 3, A[0] and A[1] are compared, then A[1] is compared with A[2], A[2] is compared with A[3], and so on. Finally, A[N–4] is compared with A[N–3]. Pass 3 involves n–3 comparisons and places the third biggest element at the third highest index of the array.

d. In Pass n–1, A[0] and A[1] are compared so that A[0]<A[1]. After this step, all the elements of the array are arranged in ascending order.

Function

```
BubbleSort(A, n)
{
    for(i = 0; i <n-1; i++)
    {
        for(j = 0; j < n-i-1; j++)
        {
            if(A[j] > A[j+1])
            {
                temp = A[j];
                A[j] = A[j+1];
                A[j+1] = temp;
            }
        }
    }
}
```

Algorithm

BUBBLE_SORT(A, N)

```
Step 1: start
Step 2: Repeat Step 3 for i = 0 to N-1
Step 3:   Repeat for J = 0 to N-1
Step 4:      IF A[J] > A[J + 1]
               SWAP A[J] and A[J+1]
          [END OF INNER LOOP]
        [END OF OUTER LOOP]
Step 5: EXIT
```

For example, consider the following array:

Initially:

25	57	48	37	12	92	86	33
0	1	2	3	4	5	6	7

After Pass 1:

25	48	37	12	57	86	33	92

After Pass 2:

25	37	12	48	57	33	86	92
0	1	2	3	4	5	6	7

After Pass 3:

25	12	37	48	33	57	86	92
0	1	2	3	4	5	6	7

After Pass 4:

12	25	37	33	48	57	86	92
0	1	2	3	4	5	6	7

After Pass 5:

12	25	33	37	48	57	86	92
0	1	2	3	4	5	6	7

After Pass 6:

12	25	33	37	48	57	86	92
0	1	2	3	4	5	6	7

After Pass 7:

12	25	33	37	48	57	86	92
0	1	2	3	4	5	6	7

Here, we notice that after each pass, an element is placed in its proper order and is not considered in succeeding passes. Furthermore, we need n – 1 passes to sort n elements.

Time Complexity:

Inner loop executes for (n-1) times when i=0, (n-2) times when i=1 and so on:

Time complexity = (n-1) + (n-2) + (n-3) + ……. +2 +1 = $O(n^2)$.

There is no best-case linear time complexity for this algorithm.

Space Complexity:

Since no extra space besides 3 variables is needed for this sorting: Space complexity = O(n)

Bubble Sort Optimization

Consider a case when the array is already sorted. In this situation no swapping is done but we still have to continue with all n–1 passes. We may even have an array that will be sorted in 2 or 3 passes but we still have to continue with rest of the passes. So once we have detected that the array is sorted, the algorithm must not be executed further. This is the optimization over the original bubble sort algorithm. In order to stop the execution of further passes after the array is sorted, we can have a variable flag which is set to FALSE before each pass and is made TRUE when a swapping is performed.

Optimized Bubble Sort Function

```
BubbleSort(A, n)
{
     bool expression
    for(i = 0; i <n-1; i++)
    {
        flag = false;
        for(j = 0; j < n-i-1; j++)
        {
            if(A[j] > A[j+1])
            {
                temp = A[j];
                 A[j] = A[j+1];
                A[j+1] = temp;
                flag  = true;
            }
        }
        If (flag == false)
            break;
    }
}
```

Complexity of Optimized Bubble Sort Algorithm

In the best case, when the array is already sorted, the optimized bubble sort will take O(n) time. In the worst case, when all the passes are performed, the algorithm will perform slower than the original algorithm.

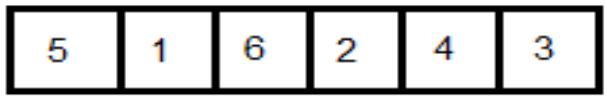
INSERTION SORT

Insertion sort is a very simple sorting algorithm in which the sorted array (or list) is built one element at a time. It works the way we might put a hand of cards in order. The hand is scanned for the first card that is lower than the one to the left. When such a case is found, the smaller card is picked out and moved to or inserted at the correct location.

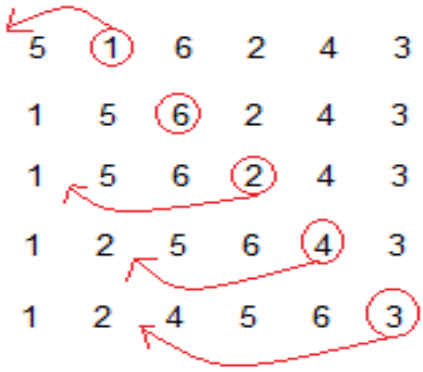
Technique

Insertion sort works as follows:

- The array of values to be sorted is divided into two sets. One that stores sorted values and another that contains unsorted values.
- The sorting algorithm will proceed until there are elements in the unsorted set.
- Suppose there are n elements in the array. Initially, the element with index 0 (assuming LB = 0) is in the sorted set. Rest of the elements are in the unsorted set.
- The first element of the unsorted partition has array index 1 (if LB = 0).
- During each iteration of the algorithm, the first element in the unsorted set is picked up and inserted into the correct position in the sorted set.



Lets take this Array.



(Always we start with the second element as key.)

As we can see here, in insertion sort, we pick up a key, and compares it with elemnts ahead of it, and puts the key in the right place

5 has nothing before it.

1 is compared to 5 and is inserted before 5.

6 is greater than 5 and 1.

2 is smaller than 6 and 5, but greater than 1, so its is inserted after 1.

And this goes on...

Insertion Sort Function

```
InsertionSort(a, n)
{
    int i, j, key;
    for(i=1; i<n; i++)
    {
        key = a[i];
        j = i-1;
        while(j>=0 && key < a[j])
        {
            a[j+1] = a[j];
            j--;
        }
        a[j+1] = key;
    }
}
```

Algorithm

INSERTION-SORT (ARR, N)

```
Step 1: Start
Step 2: Repeat Steps 2 to 5 for K = 1 to N – 1
Step 3:     SET TEMP = ARR[K]
Step 4:     SET J = K - 1
Step 5:     Repeat while J>=0 && TEMP <= ARR[J]
                  SET ARR[J + 1] = ARR[J]
                  SET J = J - 1
            [END OF INNER LOOP]
Step 6:     SET ARR[J + 1] = TEMP
        [END OF LOOP]
Step 7: EXIT
```

Time Complexity:

Best Case:

The best case occurs when the array is already sorted. In this case, the running time of the algorithm has a linear running time (i.e., O(n)). This is because, during each iteration, the first element from the unsorted set is compared only with the last element of the sorted set of the array.

Worst Case:

The worst case occurs when the array is sorted in the reverse order.

Inner loop executes for 1 time when I = 1, 2 times when I = 2 and n-1 times when I = n-1:

Therefore, Time complexity = 1+2+…….+ N-2 + N-1 = O(n2).

SELECTION SORT

The basic idea of a **selection sort** of a list is to make a number of passes through the list or a part of the list and, on each pass, select one element to be correctly positioned. For example, on each pass through a sublist, the smallest element in this sublist might be found and then moved to its proper location. Selection sorting is conceptually the simplest sorting algorithm.

