Symbiosis Institute of Technology

Advanced Algorithms Lab Manual

Department of Computer Science Engineering

Lab Manual for Academic Session 2022-23

Programme Name: B.Tech Third Year CSE & IT (Generic Elective III)

Semester: VI

Lab Course Details

Programme Name:- UG (Third Year B.Tech)

Course Name – Advanced Algorithms Lab

Semester -VI

Credit Points - 01

Practical Hrs/Week - 02

Term work - 25 marks

Sr. No.	Component	CO	Max marks	Weight	Tentative date
1	Lab submission	CO1,CO2,CO3,CO4,CO5, CO6,CO7,CO8,CO9,CO10	10	40%	Throughout the semester
2	Lab performance	CO1,CO2,CO3,CO4,CO5, CO6,CO7,CO8,CO9,CO10	10	40%	Lab exam week of the semester
3	Lab Viva	CO1,CO2,CO3,CO4,CO5, CO6,CO7,CO8,CO9,CO10	5	20%	Lab exam week of the semester

Assignment List

Assignment No.	Assignment Name
1	Analysis, Proof of analysis and Implementation of Insertion Sort Algorithm
2	Analysis, Proof of analysis and Implementation of Merge Sort Algorithm
3	Analysis, Proof of analysis and Implementation of Quick Sort Algorithm
4	Analysis, Proof of analysis and Implementation of Heap Sort Algorithm
5	Analysis, Proof of analysis and Implementation of BFS and DFS graph traversal Algorithms
6	Analysis, Proof of analysis and Implementation of Prim's and Kruskal's MST Algorithm
7	Analysis, Proof of analysis and Implementation of Dijkstra's shortest path Algorithm
8	Parallel Algorithm Analysis

ADVANCED ALGORITHM LAB

Title - Insertion Sort - Analysis and Implementation

INSERTION SORT-

1. WORKING MECHANISM

- Compare the element to the element next to it.
- If we can find a point in the sorted array where the element may be added at each comparison, we can free up space by shifting the items to the right and inserting the element there.
- Repeat the steps above until the last element of the unsorted array is in its proper location.

EXAMPLE

13	32	26	9	33	18

Initially, the first two elements are compared.

13	32	26	9	33	18

32 > 13, therefore elements are already in ascending order. So, 13 is stored in a sorted sub-array.

13	32	26	9	33	18

Compare next 2 elements.

13	32	26	9	33	18

13	32	26	9	33	18

26 < 32. So, swap them

Now, swap 32 with 26. Along with swapping, insertion sort also checks it with all elements in the sorted array.

For now, the sorted array has only one element, i.e. 13. So, 26 > 13. Hence, the sorted array remains sorted after swapping.

13	26	32	9	33	18

13	26	32	9	33	18

31 and 8 are not sorted. So, swap them.

13	26	9	32	33	18

25 and 8 are unsorted.

13	26	9	32	33	18

Swap

13	9	26	32	33	18

13 and 9 are unsorted.

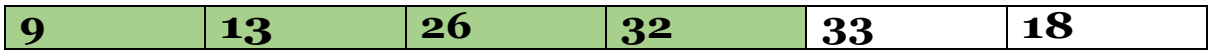
9	13	26	32	33	18

Swap them.

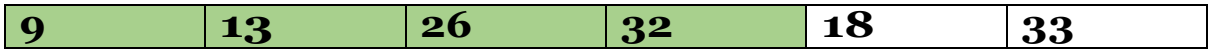
9	13	26	32	33	18

Sorted array has three items that are 9,13 and 26. Next we have 32 and 33.

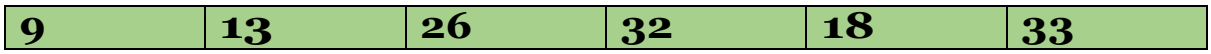
They are already sorted. Now, the sorted array includes 9, 13, 26 and 32.



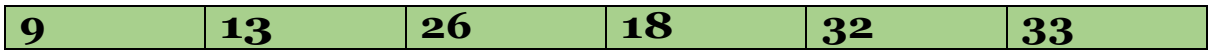
Next elements that are 33 and 18.



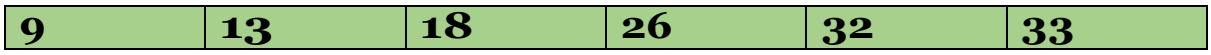
18 < 33. So, swap.



Swapping makes 32 and 18 unsorted. So, swap them too.



Swapping makes 26 and 18 unsorted. So, swap again.



Now the array is sorted.

2. PSEUDO CODE

ITERATIVE-

```
1.  for i = 0 to n
2.      key = A[i]
3.      j = i - 1
4.      while j >= 0 and A[j] > key
5.          A[j + 1] = A[j]
6.          j = j - 1
7.      end while
8.  A[j + 1] = key
9.  end for
```

RECURSIVE-

1. recursive_insertion_sort(A, n)
2. IF n > 1 THEN
    a. recursive_insertion_sort(A, n-1)
    b. key = A[n]
    c. i = n-1
    d. DOWHILE A[i] > key AND i > 0
        i. A[i+1] = A[i]
        ii. i =i-1
    e. ENDDO
    f. A[i+1] = temp
3. ENDIF
4. END

3. PROGRAM CODE SNIPPET (both iterative and recursive)

insertion_sort.cpp

```cpp
#include <iostream>
using namespace std;
#define MAX_SIZE 20;

int i, size;
int a[20];

void create_array()
{
    int data;
    cout << "Enter the size of array: \n";
    cin >> size;
    for (int i = 0; i < size; i++)
    {
       cout << "Enter the value: ";
       cin >> data;
       a[i] = data;
    }
}

void print_array()
{
    for (int i = 0; i < size; i++)
    {
      cout << a[i] << " ";
    }
}

void iterative_insertion(int arr[], int n)
{
    for (int i = 1; i < n; i++)
    {
      int value = arr[i];
      int j = i;
      while (j > 0 && arr[j - 1] > value)
      {
          arr[j] = arr[j - 1];
          j--;
      }
     arr[j] = value;
    }
}

```

```
44  void recursive_insertion(int arr[], int n)
45  {
46      if (n<=1)
47        return;
48      recursive_insertion (arr, n-1);
49     int value = arr[n-1];
50     int j = n-2;
51     while (j >=0 && arr[j] > value)
52     {
53       arr[j+1] = arr[j];
54       j--;
55     }
56     arr[j+1] = value;
57
58  }
59  int main()
60  {
61     int choice, ans;
62     do
63     {
64        cout << "----------INSERTION SORT-----------\n";
65        cout << "1] Iterative Method\n";
66        cout << "2] Recursive Method\n";
67        cout << "----------------------------------\n";
68        cout<<"\n\nEnter your choice:";
69        cin >> choice;
70        switch (choice)
71        {
72        case 1:
73            create_array();
74            cout << "\nArray before Insertion Sort: \n";
75            print_array();
76            iterative_insertion(a, size);
77            cout << "\nArray after Insertion Sort: \n";
78            print_array();
79           break;
80        case 2:
81            create_array();
82            cout << "\nArray before Insertion Sort: \n";
83            print_array();
84            recursive_insertion(a, size);
85            cout << "\nArray after Insertion Sort: \n";
```

```
80        case 2:
81            create_array();
82            cout << "\nArray before Insertion Sort: \n";
83            print_array();
84            recursive_insertion(a, size);
85            cout << "\nArray after Insertion Sort: \n";
86            print_array();
87            break;
88        default:
89            cout << "Enter a valid choice!!\n";
90            break;
91        }
92       cout << "\n\nDo you want to continue (Yes=1) :";
93       cin >> ans;
94     } while (ans == 1);
95  }
96
```

4. O/P SNIPPET (both iterative and recursive)

```
G:\SEM6\AA LAB\codes\insertion_sort.exe
----------INSERTION SORT-----------
1] Iterative Method
2] Recursive Method
-----------------------------------


Enter your choice:1
Enter the size of array:
5
Enter the value: 5
Enter the value: 4
Enter the value: 3
Enter the value: 2
Enter the value: 1

Array before Insertion Sort:
5 4 3 2 1
Array after Insertion Sort:
1 2 3 4 5

Do you want to continue (Yes=1) :1
----------INSERTION SORT-----------
1] Iterative Method
2] Recursive Method
-----------------------------------


Enter your choice:2
Enter the size of array:
5
Enter the value: 3
Enter the value: 4
Enter the value: 5
Enter the value: 2
Enter the value: 1

Array before Insertion Sort:
3 4 5 2 1
Array after Insertion Sort:
1 2 3 4 5

Do you want to continue (Yes=1) :0

--------------------------------
Process exited after 24.7 seconds with return value 0
Press any key to continue . . .
```

5. ANALYSIS AND ITS PROOF –

NO OF TIMES EXECUTION OF EACH LINE

a. RUNNING TIME:

The amount of operations performed determines the algorithm's Running Time. We can see from the Pseudocode that this algorithm has exactly seven operations. So, our objective is to calculate the Cost or Time Complexity of each, and the total Time Complexity of our Algorithm will be the sum of them.

We calculate the number of times each i operation is conducted by assuming the cost of each i operation is C i where i is 1,2,3,4,5,6,8. As a result, the total cost of one such procedure is the product of the cost of one operation multiplied by the number of times it is carried out.

COST OF LINE	NO. OF TIMES IT IS RUN
C_1	n
C_2	n - 1
C_3	n - 1
C_4	$\Sigma^{n-1}_{j=1}(t_j)$
C_5	$\Sigma^{n-1}_{j=1}(t_j - 1)$
C_6	$\Sigma^{n-1}_{j=1}(t_j - 1)$
C_8	n - 1

Then Total Running Time of Insertion sort (T(n)) = C1 * n + (C2 + C3) * (n -1) + C4 * Σ n - 1j = 1(t j) + (C5 + C6) * Σ n - 1j = 1(t j) + C8 * (n - 1)

b. MEMORY USAGE

We didn't need any additional space. We're only rearranging the input array to get the result we want. As a result, we can state that no additional RAM is required to perform this Algorithm. Although each of these operations will be put to the stack, they will not be done at the same time, the Memory Complexity will be O. (1)

COMPLEXITY(ITERATIVE)

- Worst Case Time Complexity = O(n^2)
- Average Case Time Complexity = O(n^2)
- Best Case Time Complexity = O(n)
- Space Complexity = O (1)

COMPLEXITY(RECURSIVE)

- Worst Case Time Complexity = O(n^2)
- Average Case Time Complexity = O(n^2)
- Best Case Time Complexity = O(n)
- Space Complexity = O (1)

6. TIME COMPLEXITY

A. BEST CASE ANALYSIS

In Best Case i.e., when the array is already sorted, tj = 1

Therefore,

T (n) = C1 * n + (C2 + C3) * (n - 1) + C4 * (n - 1) + (C5 + C6) * (n - 2) +C8 * (n - 1)

which when further simplified has dominating factor of n and gives

T(n) = C * (n) or O(n)

EXAMPLE-

Ascending order-

1	**2**	**3**	**4**	**5**

ELEMENT	COMPARISON	SWAPS
1	**0**	**0**
2	**1**	**0**
3	**1**	**0**
4	**1**	**0**
5	**1**	**0**

B. Average Case Analysis

Let's assume that tj = (j-1)/2 to calculate the average case

Therefore,

T (n) = C1 * n + (C2 + C3) * (n - 1) + C4/2 * (n - 1) (n) / 2 + (C5 + C6)/2 * ((n - 1) (n) / 2 - 1) + C8 * (n - 1)

which when further simplified has dominating factor of n2 and gives

T(n) = C * (n^2) or O (n^2)

Example-

2	**5**	**6**	**3**	**4**

ELEMENTS	COMPARISON	SwAPS
2	0	0
5	1	1
6	1	1
3	2	2
4	2	2

C. worst Case Analysis

In Worst Case i.e., when the array is reversely sorted (in descending order), tj = j

Therefore, T (n) = C1 * n + (C2 + C3) * (n - 1) + C4 * (n - 1) (n) / 2 + (C5 + C6) * ((n - 1) (n) / 2 - 1) + C8 * (n - 1)

which when further simplified has dominating factor of n2 and gives

T(n) = C * (n^2) or O (n^2)

EXAMPLE-

Descending order-

6	**5**	**4**	**3**	**2**	**1**

COMPARISON	SWAPS
0	**0**
1	**1**
2	**2**
3	**3**
.	**.**
.	**.**
(n-1)	**(n-1)**

Can be solved using n(n-1)/2

So, Worst case is O(n^2)

7. SPACE COMPLEXITY

- It is an in-place algorithm.
- It performs all computation in the original array and no other array is used.
- We didn't require any extra space. We are only re-arranging the input array to achieve the desired output. Hence, we can claim that there is no need of any auxiliary memory to run this Algorithm.
- Hence, the space complexity works out to be **O (1).**

8. UNIQUE CHARACTERISTICS

- It works well for smaller data sets but not so well for longer lists
- Insertion Sort is adaptive, which means that if given a partially sorted list, it reduces the overall number of steps and so improves performance.
- It has a lower spatial complexity. Insertion sort necessitates a single extra byte of memory.