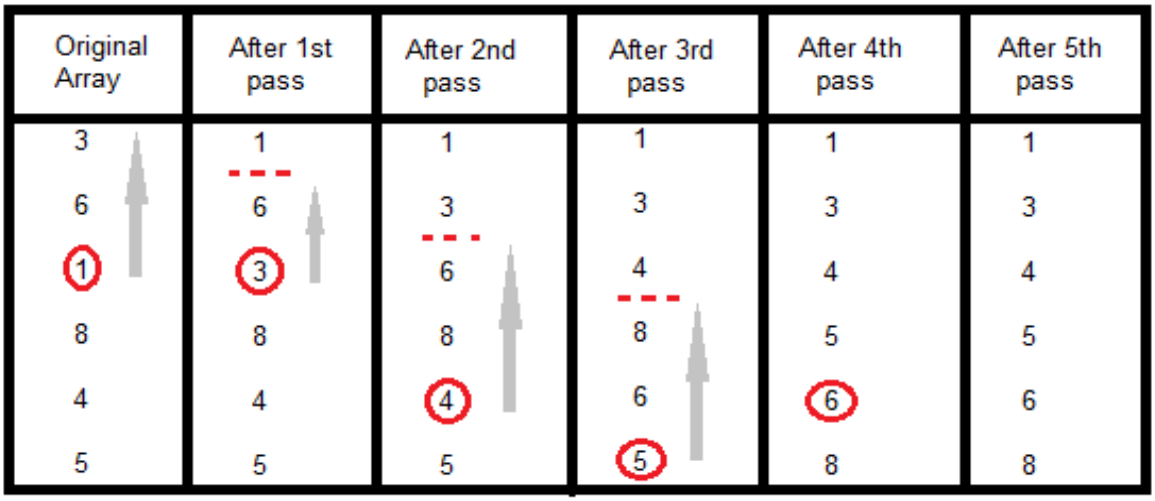
Technique

Consider an array *ARR* with *N* elements. Selection sort works as follows:

First find the smallest value in the array and place it in the first position. Then, find the second smallest value in the array and place it in the second position. Repeat this procedure until the entire array is sorted. Therefore,

- In Pass 1, find the position *POS* of the smallest value in the array and then swap *ARR[POS]* and *ARR[0]*. Thus, *ARR[0]* is sorted.
- In Pass 2, find the position *POS* of the smallest value in sub-array of *N–1* elements. Swap *ARR[POS]* with *ARR[1].* Now, *ARR[0]* and *ARR[1]* is sorted.
- In Pass *N–1*, find the position *POS* of the smaller of the elements *ARR[N–2]* and *ARR[N–1]*. Swap *ARR[POS]* and *ARR[N–2]* so that *ARR[0], ARR[1], ..., ARR[N–1]* is sorted.

How Selection Sorting Works



Selection Sort Function:

```
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 element at j is less than element at min position
            if(a[j] < a[min])
            {
                min = j; //then set min as j
            }
        }
        temp = a[i];
        a[i] = a[min];
        a[min] = temp;
```

Algorithm

SELECTION_SORT(A, N)

```
Step 1: start
Step 2: Repeat Step 3 to 6 for I = 0 to N-1
Step 3:   set min = I
Step 4:   Repeat for J = i+1 to N-1
Step 5:      IF A[J] < A[min]
               Set min = J
           [END OF INNER LOOP]
Step 6:   SWAP A[I] and A[min]
        [END OF OUTER LOOP]
Step 7: EXIT
```

```
    }
  }
```

Time Complexity:

- ✓ Inner loop executes for (n-1) times when i=0, (n-2) times when i=1 and so on:
  Time complexity = (n-1) + (n-2) + (n-3) + …… +2 +1 = **$O(n^2)$**
- ✓ There is no best-case linear time complexity for this algorithm.

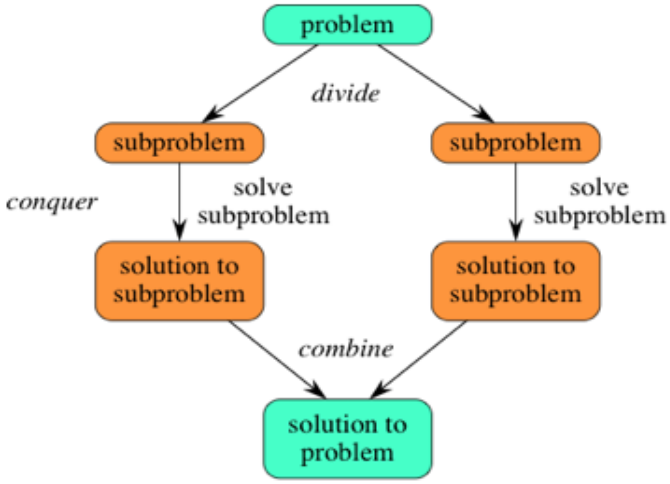
DIVIDE AND CONQUER PARADIGM.

The Main Ideas:

It is almost always much easier to sort short lists than long ones. If the number of entries to be sorted doubles, then the work more than doubles (with insertion or selection sort it quadruples, roughly). Hence if we can find a way to divide the list into two roughly equal-sized lists and sort them separately, then we will save work. If, for example, you were working in a library and were given a thousand index cards to put in alphabetical order, then a good way would be to distribute them into piles according to the first letter and sort the piles separately.
We have an application of the idea of dividing a problem into smaller but similar subproblems; that is, of *divide and conquer*.

Divide-and-conquer, breaks a problem into subproblems that are similar to the original problem, recursively solves the subproblems, and finally combines the solutions to the subproblems to solve the original problem.
A divide-and-conquer algorithm can be thought of as having three parts:

- ❖ **Divide** the problem into a number of subproblems that are smaller instances of the same problem.
- ❖ **Conquer** the subproblems by solving them recursively. If they are small enough, solve the subproblems as base cases.
- ❖ **Combine** the solutions to the subproblems into the solution for the original problem.



We still must decide how we are going to partition the list into two sublists and, after they are sorted, how we are going to combine the sublists into a single list. There are two methods, each of which works very well in different circumstances; ***Mergesort* and *Quicksort*.**

Quick Sort

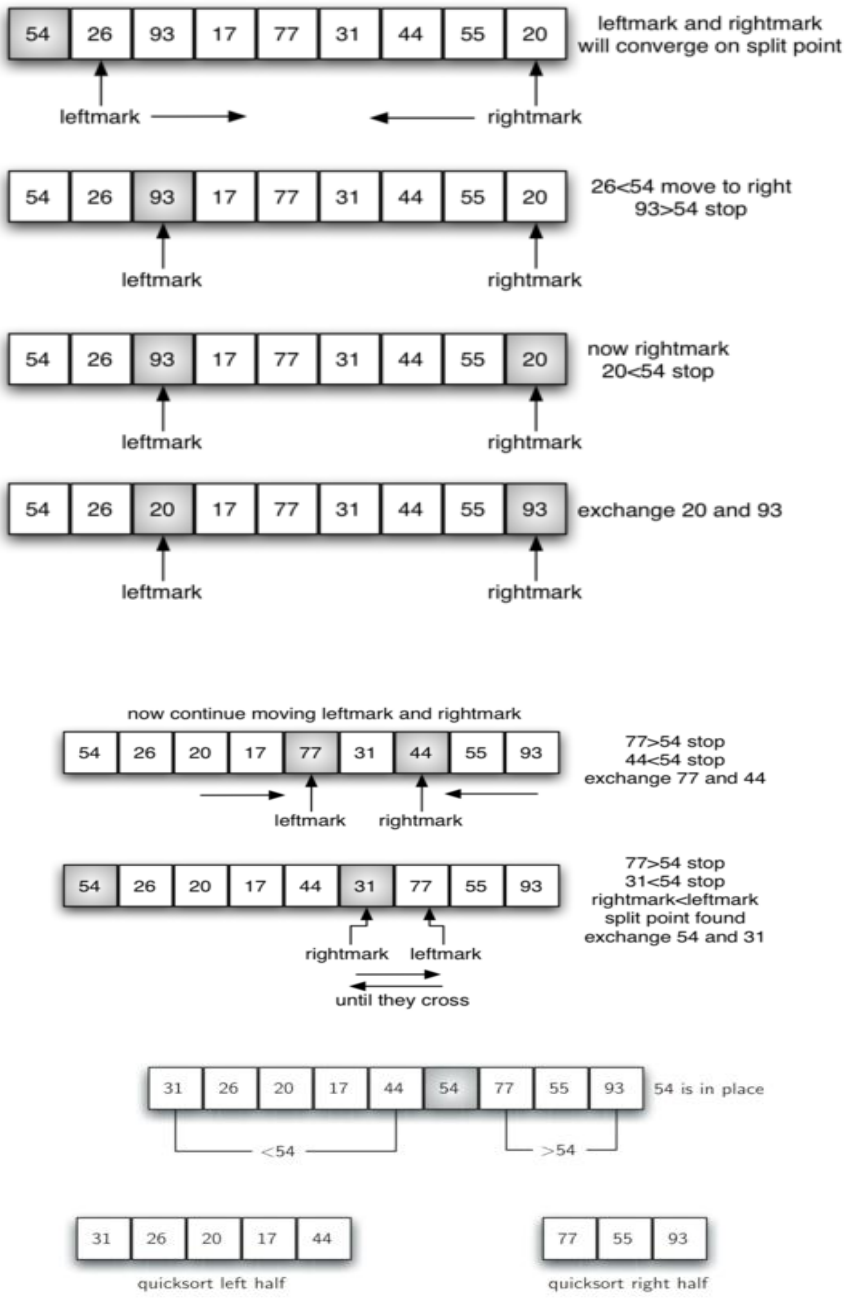
Quicksort is a divide-and-conquer method for sorting. It works by *partitioning* an array into two subarrays, then sorting the subarrays independently. That is why it is also known as partition exchange sort. This sorting algorithm is probably used more widely than any other sorting algorithm because, as the name suggests, sorts any list very quickly.
This method was invented and named by C. A. R. HOARE, and makes *O(n log n)* comparisons in the average case to sort an array of *n* elements. Basically, the quick sort algorithm is faster than other `O(nlogn)` algorithms, because its efficient implementation can minimize the probability of requiring quadratic time.

This algorithm works as follows:

1. Select an element as **pivot** from the array elements. [The role of **the pivot value** is to assist with splitting the list. The actual position where the pivot value belongs in the final sorted list, commonly called the **split point**, will be used to divide the list for subsequent calls to the quick sort.]
2. Rearrange the elements in the array in such a way that all elements that are less than the pivot appear before the pivot and all elements greater than the pivot element come after it (equal values can go either way). After such a partitioning, the pivot is placed in its final position.
   This is called the *partition* operation.
3. Recursively sort the two sub-arrays thus obtained. (One with sub-list of values smaller than that of the pivot element and the other having higher value elements.)

The *base case* of the recursion occurs when the array has zero or one element because in that case the array is already sorted. After each iteration, one element (pivot) is always in its final position. Hence, with every iteration, there is one less element to be sorted in the array.

Thus, the main task is to find the pivot element, which will partition the array into two halves. We must first decide, in order to partition the list into two pieces, what key to choose as the pivot. We are free to choose any number we wish, but for consistency, we shall adopt a definite rule. Perhaps the simplest rule is to choose the first number on a list as the pivot.



Algorithm

```
QUICKSORT(ARR, lo, hi)
Step 1: Start
Step 2: if (lo < hi) then
            INDEX = partition(ARR, lo, hi);
            QUICKSORT(ARR, lo, INDEX);
            QUICKSORT(ARR, INDEX+1, hi);
        END IF
Step 3: EXIT
```

```
PARTITION(ARR, left, right)
{
        x =left; y =right ;
        pivot = ARR[left];
        while(x<y)
        {
          do{
            x++;
          }while(ARR[x] < pivot) ;
          while(ARR[y] >pivot) y--;
          if(x<y)
            swap(ARR[x], ARR[y]);
        }
        ARR[left] = ARR[y];
        ARR[y] = pivot;
        return y;    //return position of pivot
}
```

QuickSort Time Complexity:

Best Case:

Divides the array into two partitions of equal size, therefore
T(n) = 2T(n/2) + O(n) , Solving this recurrence we get,
T(n)=O(nlogn)

Worst case:

When one partition contains the n-1 elements and another partition contains only one element. Therefore its recurrence relation is:
T(n) = T(n-1) + O(n), Solving this recurrence we get $T(n)=O(n^2)$

Average case:

Good and bad splits are randomly distributed across throughout the tree
T1(n)= 2T'(n/2) + O(n) Balanced
T'(n)= T(n –1) + O(n) Unbalanced
Solving:

B(n)= 2(B(n/2 –1) + Θ(n/2)) + Θ(n)
= 2B(n/2 –1) + Θ(n)
= O(nlogn)
=>T(n)=O(nlogn)

```
/*Implementation of Selection Sort
 * selectionsort.c
 *
 * Copyright 2020 gobinda
<gobinda@GOBINDA-PC>
 *
 */
#include <stdio.h>
#include <stdlib.h>
//Function Prototypes
void fillArray(int [], int);//for filling the array
void displayArray(int [], int);//Displays array
values
void selectionSort(int [], int);//Sorting function
itself
void swap(int *, int *);// swap two values.
Pointers because it swaps two value at two
index of an array.

int main(int argc, char **argv)
{
    int *arr;
    int size;
    printf(" How many numbers to store in
array  ");
    scanf("%d", &size);
    arr = malloc(size*sizeof(int));
    printf("\n Filling Array Now");
    fillArray(arr, size);
    printf("\n Unsorted array\n");
    displayArray(arr, size);
    selectionSort(arr, size);
    printf("\n\n Sorted Array\n");
    displayArray(arr, size);

    return 0;
}

//This function fills an array with random
numbers
void fillArray(int arr[], int sz)
{
    printf("\n Enter random values to fill an
array: ");
    for(int i = 0; i < sz; i++)
            scanf("%d",&arr[i]);

}

//This function displays elements of the array
```

```
void displayArray(int arr[], int sz)
{
    printf("\n Array Elements are: ");
    for(int i = 0; i < sz; i++)
        printf("\t%d",arr[i]);
    printf("\n");
}

//This function sorts array element with
selection sort algorithm
void selectionSort(int arr[], int sz)
{
    int i, j, min;
    for(i=0; i < sz-1; i++ )
    {
        min = i; //setting min as i
        for(j=i+1; j < sz; j++)
        { //if element at j is less than
element at min position
            if(arr[j] < arr[min])
            {
                min = j; //then set
min as j
            }
        }
        swap(&arr[i], &arr[min]);
    }
}

//This function swaps two values
void swap(int *a, int *b)
{
    int temp;
        temp = *a;
        *a= *b;
        *b = temp;
}
```