9. REAL LIFE APPLICATIONS

- It can be used to sort short lists.
- It could be used to sort lists that are "nearly sorted."

- In Quick Sort, it could be used to sort smaller sub problems.
- When four players are playing a card game and each has 13 cards, they arrange the cards according to color, type, ascending or descending order, etc. It is entirely up to the players' preferences, but one thing is clear: they take a card from the 13 and place it in the correct position.

10. OPTIMIZATIONS -

- We may enhance the searching complexity by utilizing Binary Search, which reduces it from O(n) to O(log n) for one element and to n * O(log n) or O(n log n) for n elements.
  However, because one element takes O(n) time to place in its proper location, n elements will require n * O(n) or O(n^2) time to place in their proper locations. As a result, the overall complexity is still O. (n^2).

- We can enhance the swapping performance by utilizing a Doubly Linked List instead of an array. This reduces the swapping complexityfrom O(n) to O (1) because we can add an element to a linked list by changing pointers (without shifting the rest of elements).
  However, because we can't use binary search in linked lists, the search complexity remains O(n^2). As a result, the overall complexityis still O. (n^2).

  As a result, we can conclude that the worst-case time complexity of insertion sort cannot be reduced to O(n^2).

11. ADVANTAGES-

- Simple and easy to understand implementation
- Efficient for small data
- If the input list is sorted beforehand (partially) then insertions sort takes O(n+d) where d is the number of inversions
- Chosen over bubble sort and selection sort, although all have worst case time complexity as O(n^2)
- Maintains relative order of the input data in case of two equal values (stable)
- It requires only a constant amount O (1) of additional memory space(in-place Algorithm)

ADVANCED ALGORITHM LAB - 2

Title - Merge Sort - Analysis and Implementation

MERGE SORT-

Merge sort is a sorting algorithm based on the Divide and conquer strategy.

1. WORKING MECHANISM

- Divide the unsorted list into n sublists; each sublist has one element (which is sorted as it has one element)
- Merge two lists at a time. While merging compare elements for the two sublists and prepare a new sorted list. This takes linear time.
- Do step 2 for all sub-list pairs
- Repeat step 2 to 3 for the new list of sub-lists until we have only one list

EXAMPLE 1

22	41	35	18	42	27	50	52

According to the merge sort, first divide the given array into two equal halves. Merge sort keeps dividing the list into equal parts until it cannot be further divided.

As there are eight elements in the given array, so it is divided into two arrays of size 4.

DIVIDE

22	41	35	18

42	27	50	52

Again divide these two arrays into halves

.

DIVIDE

22	41

35	18

42	27

50	52

Again divide these arrays to get the atomic value that cannot be further divided.

DIVIDE

22

41

35

18

42

27

50

52

Now, merge them in the same manner they were broken.

First compare the element of each array and then combine them into another array in sorted order.

So, first compare 22 and 41, both are in sorted positions.
Then 35 and 18, and in the list of two values, put 18 first followed by 35.
Then 42 and 27, sort them and put 27 first followed by 42.
After that, compare 50 and 52, and place them sequentially.

MERGE

22	41

18	35

27	42

50	52

In the next iteration of combining, compare the arrays with two data values and merge them into an array of found values in sorted order.

MERGE

18	22	35	41

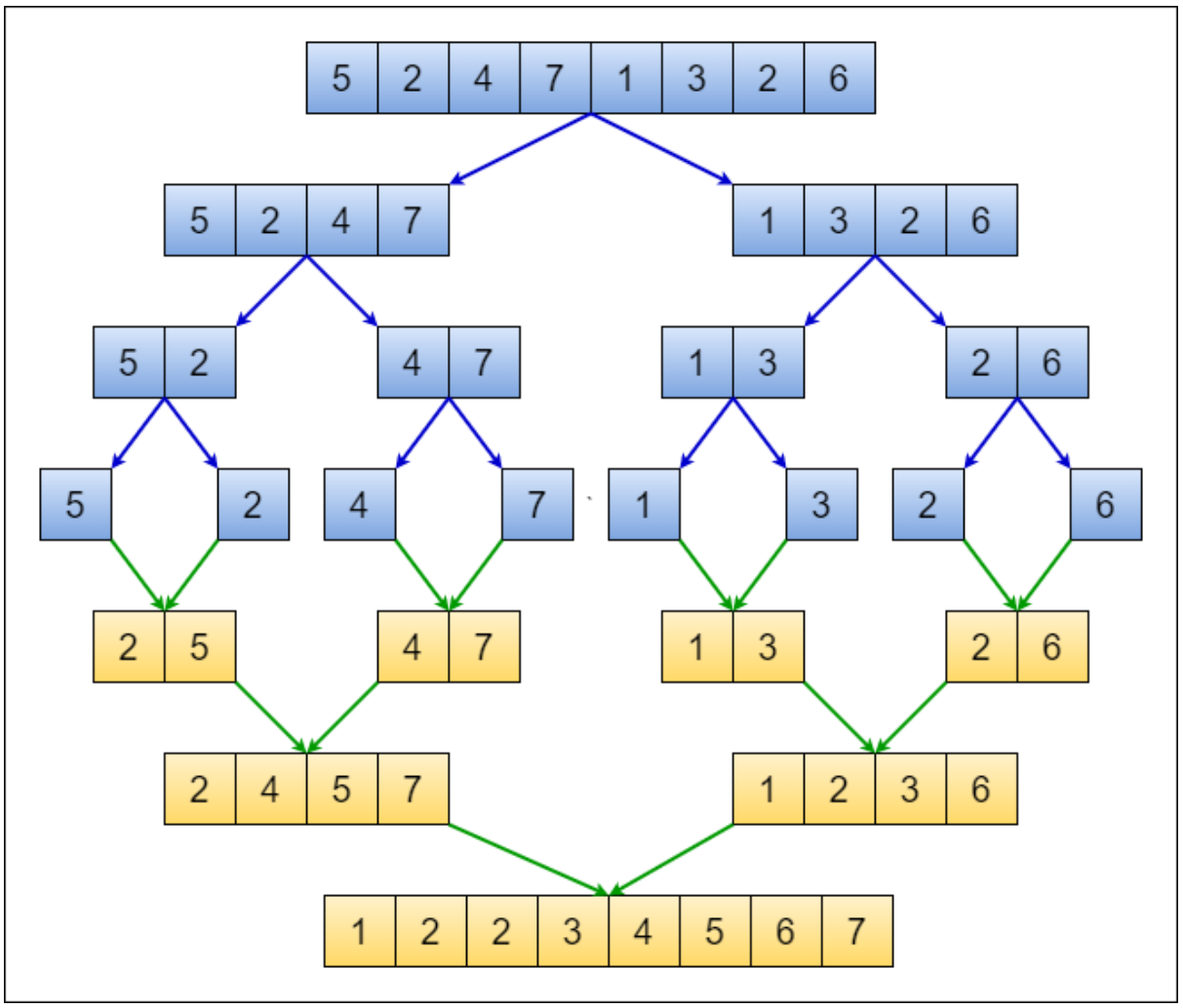
27	42	50	52

Now, there is a final merging of the arrays.

Sorted array-

18	22	27	35	41	42	50	52

EXAMPLE 2



2. PSEUDO CODE

RECURSIVE-

```
1. MERGE_SORT_REC(arr, beg, end)
2. {
3.  if beg < end
4.  set mid = (beg + end)/2
5.  MERGE_SORT(arr, beg, mid)
6.  MERGE_SORT(arr, mid + 1, end)
7.  MERGE (arr, beg, mid, end)
8.  end of if
9.  }
10. END MERGE_SORT
```

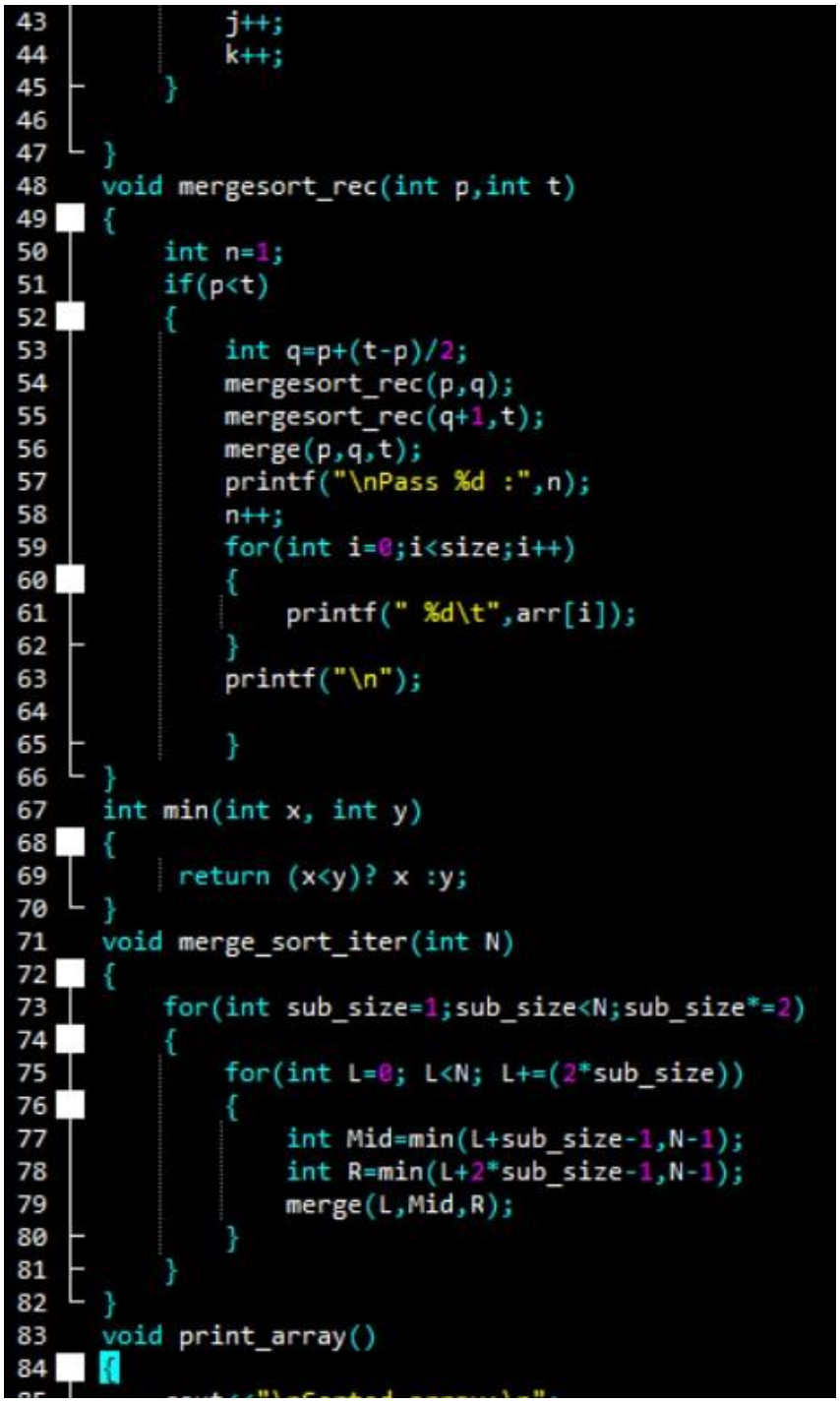
ITERATIVE-

```
1. void merge_sort_iter(int N)
2. {
3. for(int sub_size=1;sub_size<N;sub_size*=2)
4. {
5. for(int L=0; L<N; L+=(2*sub_size))
6. {
            int Mid=min(L+sub_size-1,N-1);
            int  R=min(L+2*sub_size-1,N-1);
            merge(L,Mid,R);
7.  }
8.  }
9.  }
```