//

```
/* Implementation of Insertion Sort
 * insertionSort.c
 *
 * Copyright 2020 gobinda
<gobinda@GOBINDA-PC>
 *
 */
#include <stdio.h>
#include <stdlib.h>
//function prototypes
void fillArray(int [], int);
void displayArray(int [], int);
void insertionSort(int [], int);

int main(int argc, char **argv)
{
    int *arr;
    int size;
    printf(" How many numbers to store in
    array  ");
    scanf("%d", &size);
    arr = malloc(size*sizeof(int));
    /*address of      memory block defined by
    size*int is assigned to pointer arr. */
    printf("\n Filling Array Now");
    fillArray(arr, size);
    printf("\n Unsorted array\n");
    displayArray(arr, size);
    insertionSort(arr, size);
    printf("\n\n Sorted Array\n");
    displayArray(arr, size);

    return 0;
}

//This function fills an array with random
numbers
void fillArray(int arr[], int sz)
{
    printf("\n Enter random values to fill an
array: ");
    for(int i = 0; i < sz; i++)
        scanf("%d",&arr[i]);

}

//This function displays elements of the array
void displayArray(int arr[], int sz)
{
    printf("\n Array Elements are: ");
    for(int i = 0; i < sz; i++)
        printf("\t%d",arr[i]);
    printf("\n");
}
```

```
    void insertionSort(int arr[], int sz)
    {
        int i, j, key;
        for(i=1; i < sz; i++)
        {
                key = arr[i];
                j = i-1;
                while(j >= 0 && key < arr[j])
                { //move items in array so that there
                is the right space for 'key' item.
                        arr[j+1] = arr[j];
                        j--;
                }
                arr[j+1] = key;
        }
    }
```

//

```
/*Implementation of Quick Sort
 * quickSort.c
 *
 * Copyright 2020 gobinda <gobinda@GOBINDA-
PC>
 *
 */
#include <stdio.h>
#include <stdlib.h>
//function prototypes
void fillArray(int [], int);
void displayArray(int [], int);
void insertionSort(int [], int);
void swap(int *, int *);
void QUICKSORT(int [], int, int);
int partition(int [], int, int);

int main(int argc, char **argv)
{
        int *arr;
        int size;
        printf(" How many numbers to store in
array  ");
        scanf("%d", &size);
        arr = malloc(size*sizeof(int));//address of
block of memory defined by size*int is assigned to
pointer arr.
        printf("\n Filling Array Now");
        fillArray(arr, size);
        printf("\n Unsorted array\n");
        displayArray(arr, size);
        QUICKSORT(arr, 0, size-1);
        printf("\n\n Sorted Array\n");
        displayArray(arr, size);
        return 0;
}
//This function fills an array with random numbers
void fillArray(int arr[], int sz)
{
        printf("\n Enter random values to fill an
array: ");
        for(int i = 0; i < sz; i++)
                scanf("%d",&arr[i]);

}

//This function displays elements of the array
void displayArray(int arr[], int sz)
{
        printf("\n Array Elements are: ");
        for(int i = 0; i < sz; i++)
                printf("\t%d",arr[i]);
        printf("\n");
}

//This function swaps two values
void swap(int *a, int *b)
{
        int temp;
                temp = *a;
                *a = *b;
                *b = temp;
}

void QUICKSORT(int ARR[], int lo, int hi)
{
        if (lo < hi)
        {

           int INDEX = partition(ARR, lo, hi);
           //printf("\n**I am here**\n");
           QUICKSORT(ARR, lo, INDEX-1);
           QUICKSORT(ARR, INDEX+1, hi);
        }
```

```
}

int partition(int ARR[], int left, int right)
{
        int x = left ;
        int y = right ;
        int pivot = ARR[left];
        while(x < y)
        {
            do{//looks for larger item than pivot from left to right
                        x++;
            }while(ARR[x] < pivot);
            while(ARR[y] > pivot) //looks for smaller item than pivot from right to left
                        y--;
            if(x < y)//if larger item is at left than is switched with smaller item at the right
                swap(&ARR[x], &ARR[y]);
        }
        ARR[left] = ARR[y];
        ARR[y] = pivot;          //set pivot at the sorted place.
        return y;    //return position of pivot
}
```

****

MERGE SORT

Mergesort uses the divide-and-conquer technique to sort a list. Mergesort also partitions the list into two sublists, sorts the sublists, and then combines the sorted sublists into one sorted list.

Merge sort algorithm focuses on two main concepts to improve its performance (running time):

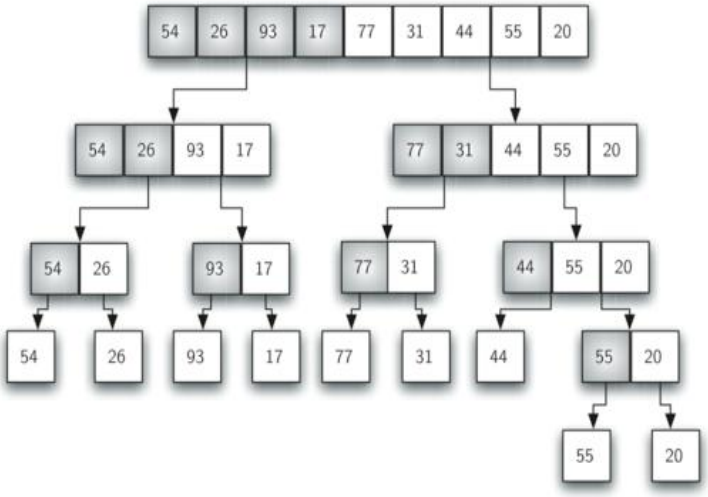
- A smaller list takes fewer steps and thus less time to sort than a large list.
- As number of steps is relatively less, thus less time is needed to create a sorted list from two sorted lists rather than creating it using two unsorted lists.

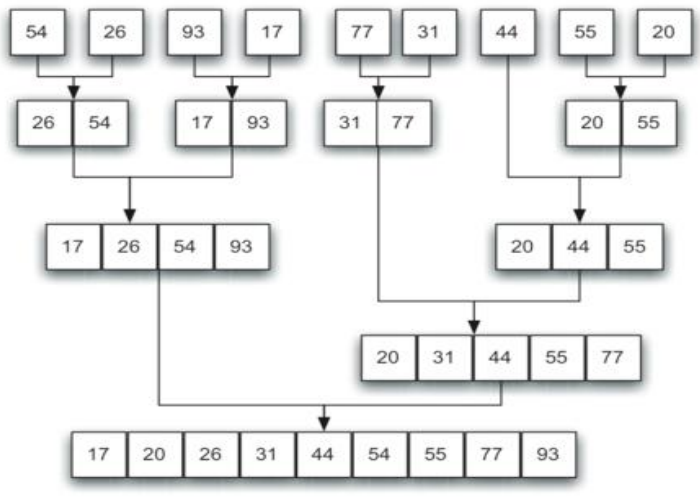
The basic steps of a merge sort algorithm are as follows:

- If the array is of length 0 or 1, then it is already sorted.
- Otherwise, divide the unsorted array into two sub-arrays of about half the size.
- Use merge sort algorithm recursively to sort each sub-array.
- Merge the two sub-arrays to form a single sorted list.

Mergesort and quicksort differ in how they partition the list. As discussed earlier, quicksort first selects an element in the list, called pivot, and then partitions the list so that the elements in one sublist are less than pivot and the elements in the other sublist are greater than or equal to pivot. By contrast, mergesort divides the list into two sublists of nearly equal size.

To understand the merge sort algorithm, consider the figure below which shows how we merge two lists to form one list.





Algorithms

```
MERGE_SORT(ARR, BEG, END)
Step 1: START
Step 2: IF BEG < END
          SET MID = (BEG + END)/2
          MERGE_SORT(ARR, BEG, MID)
          MERGE_SORT(ARR, MID+1, END)
          MERGE (ARR, BEG, MID, END)
        [END OF IF]
Step 3: END
```

```
MERGE (ARR, BEG, MID, END)
Step 1: [INITIALIZE] SET I = BEG, J = MID + 1, INDEX = 0
Step 2: Repeat while (I <= MID) AND (J<=END)
           IF ARR[I] < ARR[J]
                SET TEMP[INDEX] = ARR[I]
                SET I = I + 1
           ELSE
                SET TEMP[INDEX] = ARR[J]
                SET J = J + 1
          [END OF IF]
          SET INDEX = INDEX + 1
        [END OF LOOP]
Step 3: [Copy the remaining elements of right sub-array, if
any]
        IF I > MID
          Repeat while J <= END
                SET TEMP[INDEX] = ARR[J]
                SET INDEX = INDEX + 1, SET J = J + 1
                [END OF LOOP]
        [Copy the remaining elements of left sub-array, if any]
        ELSE
          Repeat while I <= MID
                SET TEMP[INDEX] = ARR[I]
                SET INDEX = INDEX + 1, SET I = I + 1
          [END OF LOOP]
        [END OF IF]
Step 4: [Copy the contents of TEMP back to ARR]
 SET K= 0
Step 5: Repeat while K < INDEX
                SET ARR[K] = TEMP[K]
                SET K = K + 1
        [END OF LOOP]
Step 6: END
```

****

```
/*Implementation of Merge Sort
 * mergesort.c
 *
 * Copyright 2020 gobinda <gobinda@GOBINDA-PC>
 *
 */
#include<stdio.h>
#define max 20

//function prototypes
void fillArray(int [], int);
void display(int [], int);
void merge(int [], int, int, int);
void merge_sort(int [], int, int);

int main()
{
   int arr[max];
   int n;
   printf("How many integers to input in array? ");
   scanf("%d",&n);
   fillArray(arr, n);
   printf("\n\n***Array Before Sorting***\n");
   display(arr, n);
   merge_sort(arr, 0, n-1);
   printf("\n\n***Array After Sorting***\n");
   display(arr, n);
   printf("\n");

   return 0;
}

//This function fills array with array elements
void fillArray(int myarr[], int sz)
{
   printf("\nInput Array Elements\n");
   for(int i = 0; i < sz; i++)
   {
    scanf("%d",&myarr[i]);
   }
```

```
}

//This function displays the array elements
void display(int myarr[], int sz)
{
   printf("Array Elements are: ");
   for(int i = 0; i < sz; i++)
   {
      printf("\t%d",myarr[i]);
   }
}

//This function divides the array into sub-array at the mid of
the original array
void merge_sort(int myarr[], int BEG, int END)
{
   if (BEG < END)
   {
      int MID = (BEG + END)/2;
      merge_sort(myarr, BEG, MID);
      merge_sort(myarr, MID+1, END);
      merge(myarr, BEG, MID, END);
   }
}

//this function conquers the divided array
void merge(int myarr[], int BEG, int MID, int END)
{
  int I = BEG, J = MID + 1, INDEX = BEG;
  int TEMP[max];
   while(I <= MID && J <= END)
   {
     if(myarr[I] < myarr[J])
     {//fill temp array with elements in ascending order
       TEMP[INDEX] = myarr[I];
       I++;
     }
     else
     {
       TEMP[INDEX] = myarr[J];
       J++;
     }
     INDEX++;
   }
//Copy the remaining elements of right sub-array, if any
if(I > MID)
{
   while (J <= END)
    {
      TEMP[INDEX] = myarr[J];
      INDEX++;
      J++;
    }
}
//Copy the remaining elements of left sub-array, if any
else
{
  while (I <= MID)
  {
      TEMP[INDEX] = myarr[I];
      INDEX++;
      I++;
  }
}

//Copy the contents of TEMP back to MYARR
int K = BEG;
while(K < INDEX)
{
  myarr[K] = TEMP[K];
  K++;
}
}
```

****

TREE SORT

A tree sort is a sorting algorithm that sorts values by making use of the properties of binary search tree (Which we have studied in earlier chapter). The algorithm first builds a binary search tree using the numbers to be sorted and then does an in-order traversal so that the numbers are retrieved in a sorted order.

HEAPSORT:

In this section we describe one of these, known as **heapsort**, which is a selection sort. It was discovered by John Williams in 1964 and uses a new data structure called a **heap** to organize the list elements in such a way that the selection can be made efficiently.

HEAPS:

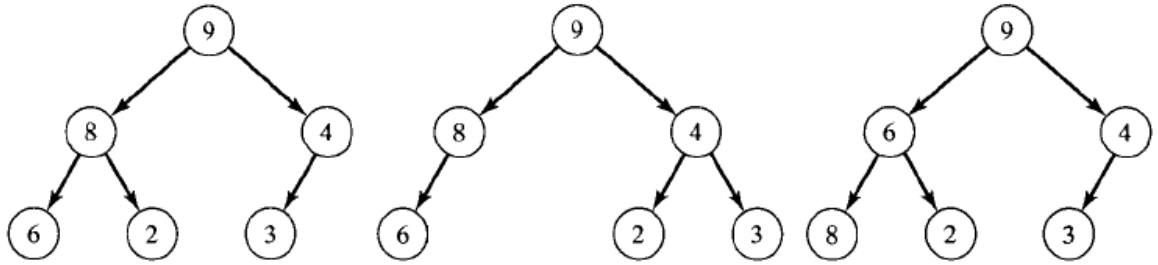
A heap is a binary tree that satisfied the following properties:-

- Shape property
- Order property.

By the ***shape property*** we mean that heap must be a complete binary tree whereas by ***order property*** we mean that for every node in the heap the value stored in the heap node is greater than or equal to the value at each of its' children. A heap that satisfied this property is known as ***max heap.***

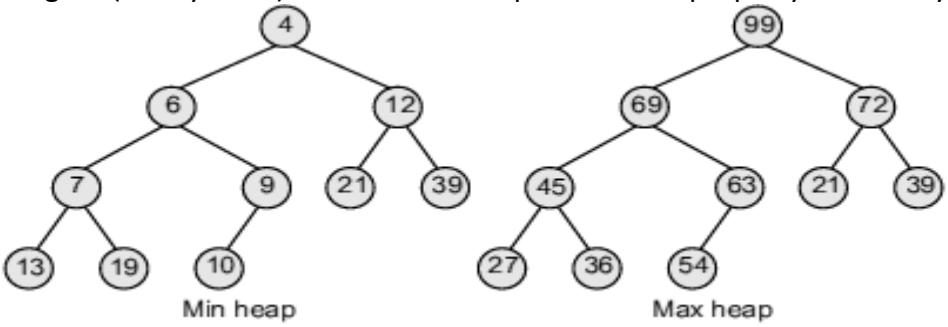
However, If the order property is such that for every node in the heap the value stored in that node is less than or equal to the value in each of its children. That heap is known as ***min heap.***

For example, determine which of the following binary trees are heaps:



The first binary tree is a heap; the second binary tree is not, because it is not complete; the third binary tree is complete, but it is not a heap because the heap-order condition is not satisfied.