3. PROGRAM CODE SNIPPET (both iterative and recursive)

merge sort.cpp

```cpp
#include <iostream>
using namespace std;
int arr[100];
int size;
void merge(int p,int q,int t)
{
    int i,j,k;
    int n1=q-p+1;
    int n2=t-q;
    int arr1[n1],arr2[n2];
    for(i=0;i<n1;i++)
    {
        arr1[i]=arr[p+i];
    }
    for(j=0;j<n2;j++)
    {
        arr2[j]=arr[q+1+j];
    }
    i=0;j=0;k=p;
    while(i<n1 && j<n2)
    {
        if(arr1[i]>=arr2[j])
        {
            arr[k]=arr2[j];
            j++;
        }
        else
        {
            arr[k]=arr1[i];
            i++;
        }
        k++;
    }
    while(i<n1)
    {
        arr[k]=arr1[i];
        i++;
        k++;
    }
    while(j<n2)
    {
        arr[k]=arr2[j];
```



```
43              j++;
44              k++;
45          }
46
47      }
48      void mergesort_rec(int p,int t)
49      {
50          int n=1;
51          if(p<t)
52          {
53              int q=p+(t-p)/2;
54              mergesort_rec(p,q);
55              mergesort_rec(q+1,t);
56              merge(p,q,t);
57              printf("\nPass %d :",n);
58              n++;
59              for(int i=0;i<size;i++)
60              {
61                  printf(" %d\t",arr[i]);
62              }
63              printf("\n");
64
65              }
66      }
67      int min(int x, int y)
68      {
69          return (x<y)? x :y;
70      }
71      void merge_sort_iter(int N)
72      {
73          for(int sub_size=1;sub_size<N;sub_size*=2)
74          {
75              for(int L=0; L<N; L+=(2*sub_size))
76              {
77                  int Mid=min(L+sub_size-1,N-1);
78                  int R=min(L+2*sub_size-1,N-1);
79                  merge(L,Mid,R);
80              }
81          }
82      }
83      void print_array()
84      {
```

```
83  void print_array()
84  {
85      cout<<"\nSorted array:\n";
86      for (int i = 0; i < size; i++)
87      {
88        cout << arr[i] << " ";
89      }
90  }
91  int main()
92  {
93     int choice, ans;
94     do
95     {
96        cout << "----------MERGE SORT-----------\n";
97        cout << "1] Iterative Method\n";
98        cout << "2] Recursive Method\n";
99        cout << "-----------------------------------\n";
100       cout<<"\n\nEnter your choice:";
101       cin >> choice;
102       int i;
103       cout<<"\nEnter the size of array:";
104       cin>>size;
105       cout<<"\nEnter the elements: ";
106       for(i=0;i<size;i++)
107       {
108          scanf("%d",&arr[i]);
109       }
110       switch (choice)
111       {
112         case 1:
113             merge_sort_iter(size);
114             print_array();
115             break;
116         case 2:
117             mergesort_rec(0,size-1);
118             print_array();
119             break;
120         default:
121             cout << "Enter a valid choice!!\n";
122             break;
123       }
124       cout << "\n\nDo you want to continue (Yes=1) :";
125       cin >> ans;
126     } while (ans == 1);
127 }
128
```

4. O/P SNIPPET (both iterative and recursive)

```
G:\SEM6\AA LAB\codes\merge sort.exe
----------MERGE SORT-----------
1] Iterative Method
2] Recursive Method
-----------------------------------


Enter your choice:1

Enter the size of array:5

Enter the elements: 2
3
4
1
5

Sorted array:
1 2 3 4 5

Do you want to continue (Yes=1) :1
----------MERGE SORT-----------
1] Iterative Method
2] Recursive Method
-----------------------------------


Enter your choice:2

Enter the size of array:5

Enter the elements: 5
4
3
2
1

Pass 1 : 4       5       3       2       1

Pass 1 : 3       4       5       2       1

Pass 1 : 3       4       5       1       2

Pass 1 : 1       2       3       4       5

Sorted array:
1 2 3 4 5
```

5. ANALYSIS AND ITS PROOF –

General analysis

T(N) = Time Complexity for problem size N
T(n) = Θ(1) + 2T(n/2) + Θ(n) + Θ(1)
T(n) = 2T(n/2) + Θ(n)

Let us analyze this step by step:
T(n) = 2 * T(n/2) + 0(n)

STEP-1 Is to divide the array into two parts of equal size .
2 * T(n/2) --> Part 1

STEP-2 Now to merge baiscall traverse through all the elements.
constant * n --> Part 2

STEP-3 --> COMBINE 1 + 2
T(n) = 2 * T(n/2) + constant * n --> Part 3

Now we can further divide the array into two halfs if size of the partition arrays are greater than 1. So,

n/2/2--> n/4
T(N) = 2 * (2 * T(n/4) + constant * n/2) + constant * n
T(N) = 4 * T(n/4) + 2 * constant * n

For this we can say that:
Where n can be subtituted to 2^k and the value of k is logN
T(n) = 2^k * T(n/(2^k)) + k * constant * n

Hence,
T(N) = N * T(1) + N * logN
= O(N * log(N))

Analysis using recursion tree method:

Let's assume that T(n) is the worst-case time complexity of merge sort for n integers.
When n >1 (merge sort on a single element takes constant time), then we can break down the time complexities as follows:

- Divide part: the time complexity of the divide part is O(1) because calculating the middle index takes constant time.

- Conquer part: we are recursively solving two sub-problems, each of size n/2. So the time complexity of each subproblem is T(n/2) and the overall time complexity of the conquer part is 2T(n/2).
- Combine part: As calculated above, the worst-case time complexity of the merging process is O(n).

For calculating the T(n), we need to add the time complexities of the divide, conquer, and combine part => T(n) = O(1) + 2T(n/2) + O(n) = 2T(n/2) + O(n) = 2T(n/2) + cn

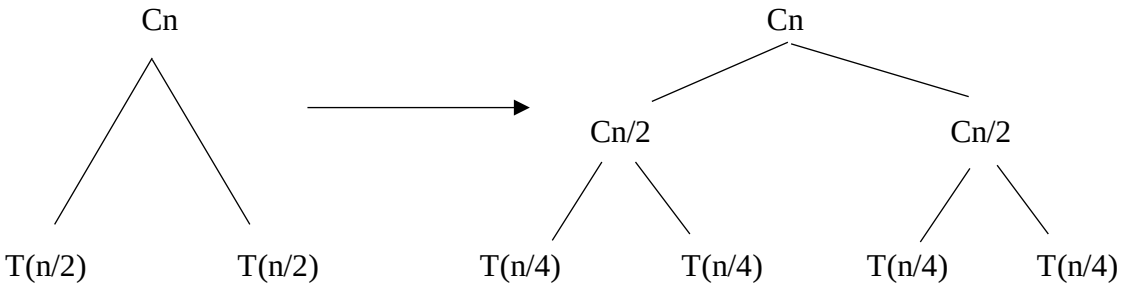
- T (n) = c, if n = 1
- T(n) = 2 T(n/2) + cn, if n > 1

We draw a recursion tree and count the total number of operations at every level.Finally we find the overall time complextity by doing sum of operations of every level.

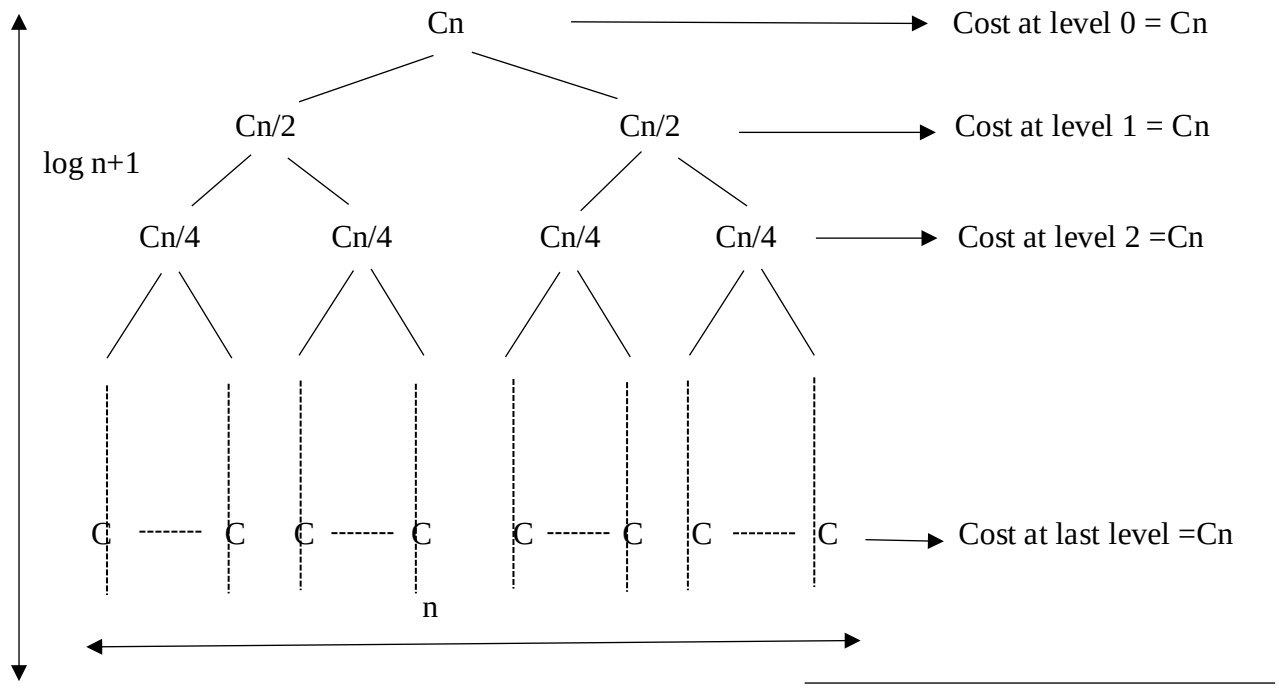
Recurrence relation T(n) = c, if n=1
of merge sort 2T(n/2) + Cn, if n>1

Recursion tree

T(n) =



Complete recursion tree diagram



TOTAL COST= Cnlogn + Cn

COMPLEXITY(ITERATIVE)

- Worst Case Time Complexity = O(N logN)
- Average Case Time Complexity = O(N logN)
- Best Case Time Complexity = O(N logN)
- Space Complexity = O(N)

COMPLEXITY(RECURSIVE)

- Worst Case Time Complexity = O(N logN)
- Average Case Time Complexity = O(N logN)
- Best Case Time Complexity = O(N logN)
- Space Complexity = O(N)

6. TIME COMPLEXITY

A. BEST CASE TIME COMPLEXITY

= O(N logN)

It occurs when there is no sorting required, i.e. the array is already sorted.

EXAMPLE -

[10,20,30,40,50]

STEP -1 divide it into equal parts
[10,20,30] and [40,50]

It can be further divided into smaller parts
[10,20] [30] [40,50]

Intially, let us assume that the count of swaps is 0, now if we check for the condition where first < second and if not than we increase the count swap by 1.