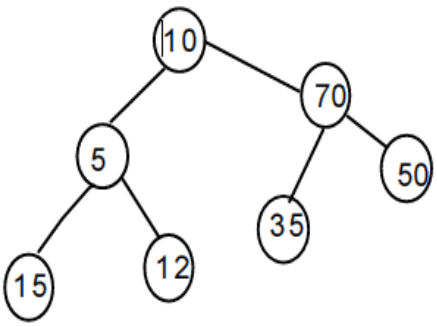
The figure (binary trees) below shows shape and order property of a binary heap.



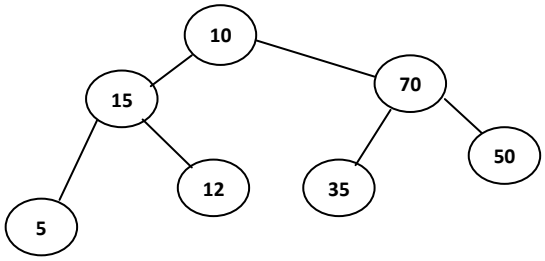
Now, let's create a heap (max heap) tree with given data below.

10 5 70 15 12 35 50

Here, created binary tree satisfies the complete binary property, however max heap order is not satisfied yet. To make it so, we need to switch some values at the nodes so that max heap order could be maintained. This is known as ***Heapify***.



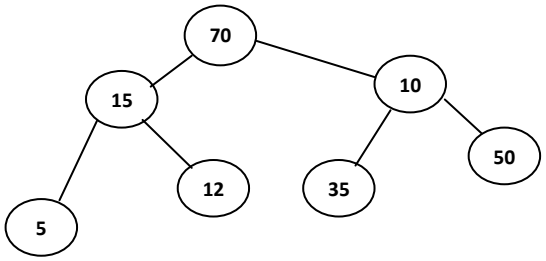
The above tree is not heap order. So, we need to make it max-heap by switching values at nodes so that max-heap order could be maintained. We need to make max-heap order at every sub-tree of the given tree.



Step 1:

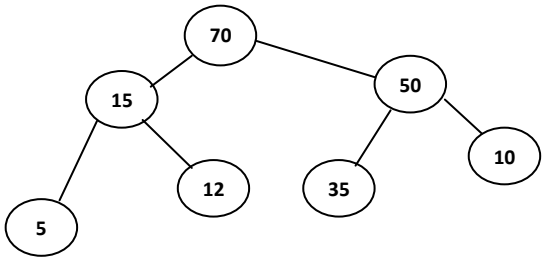
So, let's take the sub-tree with node having value 5 as the root node. And since 5 < 15 and 15 > 12; we switch nodes with 5 and 15 and this here is the tree we get after switching.

NOTE: The process of moving smaller values to the nodes at the lower level of the tree is called ***percolate down.*** *And reverse is* ***percolate up.***



Step 2:

Root node has value 10 which is less than its child node 15. But 15 < 70 (another child of 10), we switch nodes with values 10 and 70. The resultant tree is shown here, which is still not the max-heap order yet.



Step 3:

The right sub-tree is still need to be heapified because 10 is less than 35 and 50 (both child nodes). But, 50 > 35, we switch nodes values 10 and 50. The tree obtained here is max-heap order because all the values at root nodes have values greater than or equal to its child nodes' values.

Heap tree Representation:

To implement a heap, we need to store its elements in an array. The i^{th} node data is stored in the i^{th} location of the array. The completeness property of a heap guarantees that these data items will be stored in consecutive locations at the beginning of the array.

Note that in such an array-based implementation, it is easy to find the children of a given node: The children of the i^{th} node are at locations 2*i + 1 and 2*i + 2 (for i >= 0). Similarly, the parent of the i^{th} node is easily seen to be in location i / 2.

Heapsort

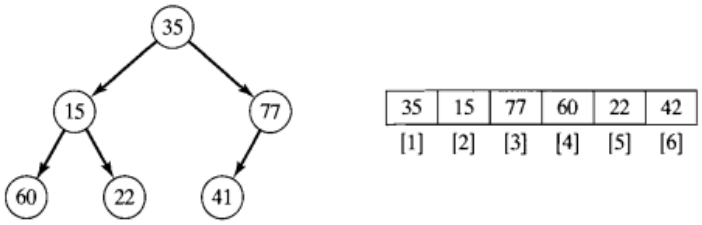
Now that we know about heaps, we can show how they can be used to develop an efficient selection sort algorithm. Given an array ARR with n elements, the heap sort algorithm can be used to sort ARR in two phases:

- In phase 1, build a heap H using the elements of ARR.
- In phase 2, repeatedly delete the root element of the heap formed in phase 1.

In a max heap, we know that the largest value in H is always present at the root node. So in phase 2, when the root element is deleted, we are actually collecting the elements of ARR in decreasing order.

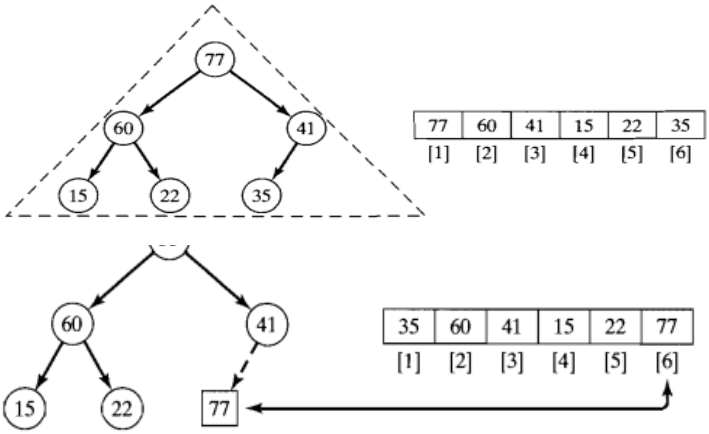
To illustrate, suppose the following list is to be sorted:
35, 15, 77, 60, 22, 41
We think of the array storing these items as a complete binary tree:

First, we must convert this tree to a heap. The given complete binary tree goes through this "heapify" process, with the subtree that is heapified at each stage and final max-heap tree is given below.

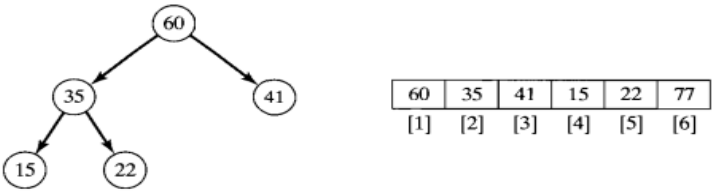


[For simplicity array here is numbered starting from 1.]

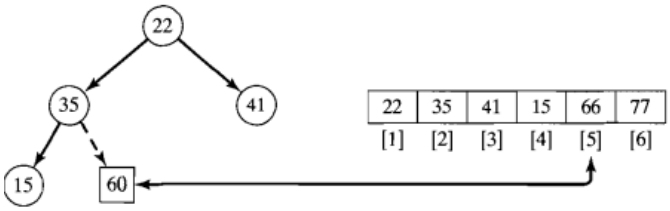
This puts the largest element in the list at the root of the tree-that is, at position 1 of the array. We now use the strategy of a selection sort and correctly position this largest element by swapping it with the element at the end of the list and turn our attention to sorting the sublist consisting of the first five elements. In terms of the tree, we are exchanging the root element and the rightmost leaf element and then "pruning" this leaf from the tree, as indicated by the dotted arrow in the following diagram:



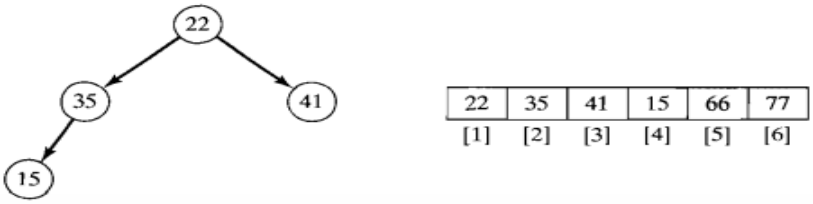
Quite obviously, the tree that results when we perform this root-leaf exchange followed by pruning the leaf is not usually a heap. In our example, the five-node tree that corresponds to the sublist 35, 60,41,15,22 is not a heap. Thus heapify algorithm to convert this tree to a heap:



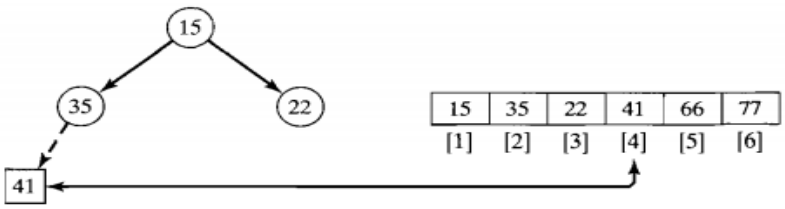
Again we swap the root with the rightmost leaf to correctly position the second largest element in the list and then prune this leaf from the tree to prepare for the next stage:



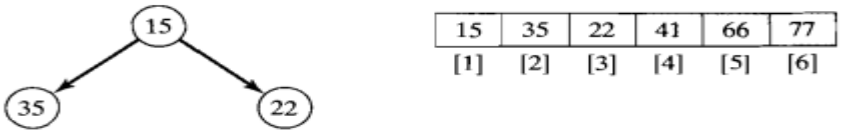
Now we use heapify to convert to a heap the tree corresponding to the sublist consisting of the first four elements,



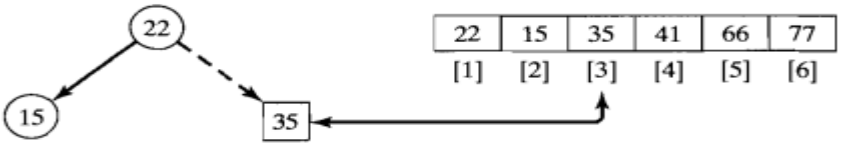
and do the root-leaf exchange and the leaf pruning to correctly position the third largest element in the list:



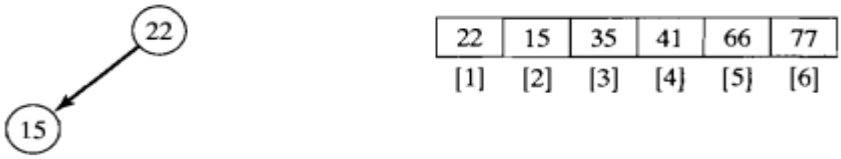
Next, the three-node tree corresponding to the sublist 15, 35, 22 is converted to a heap,



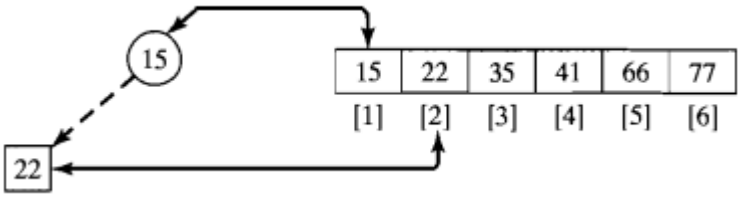
and the roof-leaf exchange and pruning operations are used to correctly position the next largest element in the list:



Finally, the two-node tree corresponding to the two-element sublist 22, 15 is converted to a heap,



and one last root-leaf swap and leaf pruning are performed to correctly position the element 22, which obviously also correctly positions the smallest element 15 at the beginning of the list:



Now, heap sort algorithm can be divided into two parts. Suppose A is an array with N elements from A[1] to A[N].

1. In the first step, a max-heap is built from the N data items. As a result, the root A[1] is to be the largest element.
2. In the second step,
   - The interchange between A[1] and A[N]. Trivially, the largest element is placed in the N position of the array at the end of pass 1. Decreasing the heap size by one and shifting down the new first value A[1] into its proper position to rebuild the heap.
   - The interchange between A[1] and A[N-1]. The second largest element is placed in the N-1 position of the array at the end of pass 2. Decreasing the heap size by one and shifting down the new first value A[1] into its proper position to rebuild the heap.
   - The interchange between A[1] and A[N-2]. The third largest element is placed in the N-2 position of the array at the end of pass 3. Decreasing the heap size by one and shifting down the new first value A[1] into its proper position to rebuild the heap.
   - Finally, the interchange between A[1] and A[2]. Decreasing the heap size by one and trivially, the smallest element is in A [1]. Therefore, the array of N elements from A[1] to A[N] is sorted after N-1 passes.

Algorithm of Heap Sort

```
Algorithm: HEAP_SORT(A, N)
[A is an array of N elements]
1. Repeat For I = N/2 to 1
   Call SHIFT_DOWN(A, I, N)
   End of Loop]
2. Repeat For I = N TO 2
      a. Temp = A[I]
      b. A[I] = A[1]
      c. A[1] = Temp
      d. Call SHIFT_DOWN(A, 1, I-1)
   [End of Loop]
3. Return

Function: SHIFT_DOWN(A, K, N)
[A is an array of N elements, K is the position of the element that shifted down]

   1. Set Parent = K, Child = 2* Parent and Temp = A[Parent]
   2. Repeat while Child<=N
      a. If Child < N and A[Child +1] > A[Child]
         then Set Child = Child + 1
      b. If Temp >= A[Child] Goto Step 3
      c. Set A[Parent] = A[Child]
      d. Set Parent = Child and Child = 2* Parent
     [End of Loop]
   3. Set A[Parent] = Temp
   4. Return
```

RADIX SORT

[The term radix is another term used in mathematics for the base of a number system.]

All the sorts we have considered thus far have a common pattern: comparing elements in the list, and moving them according to some specific rules. **Radix sort** avoids comparison by creating and distributing elements into buckets according to their radix. In that sense **Radix sort** is different. It is a linear sorting algorithm for integers and uses the concept of sorting names in alphabetical order.

There are several versions of **radix sort**, but their common characteristic is that they are based on examining the digits in some base-b numeric representation of the items (or numeric keys of records).