[10,20] == first < second count=0;
[30] == only one element so no comparisons count=0;
[40,50] == first < second count=0;

Now lets merge back all the parts into one sorted array

[10,20,30,40,50]--> no swaps required in merging as well.

Therefore in Best Case,

- Input is already sorted
- Best Case Time Complexity: O(N logN)
- Number of Comparisons: 0.5 N logN

B. AVERAGE CASE TIME COMPLEXITY

= O(N logN)

It occurs when the array elements are in jumbled order that is not properly ascending and not properly descending.

EXAMPLE-

[2,4,5,12,8,10,6]

one part is going to have more elements so if we split it into two possible way

OPTION-1

[2,4,5] and [12,8,10,6]

it can be further divided into smaller parts
[2,4] [5] [12,8] [10,6]

initially let's assume that the count of swaps is 0, now if we check for the condition where first < second and if not than we increase the count swap by 1.

[2,4] == first < second count=0;
[5] == count =0;
[12,8] != first < second count=1;
[10,6] != first < second coutn=2;

Now lets merge back all the parts into one sorted array

[2,4,5] [6,8,10,12] ---->Here one thing can be checked that if the last element of first part < first element of second part so one can say in merging into one it would have 0 swaps.

[2,4,5,6,8,10,12]

OPTION -2

[2,4,5,12] and [8,10,6]

it can be further divided into smaller parts

[2,4] [5,12] [8,10] [6]

intially lets assume that the count of swaps is 0, now if we check for the condition where first < second and if not than we increase the count swap by 1.

[2,4] == first < second count=0;
[5,12] == first < second =0;
[8,10] == first < second count=0;
[6] == count=0;

Now lets merge back all the parts into one sorted array

[2,4,5,12] [6,8,10] > here we will be comparing 6 with 2 4 5 12
-----> and then insert 6 after 5 and then same with 8 and 10.

[2,4,5,6,8,10,12]

Average case takes 0.26N less comparisons than worst case.

Therefore, in Average Case:

- Average Case Time Complexity: O(N logN)
- Number of Comparisons: 0.74 N logN

C. WORST CASE TIME COMPLEXITY

= O(N logN)

It occurs when the array elements are required to be sorted in reverse order. That means suppose you have to sort the array elements in ascending order, but its elements are in descending order.

EXAMPLE -

[40,0,60,20,50,10,70,30]

STEP -1 divide it into equal parts

[40,0,60,20] and [50,10,70,30]

It can be further divided into smaller parts

[40,0] [60,20] [50,10] [70,30]

intially lets assume that the count of swaps is 0, now if we check for the condition where first < second and if not than we increase the count swap by 1.

[40,0] != first < second count=1;
[60,20] != first < second count=2;
[50,10] != first < second count=3;
[70,30] != first < second coutn=4;

Now lets merge back all the parts into one sorted array

[0,40,20,60] [10,30,50,70] ----> maximum comparison again every pair of set compared.
[0,10,20,30,40,50,60,70]

Therefore, in Worst Case:

- Input: Specify distribution
- Worst Case Time Complexity: O(N logN)
- Number of Comparisons: N logN

7. SPACE COMPLEXITY

It takes O(N) space as we divide the array and store it into them where the total space consumed in making all the array and merging back into one array is the total number of elements present in it.

8. UNIQUE CHARACTERISTICS

- Overall time complexity of Merge sort is O(nLogn). It is more efficient as it is in worst case also the runtime is O(nlogn)

- The space complexity of Merge sort is O(n). This means that this algorithm takes a lot of space and may slower down operations for the last data sets.

9. APPLICATIONS

- Merge Sort is useful for sorting linked lists in O(n Log n) time
- Merge sort can be implemented without extra space for linked lists
- Merge sort is used for counting inversions in a list
- Merge sort is used in external sorting

10. OPTIMIZATION

There is a space-optimized approach to implement the merge sort called in-place merge sort in which instead of copying an array in left and right subarray we divide an array with help of pointers logically creating division in the original array by specifying the window for every recursive call. We shift the elements of the array to finally achieve the sorted configuration.

Thus taking no extra space and having O(1) space complexity.

11. ADVANTAGES

- It is quicker for larger lists because unlike insertion and bubble sort it doesn't go through the whole list several times.
- It has a consistent running time, carries out different bits with similar times in a stage.
- Merge sort algorithm is best case for sorting slow-access data
- Merge sort algorithm is better at handling sequential - accessed lists.

12. DISADVANTAGES

- Merge sort is not efficient for sorting input of large size if you are having low stack space.
- Merge sort while sorting the array goes through the entire process even if the array is sorted.
- Merge sort takes an extra space of O(n) in standard(Outplace) implementation.

ADVANCED ALGORITHM LAB – 3

Title - Quick Sort - Analysis and Implementation

Quick Sort is a sorting algorithm that employs the divide-and-conquer strategy. Here, we pick one element as a pivot and generate an array partition around it. We sort our array by repeating this procedure for each partition.

We can use quick sort in a variety of methods based on the pivot's position.

- pivoting on the first or final element
- pivoting on the middle element

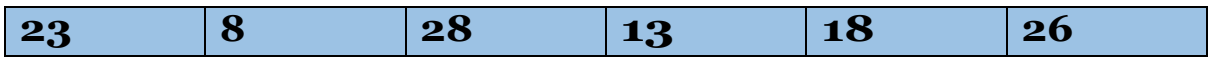
1. WORKING MECHANISM

1. In the array, look for a "pivot" element. For a single round, this element acts as the comparison point.
2. Begin with the first item in the array (the left pointer).
3. Begin a pointer (the right pointer) at the array's last item.
4. Move the left pointer to the right while the value at the left pointer in the array is smaller than the pivot value (add 1). Continue until the pivot value is larger than or equal to the value at the left pointer.

5. Move the right pointer to the left while the value at the right pointer in the array is greater than the pivot value (subtract 1). Continue until the pivot value is less than or equal to the value at the right pointer.
6. Swap the values at these points in the array if the left pointer is less than or equal to the right pointer.
7. Move the left pointer one place to the right and the right pointer one space to the left.
8. Return to step 1 if the left and right pointers do not meet.

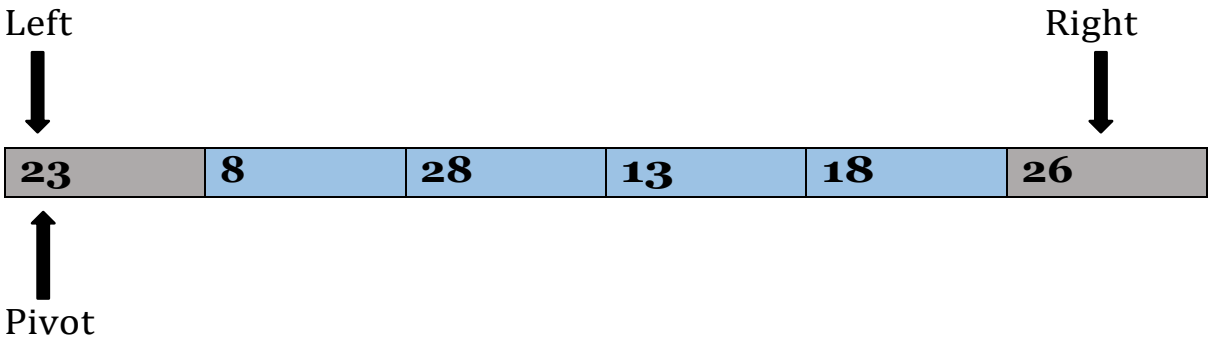
EXAMPLE

Elements of array are -

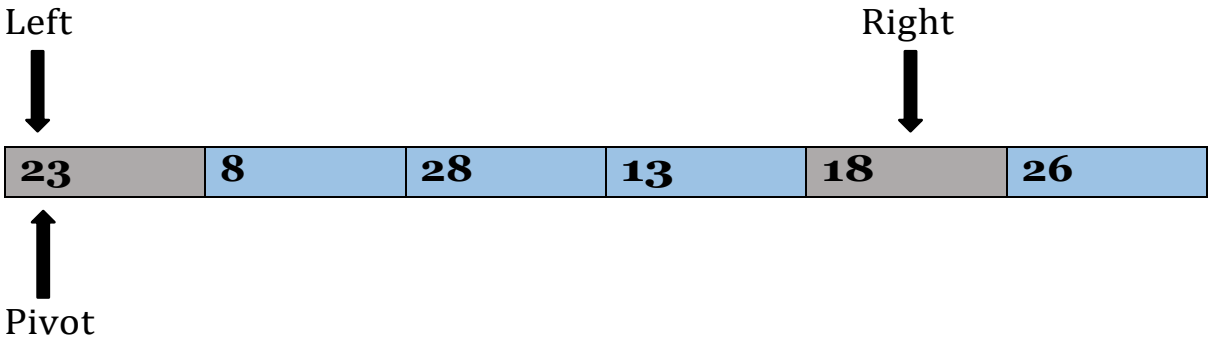


Here, the leftmost element is considered as pivot.
a[left] = 23, a[right] = 26 and a[pivot] = 23.

Since, pivot is at left, the algorithm starts from right and moves towards left.

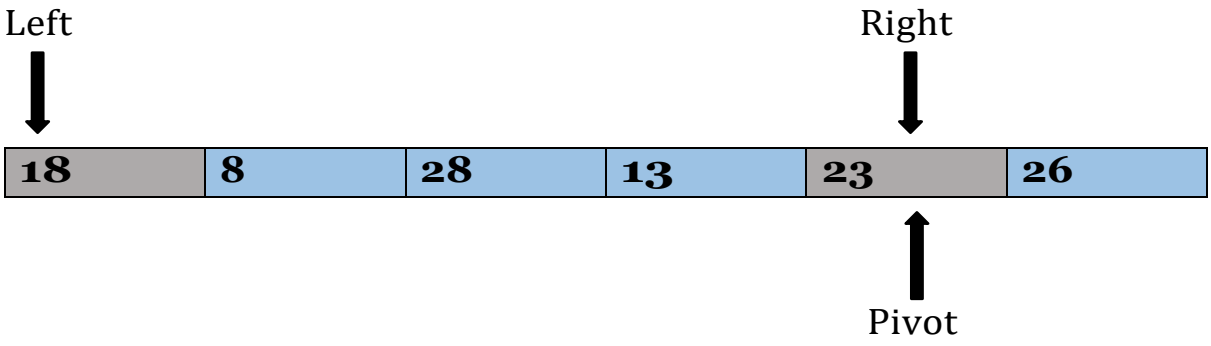


a[pivot] < a[right], so algorithm moves forward one position towards left.

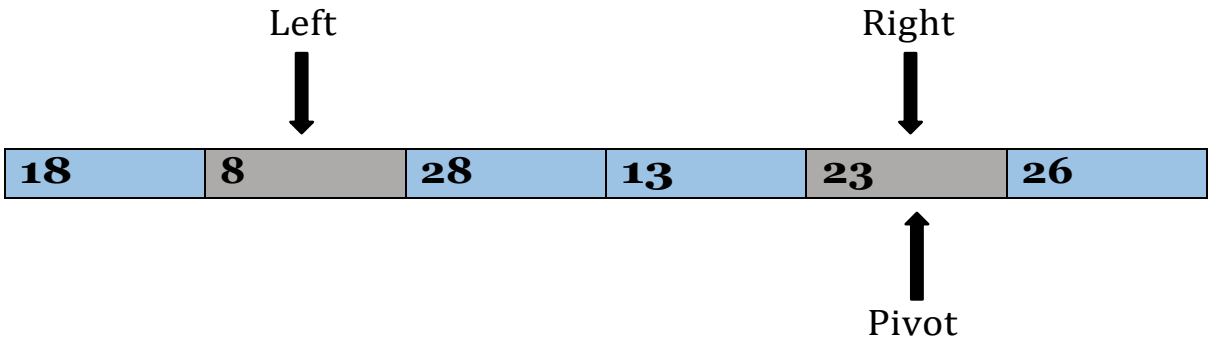


a[left] = 23, a[right] = 18, and a[pivot] = 23.

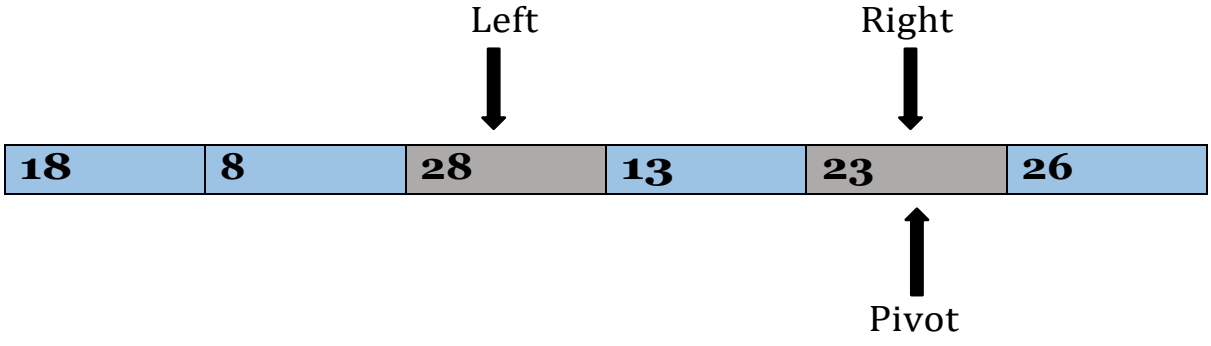
a[pivot] > a[right], so, algorithm will swap a[pivot] with a[right], and pivot moves to right.



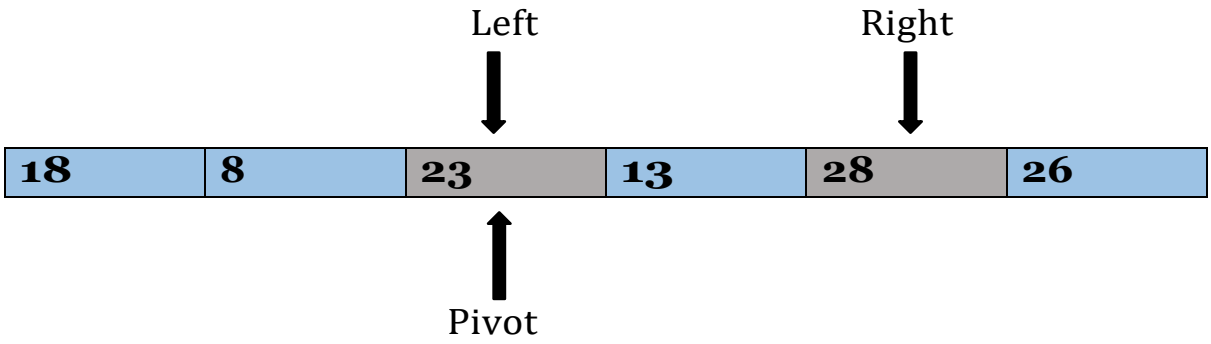
a[left] = 18, a[right] = 23, and a[pivot] = 23. Since, pivot is at right, so algorithm starts from left and moves to right.
a[pivot] > a[left], so algorithm moves one position to right.



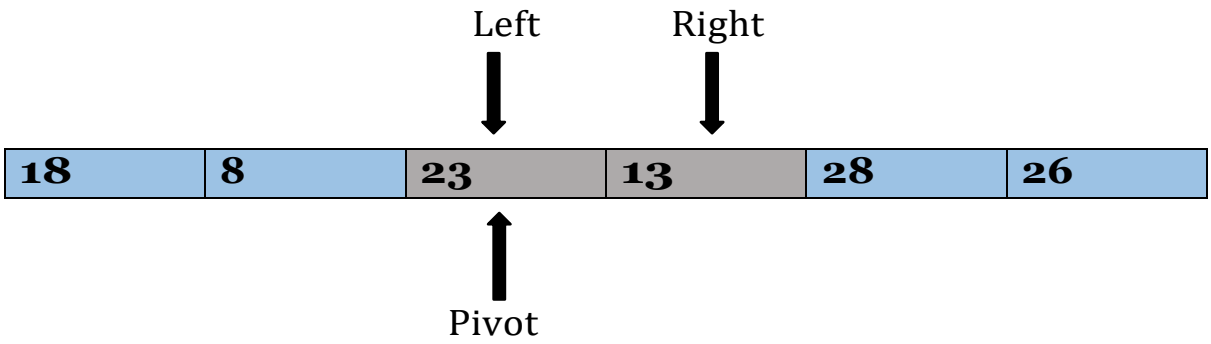
a[left] = 8, a[right] = 23, and a[pivot] = 23.
a[pivot] > a[left], so algorithm moves one position to right



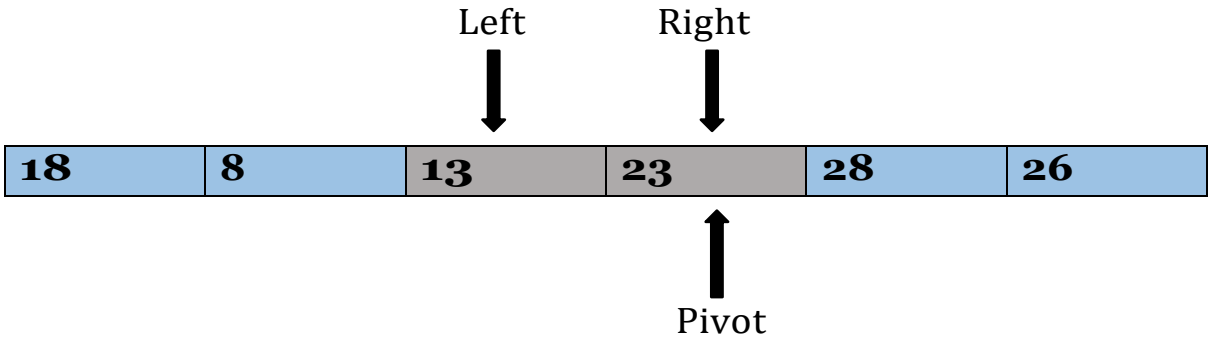
a[left] = 28, a[right] = 23, and a[pivot] = 23.
a[pivot] < a[left], swap a[pivot] and a[left], now pivot is at left.



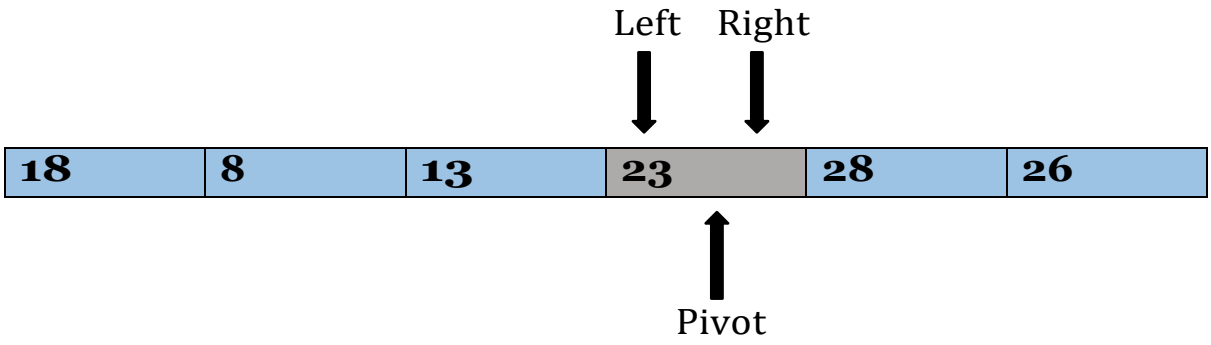
Pivot is at left, so algorithm starts from right and moves to left. a[left] = 23, a[right] = 28, and a[pivot] = 23.
a[pivot] < a[right], so algorithm moves one position to left.



a[pivot] = 23, a[left] = 23, and a[right] = 13.
a[pivot] > a[right], swap a[pivot] and a[right], now pivot is at right.



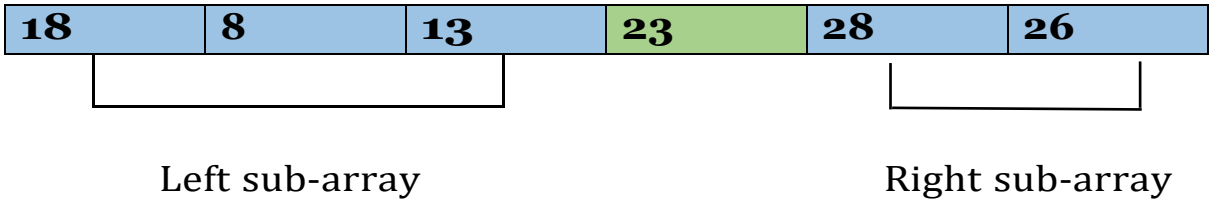
a[pivot] = 23, a[left] = 13, and a[right] = 23.
Pivot is at right, so the algorithm starts from left and move to right.



a[pivot] = 23, a[left] = 23, and a[right] = 23. So, pivot, left and right are pointing the same element. It represents the termination of procedure.

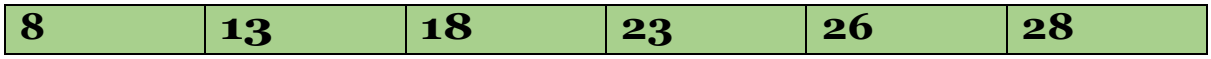
Element 23, which is the pivot element is placed at its exact position.

Elements that are right side of element 23 are greater than it, and the elements that are left side of element 23 are smaller than it.



In a similar manner, quick sort algorithm is separately applied to the left and right sub-arrays.

After sorting gets done, the array will be -



2. PSEUDO CODE

RECURSIVE-

```
1. QUICKSORT (array A, start, end)
2. {
3. if (start < end)
4. {
5. p = partition(A, start, end)
6. QUICKSORT (A, start, p - 1)
7. QUICKSORT (A, p + 1, end)
8. }
9. }
```

ITERATIVE-

```
1   void quick_sort_iter(int arr[], int l, int h) {
2   int stack[h-l+1];
3   int top = -1;
4   stack[++top]=l;
5   stack[++top]=h;
6   while(top >= 0)
7   {
8   h = stack[top--];
9   l = stack[top--];
10 int p = partition(arr, l, h);
11 if(p-1>l)
12 {
13 stack[++top] = l;
14 stack[++top] = p - 1;
15 }
16 if(p+1<h)
17 {
18 stack[++top] = p + 1;
19 stack[++top] = h;
20 }
21 }
22 }
```

3. PROGRAM CODE SNIPPET (both iterative and recursive)

quick_sort_both.cpp

```cpp
#include <bits/stdc++.h>
using namespace std;
int n,a[100];
int partition(int arr[], int l, int h) {
  int x = arr[h];
  int i = (l - 1);
  for (int j = l; j <= h - 1; j++) {
    if (arr[j] <= x) {
      i++;
      swap(arr[i], arr[j]);
    }
  }
  swap(arr[i + 1], arr[h]);
  return (i + 1);
}
void quick_sort_iter(int arr[], int l, int h) {
  int stack[h-l+1];
  int top = -1;
  stack[++top]=l;
  stack[++top]=h;
  while(top >= 0)
  {
    h = stack[top--];
    l = stack[top--];
    int p = partition(arr, l, h);
    if(p-1>l)
    {
      stack[++top] = l;
      stack[++top] = p - 1;
    }
    if(p+1<h)
    {
      stack[++top] = p + 1;
      stack[++top] = h;
    }
  }
}
void quick_sort_rec(int arr[], int l, int h) {
    if (l < h)
        {
            int q = partition(arr, l,h);
            quick_sort_rec(arr, l, q - 1);
            quick_sort_rec(arr, q + 1, h);
        }
}
```

```
void print_array(){
     for(int i=0;i<n;i++){
      cout<<a[i]<<" ";
   }
}
 int main()
{
   int choice, ans;
   do{
      cout << "----------QUICK SORT-----------\n";
      cout << "1] Iterative Method\n";
      cout << "2] Recursive Method\n";
      cout << "-----------------------------------\n";
      cout<<"\n\nEnter your choice:";
      cin >> choice;
      int i;
      cout<<"\nEnter the size of array:";
      cin>>n;
      cout<<"\nEnter the elements: ";
      for(i=0;i<n;i++)
      {
          cin>>a[i];
      }
      switch (choice)
      {
        case 1:
            quick_sort_iter(a,0,n-1);
            cout<<"\nSorted Array:\n";
            print_array();
            break;
        case 2:
            quick_sort_rec(a,0,n-1);
            cout<<"\nSorted Array:\n";
            print_array();
            break;
       default:
            cout << "Enter a valid choice!!\n";
            break;
      }
      cout << "\n\nDo you want to continue (Yes=1) :";
      cin >> ans;
    } while (ans == 1);
}
```