One simple version is sometimes called ***a least-significant-digit radix sort***, because it processes the digits from right to left. The sorting procedure proceeds by sorting the least significant to the most significant digit. While sorting the numbers, we have ten buckets, each for one digit (0, 1, 2, ..., 9) and the number of passes will depend on the length of the number having maximum number of digits.

This is the basic idea of radix sort. To illustrate it, suppose the numbers we are sorting are expressed in base ten and have at most 3 digits, for example,

64, 8, 216, 512, 27, 729, 199, 550, 343, 125, 93, 666

To help with understanding radix sort, we will write them all with three digits, padding with Os to the left where necessary:

064, 008, 216, 512, 027, 729, 199, 550, 343, 125, 093, 666

We first distribute them into 10 bins labeled 0, 1, ..., 9, according to their rightmost digit:

550	011	512	093 343	064	125	666 216	027	008	199 729
0	1	2	3	4	5	6	7	8	9

Now we collect them together from left to right, bottom to top,

550, 011, 512, 343, 093, 064, 125, 216, 666, 027, 008, 729, 199

and then distribute them again, this time using the second digit:

008	216 512 011	729 027 125		343	550	666 064			199 093
0	1	2	3	4	5	6	7	8	9

Collecting them together produces

008, 011, 512, 216, 125, 027, 729, 343, 550, 064, 666, 093, 199

Now we distribute them one last time, using the leftmost digit:

093 064 027 011 008	199 125	216	343		550 512	666	729		
0	1	2	3	4	5	6	7	8	9

This time, collecting them produces the sorted list:

008, 011, 027, 064, 093, 125, 199, 216, 343, 512, 550, 666, 729

Algorithm for RadixSort (ARR, N)

```
Step 1: Find the largest number in ARR as LARGE
Step 2: [INITIALIZE] SET NOP = Number of digits in LARGE
Step 3: SET PASS = 0
Step 4: Repeat Step 5 while PASS <= NOP-1
Step 5:          SET I= 0 and INITIALIZE buckets
Step 6:          Repeat Steps 7 to 9 while I < N-1
Step 7:                SET DIGIT = digit at PASSth place in A[I]
Step 8:                Add A[I] to the bucket numbered DIGIT
Step 9:                INCEREMENT bucket count for bucket numbered DIGIT
            [END OF LOOP]
Step10:          Collect the numbers in the bucket
      [END OF LOOP]
Step 11: END
```

Complexity of Radix Sort

Assume that there are `n` numbers that have to be sorted and `k` is the number of digits in the largest number. In this case, the radix sort algorithm is called a total of `k` times. The inner loop is executed `n` times. Hence, the entire radix sort algorithm takes `O(kn)` time to execute. When radix sort is applied on a data set of finite size (very small set of numbers), then the algorithm runs in `O(n)` asymptotic time.

SHELL SORT

Shell sort, invented by Donald Shell in 1959, is a sorting algorithm that is a generalization of insertion sort.

Shell sort is considered an improvement over insertion sort as it compares elements separated by a gap of several positions. This enables the element to take bigger steps towards its expected position. The difficulty for insertion sort is that an element moves towards its final position very slowly, one position at a time and it cannot make a long jump quickly to reach its destination.

In Shell sort, elements are sorted in multiple passes and in each pass; data are taken with smaller and smaller gap sizes. However, the last step of Shell sort is a plain insertion sort. But by the time we reach the last step, the elements are already 'almost sorted', and hence it provides good performance.

In this sorting algorithm, we first create a number of logical independent sub- arrays as shown in the figure below.

h number of independent sub-arrays

A[0]	A[h]	A[2h]	A[3h]	.....
A[1]	A[h+1]	A[2h+1]	A[3h+1]	.....
A[2]	A[[h+2]	A[2h+2]	A[3h+2]	.....
.	.	.	.	
.	.	.	.	
A[h-1]	A[2h-1]	A[3h-1]	A[4h-1]	.....

Sort the sub-array using insertion sorting technique. The difference between two consecutive elements of a sub-array is ***h***. In general, an array is to be sorted for several values of ***h***, which forms a monotone decreasing sequence and the last value of ***h*** sequence must be always one. The last pass is same as an ordinary insertion sort. More specifically, a sequence of numbers is

$h_t > h_{t-1} > h_{t-2} > \ldots > h_2 > h_1 = 1$

to be decided and the array is to be sorted over t different passes using the method. The general idea in insertion sort, the predecessor of a [i] is always a [i-1], but in the shell sort predecessor of a [i] is a [i-h] for some value of ***h***.

Example: Suppose A is an array with 12 elements from A[1] to A[12]. Array elements are: 60, 80, 20, 50, 10, 15, 95, 90, 45, 70, 25 and 30

	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]
Input Data	**60**	**80**	**20**	**50**	**10**	**15**	**95**	**90**	**45**	**70**	**25**	**30**
After 5-sorting	**15**	**30**	**20**	**45**	**10**	**25**	**80**	**90**	**50**	**70**	**60**	**95**
After 3-sorting	15	10	20	45	30	25	70	60	50	80	90	95
After 1-sorting	10	15	20	25	30	45	50	60	70	80	90	95

- ✓ At the first pass, 5-sorting (h = 5) performs insertion sort on the separate sub-arrays (A[1], A[6], A[11]), (A[2], A[7], A[12]), (A[3], A[8]), (A[4], A[9]) and (A[5], A[10]).
- ✓ In the second pass, 3-sorting (h = 3) performs insertion sort on the separate sub-arrays (A[1], A[4], A[7], A[10]), (A[2], A[5], A[8], A[11]) and (A[3], A[6], A[9], A[12]).
- ✓ In the third and final pass, 1 sorting (h = 1), is an ordinary insertion sort of the entire array (A[1], A[2], ...A[12]).

```
Algorithm: SHELL_SORT(A, N)
[A is an array of N elements]
1. Set GAP = LAST_GAP_SEQUENCE
2. Repeat steps 3 to 8 while GAP >= 1
3. Set I = GAP
4. Repeat steps 5 to 7 For I = GAP to N - 1
5.       Set Temp = A[I]
6.       Repeat while J >= GAP and A[J-GAP] > Temp
            a) Set A[J] = A[J-GAP]
            b) Set J = J - GAP
      [End of loop]
7. Set A[J+GAP] = Temp
[End of loop]
8. Set GAP = PREV_GAP
[End of Loop]
9. Return
```

GAP SEQUENCES:

There are different gap sequences; each of them gives correct sort. However, the properties of thus obtained versions of Shell sort may be different. Some of the gap sequences are given in the following table.

Gap Sequence	General Term (k≥1)	Worst-case time complexity
1, 4, 13, 40, 121, …	$(3^k-1)/2$ less than $\lceil n/3 \rceil$	$O(n^{3/2})$
1, 3, 5, 9, 17, 33, ….	2^k+1, prefixed with 1	$O(n^{3/2})$
1, 3, 7, 15, 31, ….	2^k-1	$O(n^{3/2})$
1, 2, 3, 4, 6, 8, 9, 12…	2^p3^q	$O(n\log^2 n)$

Time complexity of Shell Sort

The worst-case time complexity of the shell sort depends on the gap sequence. Different gap sequences may generate different worst time complexity. Therefore, neither tight upper bounds on time complexity nor the best increment sequence are known.

Shell Sort algorithm	Best Case	Average Case	Worst Case
	$O(n\log^2 n)$	$O(n\log n)$	$O(n^{3/2})$