4. O/P SNIPPET (both iterative and recursive)

```
G:\SEM6\AA LAB\codes\quick_sort_both.exe
----------QUICK SORT-----------
1] Iterative Method
2] Recursive Method
-----------------------------------


Enter your choice:1

Enter the size of array:5

Enter the elements: 9
8
7
6
5

Sorted Array:
5 6 7 8 9

Do you want to continue (Yes=1) :1
----------QUICK SORT-----------
1] Iterative Method
2] Recursive Method
-----------------------------------


Enter your choice:2

Enter the size of array:5

Enter the elements: 5
4
3
2
1

Sorted Array:
1 2 3 4 5

Do you want to continue (Yes=1) :0

--------------------------------
Process exited after 20.09 seconds with return value 0
Press any key to continue . . .
```

5. ANALYSIS AND ITS PROOF –

➢ BEST CASE RUNNING TIME:

Let T(n) be the time complexity for best cases
n = total number of elements
then
T(n) = 2*T(n/2) + constant*n
2*T(n/2) is because we are dividing array into two array of equal size
constant*n is because we will be traversing elements of array in each level of tree

therefore,
T(n) = 2*T(n/2) + constant*n
further we will divide array in to array of equal size so
T(n) = 2*(2*T(n/4) + constant*n/2) + constant*n == 4*T(n/4) + 2*constant*n

for this we can say that
T(n) = 2^k * T(n/(2^k)) + k*constant*n
then n = 2^k
k = log2(n)

therefore,
T(n) = n * T(1) + n*logn = O(n*log2(n))

➢ AVERAGE CASE RUNNING TIME:

Let T(n) be total time taken

then for average we will consider random element as pivot
lets index i be pivot

then time complexity will be
T(n) = T(i) + T(n-i)

T(n) = 1/n *[\sum_{i=1}^{n-1} T(i)] + 1/n*[\sum_{i=1}^{n-1} T(n-i)]

As [\sum_{i=1}^{n-1} T(i)] and [\sum_{i=1}^{n-1} T(n-i)] equal likely functions therefore
T(n) = 2/n*[\sum_{i=1}^{n-1} T(i)]

multiply both side by n
n*T(n) = 2*[\sum_{i=1}^{n-1} T(i)] ..... (1)

put n = n-1
(n-1)*T(n-1) = 2*[\sum_{i=1}^{n-2} T(i)] .....(2)

substract 1 and 2
then we will get
n*T(n) - (n-1)*T(n-1) = 2*T(n-1) + c*n^2 + c*(n-1)^2
n*T(n) = T(n-1)[2+n-1] + 2*c*n - c
n*T(n) = T(n-1)*(n+1) + 2*c*n [removed c as it was constant]

divide both side by n*(n+1),
T(n)/(n+1) = T(n-1)/n + 2*c/(n+1) .....(3)

put n = n-1,
T(n-1)/n = T(n-2)/(n-1) + 2*c/n .....(4)

put n = n-2,
T(n-2)/n = T(n-3)/(n-2) + 2*c/(n-1) .....(5)

by putting 4 in 3 and then 3 in 2 we will get
T(n)/(n+1) = T(n-2)/(n-1) + 2*c/(n-1) + 2*c/n + 2*c/(n+1)

also we can find equation for T(n-2) by putting n = n-2 in (3)

at last we will get

T(n)/(n+1) = T(1)/2 + 2*c * [1/(n-1) + 1/n + 1/(n+1) + ...]

T(n)/(n+1) = T(1)/2 + 2*c*log(n) + C

T(n) = 2*c*log(n) * (n+1)

now by removing constants,

T(n) = log(n)*(n+1)

therefore,

T(n) = O(n*log(n))

➢ WORST CASE RUNNING TIME:

let T(n) ne total time complexity for worst case
n = total number of elements

T(n) = T(n-1) + constant*n
as we are dividing array into two parts one consist of single element and other of n-1

and we will traverse individual array

T(n) = T(n-2) + constant*(n-1) + constant*n = T(n-2) + 2*constant*n - constant
T(n) = T(n-3) + 3*constant*n - 2*constant - constant
T(n) = T(n-k) + k*constant*n - (k-1)*constant ….. - 2*constant - constant

T(n) = T(n-k) + k*constant*n - constant*[(k-1).....+ 3 + 2 + 1]
T(n) = T(n-k) + k*n*constant - constant*[k*(k-1)/2]
put n=k
T(n) = T(0) + constant*n*n - constant*[n*(n-1)/2]
removing constant terms
T(n) = n*n - n*(n-1)/2
T(n) = O(n^2)

COMPLEXITY(ITERATIVE)

- Worst Case Time Complexity = O(N^2)
- Average Case Time Complexity = O(N logN)
- Best Case Time Complexity = O(N logN)
- Space Complexity:
    a. Best = O(logN)
    b. Worst = O(N)

COMPLEXITY(RECURSIVE)

- Worst Case Time Complexity = O(N^2)
- Average Case Time Complexity = O(N logN)
- Best Case Time Complexity = O(N logN)
- Space Complexity:
    a. Best = O(logN)
    b. Worst = O(N)

6. TIME COMPLEXITY

A. BEST CASE TIME COMPLEXITY

= O(Nlog(N))

- When we choose pivot as a mean element, we get the best quick sort result.

- In this situation, the recursion will be where the height of the tree is logN and we will traverse all of the items at each level with total operations of logN * N.
- Because we chose the mean element as the pivot, the array will be divided into equal-sized branches, resulting in a tree with a minimum height.

B. Average Case Time Complexity

= **O(Nlog(N))**

- It happens when the array elements are in a jumbled sequence that isn't ascending or descending properly.
- This is the general average scenario for speedy sorting, which we shall obtain by averaging all complexities.

C. worst Case Time Complexity

= **O(N^2)**

- The worst case scenario arises when the pivot element is either the greatest or the smallest element in a quick sort. If the pivot element is always the last element of the array, the worst case scenario is that the array is already sorted ascending or descending.

7. SPACE COMPLEXITY

In the worst-case scenario, the partition is always unbalanced, and there will be only 1 recursive call at each level of recursion. In such a scenario, the generated tree is skewed in nature. So the height of the tree = O(n) and recursion requires a call stack of size O(n).
The worst-case space complexity of the quick sort = O(n).

The partition is always balanced in the best-case scenario, and there will be 2 recursive calls at each recursion level. In such a
scenario, the generated tree is balanced in nature. So the height of the tree = O(logn), and recursion requires a call stack of size O(logn).
The best-case space complexity of the quick sort = O(logn).

8. Unique Characteristics

- On average, it performs well. Quicksort is perhaps the greatest option for those who are unsure which sort to utilize.
- One of the most effective algorithms for learning the concept of recursion. It has a recursive structure, a recursive flow, and a base case that are intuitive.

- In the virtual memory environment, it's an in-place sorting that works well.
- The Quicksort partition technique is a great way to learn how to solve problems with two-pointers.
- A useful algorithm for grasping the **randomized algorithm's concept**. Randomization is a tool that can be used to improve real-time system efficiency

9. APPLICATIONS

- Quicksort is used in situations where a stable sort isn't required.
- To separate the k smallest or largest components, quicksort variants are utilized.
- Quicksort's divide-and-conquer feature allows parallelization to be used.
- When used for arrays, Quick Sort is a cache-friendly sorting algorithm since it has strong locality of reference.
- Because Quick Sort is tail recursive, all tail call optimizations are possible.
- Quick Sort is an in-place sort that requires no additional storage.

10. OPTIMIZATIONS

- Recurse first into the smaller side of the partition, then into the other, to ensure that no more than O(log n) space is used.
  If the array has fewer than ten elements, utilize a non-recursive sorting method like insertion sort, which performs fewer swaps, comparisons, and other operations on such small arrays.

- Keep it on hold until the entire array has been processed when the number of elements is less than a threshold k. After that, sort it by insertion. When the recursion is terminated early, the array is k-sorted, which means that each element is only k positions away from its final sorted location. Insertion sort takes O(kn) time to complete in this scenario, which is linear if k is a constant. In current architectures, this results in an inefficient usage of cache memories.

11. ADVANTAGES

- It's in place since it just requires a modest auxiliary stack.
- Sorting n things takes only n (log n) seconds.
- The inner loop is extremely short.
- After a thorough mathematical investigation of this Programme, a fairly specific comment about performance issues can be made.

12. DISADVANTAGES

- It's a recursive mechanism. The implementation is highly difficult, especially if recursion is not provided.
- In the worst-case scenario, quadratic (n^2) time is required.
- It is fragile, in the sense that a simple implementation error can go unnoticed and lead it to function poorly.

ADVANCED ALGORITHM LAB – 4

Title - Heap Sort - Analysis and Implementation

- A heap is a complete binary tree, and the binary tree is a tree in which the node can have the utmost two children. A complete binary tree is a binary tree in which all the levels except the last level, i.e., leaf node, should be completely filled, and all the nodes should be left-justified.
- Heapsort is a popular and efficient sorting algorithm. The concept of heap sort is to eliminate the elements one by one from the heap part of the list, and then insert them into the sorted part of the list.
- Heapsort is the in-place sorting algorithm.

1. WORKING MECHANISM

Step 1: Create a heap using the input data. Create a max heap for sorting in ascending order and a min heap for sorting in descending order.

Step 2: Swap the root element with the heap's last item.

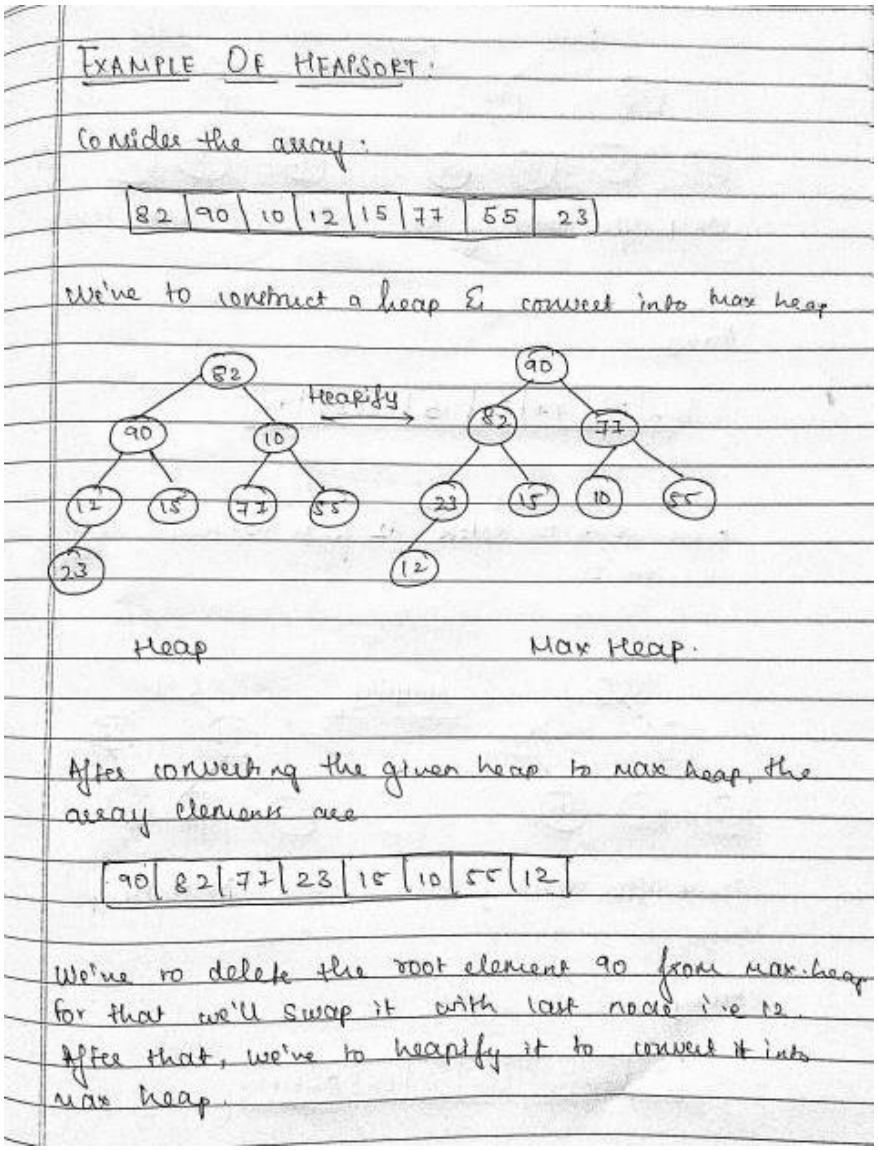
Step 3: Reduce the size of the heap by one.

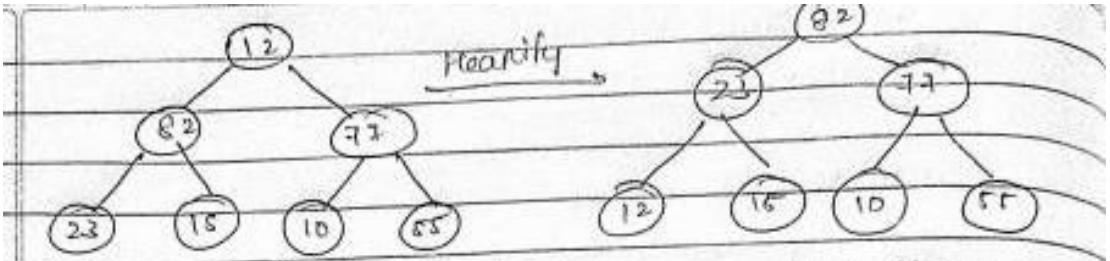
Step 4: Call heapify on the root node to heapify the remaining items into a heap of the new heap size.

Step 5: Recursively call. Repeat steps 2,3,4 as needed until the heap is larger than 2.

When the last array position has the correct element, it is discarded from the heap. The method is repeated until all of the elements in the input array have been sorted. This occurs when the heap size is reduced to 2, because the first two components of a heap that meets the heap property will be in order by default.

EXAMPLE-



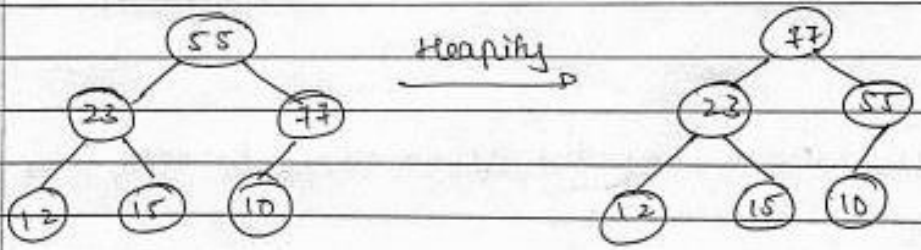


Heap after deleting '90'. Max Heap.

Array:

82	23	77	12	15	10	55	90

Now, we've to delete '82' from max-heap. We'll swap it with '55'.

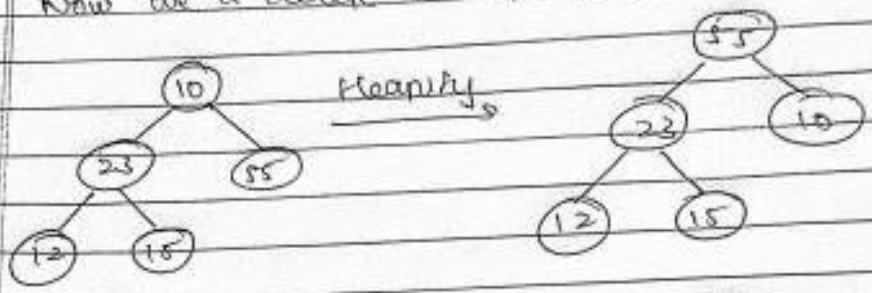


Heap after deleting '82' Max. Heap

Array:

77	23	55	12	15	10	82	90

Now we'll delete '77' & swap it with '10'.

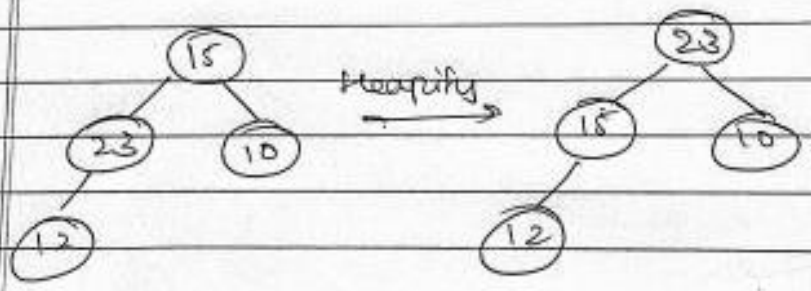


Heap after deleting '77'. Max-Heap

Array:

55	23	10	12	15	77	82	90

We'll delete '55' & swap it with '15'.

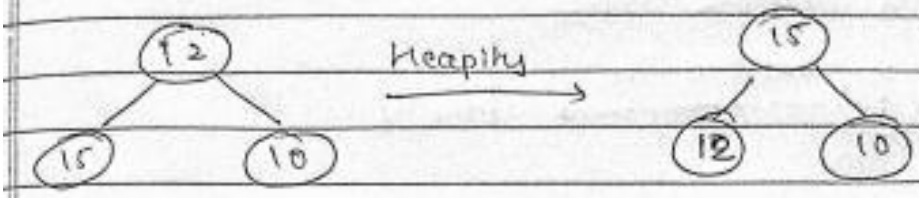


Heap after deleting '55'. Max-Heap

Array:

23	15	10	12	55	77	82	90

Now, We'll delete '23' & swap with '12'.

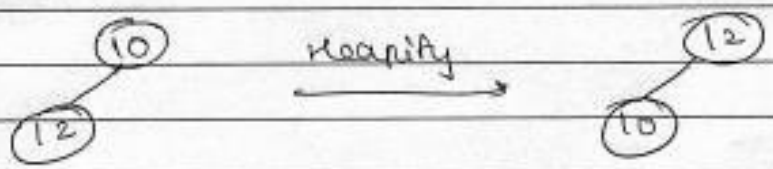


Heap after deleting '23'. Max-Heap.

Array:

15	12	10	23	55	77	82	90

We'll delete '15' & swap with '10'.

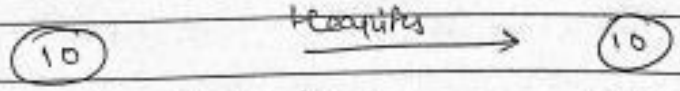


Heap after deleting '15'. Max. Heap.

Array:

12	10	15	23	55	77	82	90

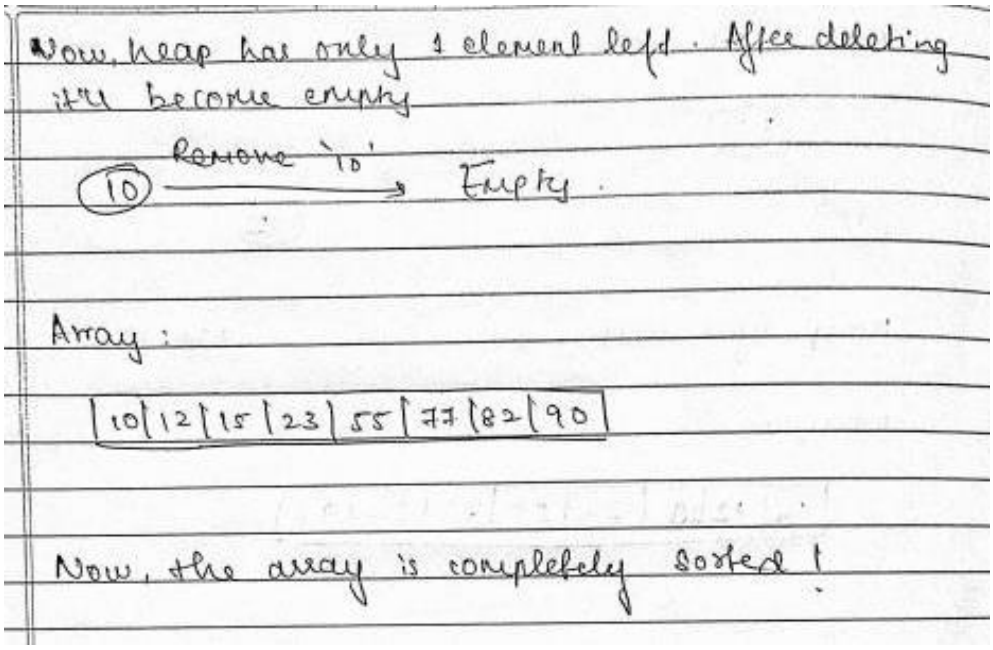
We'll delete '12' & swap with '10'



Heap after deleting '12' Max-Heap

Array:

10	12	15	23	55	77	82	90



2. PSEUDO CODE

ITERATIVE-

```
1.  void swap(int arr[],int first,int second){
2.    int auxiliary=arr[first];
3.    arr[first]=arr[second];
4.    arr[second]=auxiliary; 5.
}
6.  int compare(int arr[],int left,int right,int root,int size){
7.    int location = -1;
8.          if(left < size && arr[left] > arr[root] ){
9.          if(right < size && arr[right] > arr[left]){
10.               swap(arr,right,root);
11.               location = right;
12.             }
13.             else{
14.               swap(arr,left,root);
15.               location = left;
16.             }}
17.           else if(right < size && arr[right] > arr[root]){
18.             swap(arr,right,root);
19.             location = right;
```

```
20.          }
21.          return location;   }
22.        void heap(int arr[],int size,int root){
23.          int left,right;
24.          while(root!=-1) {
25.            left = 2*root+1;
26.            right = 2*root+2;
27.          root = compare(arr, left, right, root,  size);28.
             } }
29.        void heap_sort(int arr[],int size){
30.          for (int i = (size/2)-1; i >= 0; i--){
31.            heap(arr, size, i);
32.          }
33.          for (int i = size-1; i >= 0; i--){
34.            swap(arr, 0, i);
35.            heap(arr, i, 0);
36.          } }
```

RECURSIVE-

```
1.  void max_heapify(int arr[],int n,int i){
2.          int par, left, right, tmp;
3.          par=i;
4.          left=2*i+1;
5.          right=2*i+2;
6.          if (left<n && arr[left]>arr[par])
7.                  par = left;
8.          if (right <n && arr[right] > arr[par])
9.                  par = right;
10.                 if(par!=i){
11.                         tmp=arr[i];
12.                         arr[i]=arr[par];
13.                         arr[par]=tmp;
14.                         max_heapify(arr,  n,  par);          }          }
15.         void heapsort(int arr[],int n){
16.                 int i, tmp;
17.                 for(i=n/2-1;i>=0;i--)
18.                         max_heapify(arr, n, i);
19.                 for(i=n-1;i>=0;i--){
20.                         tmp=arr[0];
21.                         arr[0]=arr[i];
22.                         arr[i]=tmp;
23.                         max_heapify(arr,i,0);          }        }
```

3. PROGRAM CODE SNIPPET (both iterative and recursive)

ITERATIVE-

heap_iter.cpp

```
#include <stdio.h>
//Swap two element in array
void swap(int arr[],int first,int second)
{
  int auxiliary=arr[first];
  arr[first]=arr[second];
  arr[second]=auxiliary;
}
int compare(int arr[],int left,int right,int root,int size)
{
  int location = -1;
    if(left < size &&  arr[left] > arr[root] )
  {
    if(right < size && arr[right] > arr[left])
    {
      swap(arr,right,root);
      location = right;
    }
    else
    {
      swap(arr,left,root);
      location = left;
    }
  }
  else if(right < size && arr[right] > arr[root])
  {
    swap(arr,right,root);
    location = right;
  }
  return location;
}
//Perform the operation of heap sort
void heap(int arr[],int size,int root)
{
  int left,right;
  while(root!=-1)
  {
    left  = 2*root+1;
    right = 2*root+2;

    root = compare(arr, left, right, root, size);
  }
}
//Sort array element using heap sort
void heap_sort(int arr[],int size)
```

```c
void heap_sort(int arr[],int size)
{

  for (int i = (size/2)-1; i >= 0; i--)
  {
    heap(arr, size, i);
  }
  for (int i = size-1; i >= 0; i--)
  {
    swap(arr, 0, i);
    heap(arr, i, 0);
  }
}
void print_data(int arr[],int size)
{

  for(int i = 0; i < size; i++)
  {
    printf("   %d",arr[i] );
  }
  printf("\n");
}
int main()
{
    int arr[30],size;
    printf("--------------ITERATIVE HEAPSORT-----------------");
    printf("\nEnter the size of array:");
    scanf("%d",&size);
    printf("\nEnter the elements: ");
    for(int i=0;i<size;i++)
    {
        scanf("%d",&arr[i]);
    }

  printf("\nBefore Sort\n");
  print_data(arr,size);
  heap_sort(arr,size);
  printf("\nAfter Sort\n");
  print_data(arr,size);
  return 0;
}
```

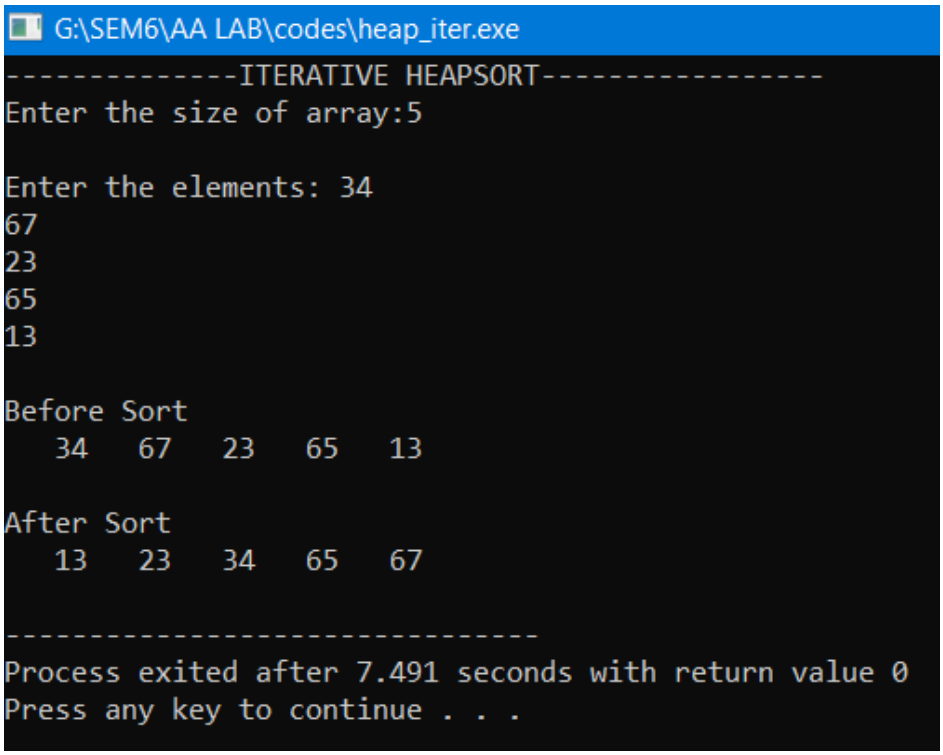
RECURSIVE-

heap_rec.cpp

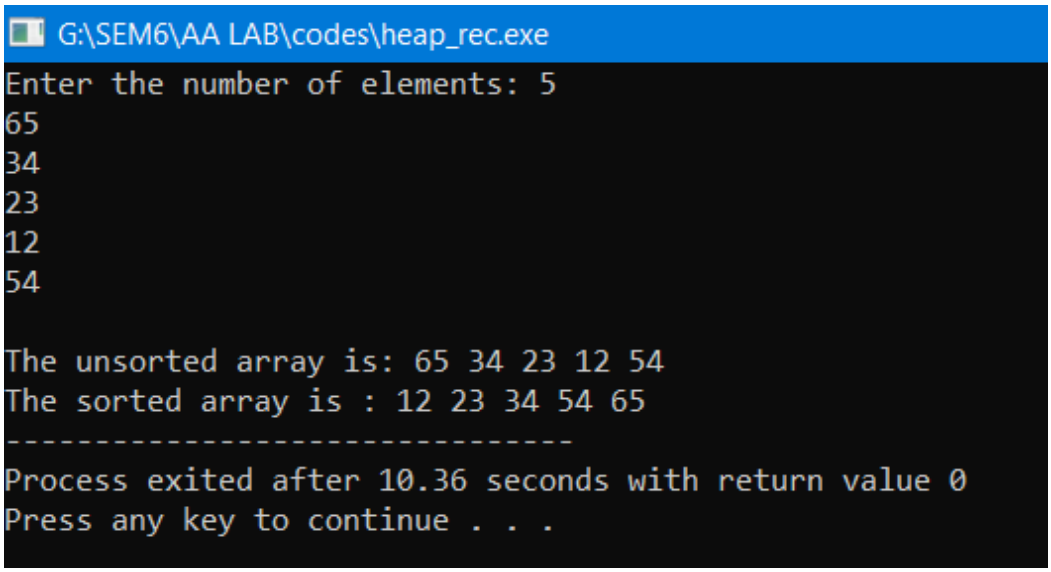
```c
#include<stdio.h>
void max_heapify(int arr[],int n,int i){
    int par, left, right, tmp;
    par=i;
    left=2*i+1;
    right=2*i+2;
    if (left<n && arr[left]>arr[par])
        par = left;
    if (right <n && arr[right] > arr[par])
        par = right;
    if(par!=i)
    {
        tmp=arr[i];
        arr[i]=arr[par];
        arr[par]=tmp;
        max_heapify(arr, n, par);
    }
}
void heapsort(int arr[],int n){
    int i, tmp;
    for(i=n/2-1;i>=0;i--)
        max_heapify(arr, n, i);
    for(i=n-1;i>=0;i--)
    {
        tmp=arr[0];
        arr[0]=arr[i];
        arr[i]=tmp;
        max_heapify(arr,i,0);
    }
}
int main() {
    int arr[20],n,i; //taking the number of elements as input
    printf("Enter the number of elements: ");
    scanf("%d",&n);
    for(i=0;i<n; i++)
        scanf("%d",&arr[i]);
    printf("\nThe unsorted array is: ");
    for(i=0;i<n;i++)
        printf("%d ", arr[i]);
    heapsort(arr,n);
    printf("\nThe sorted array is : ");
    for(i=0;i<n;i++)
        printf("%d ",arr[i]);
}
```

4. O/P SNIPPET (both iterative and recursive)

ITERATIVE-



RECURSIVE-



5. ANALYSIS AND ITS PROOF –

- **COMPLEXITY OF INSERTING A NEW NODE**

As we use binary trees, we know that the max height of such a structure is always O(log(n)). When we insert a new value in the heap, we will swap it with a value greater than it, to maintain the max-heap property. The number of such swaps would be O(log(n)). Therefore, the insertion of a new value when building a max-heap would be O(log(n)).

- **COMPLEXITY OF REMOVING THE MAX VALUED NODE FROM HEAP**

Likewise, when we remove the max valued node from the heap, to add to the end of the list, the max number of steps required would also be O(log(n)). Since we swap the max valued node till it comes down to the bottom-most level, the max number of steps we'd need to take is the same as when inserting a new node, which is O(log(n)).

Therefore, the total time complexity of the max_heapify function turns out to be O(log(n)).

- **COMPLEXITY OF CREATING A HEAP**

The time complexity of converting a list into a heap using the create_heap function is not O(log(n)). This is because when we create a heap, not all nodes will move down O(log(n)) times. It's only the root node that'll do so. The nodes at the bottom-most level (given by n/2) won't move down at all. The nodes at the second last level (n/4) would move down 1 time, as there is only one level below remaining to move down. The nodes at the third last level would move down 2 times, and so on. So if we multiply the number of moves we take for all nodes, mathematically, it would turn out like a geometric series, as explained below-

$(n/2 * 0) + (n/4 * 1) + (n/8 * 2) + (n/16 * 3) + ...h$

Here h represents the height of the max-heap structure.

The summation of this series, upon calculation, gives a value of n/2 in the end. Therefore, the time complexity of create_heap turns out to be O(n).

- **TOTAL TIME COMPLEXITY**

In the final function of heapsort, we make use of create_heap, which runs once to create a heap and has a runtime of O(n). Then using a for-loop, we call the max_heapify for each node, to maintain the max-heap property whenever we remove or insert a node in the heap. Since there are 'n' number of nodes, therefore, the total runtime of the algorithm turns out to be O(n(log(n)), and we use the max-heapify function for each node.

Mathematically, we see that-

The first remove of a node takes log(n) time

The second remove takes log(n-1) time

The third remove takes log(n-2) time

and so on till the last node, which will take log(1) time

So summing up all the terms, we get-

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

as log(x) + log(y) = log(x * y), we get

$=\log(n*(n-1)*(n-2)*...*2*1)$

$=\log(n!)$

Upon further simplification (using Stirling's approximation), log(n!) turns out to be

$=n*\log(n)-n+O(\log(n))$

Taking into account the highest ordered term, the total runtime turns out to be O(n(log(n)).

COMPLEXITY(ITERATIVE)

- Worst Case Time Complexity = O(n logn)
- Average Case Time Complexity = O(n logn)
- Best Case Time Complexity = O(n logn)
- Space Complexity = O(1)

COMPLEXITY(RECURSIVE)

- Worst Case Time Complexity = O(n logn)
- Average Case Time Complexity = O(n logn)
- Best Case Time Complexity = O(n logn)
- Space Complexity = O(1)

6. TIME COMPLEXITY

A. BEST CASE TIME COMPLEXITY

The best case for heapsort would happen when all elements in the list to be sorted are identical. It occurs when there is no sorting required, i.e. the array is already sorted. The best-case time complexity of heap sort is **O(n logn).**

B. AVERAGE CASE TIME COMPLEXITY

In terms of total complexity, we already know that we can create a heap in O(n) time and do insertion/removal of nodes in O(log(n)) time. In terms of average time, we need to take into account all possible inputs, distinct elements or otherwise. If the total number of nodes is 'n', in such a case, the max-heapify function would need to perform:

log(n)/2 comparisons in the first iteration (since we only compare two values at a time to build max-heap)
log(n-1)/2 in the second iteration
log(n-2)/2 in the third iteration
and so on
So mathematically, the total sum would turn out to be-

(log(n))/2 + (log(n-1))/2 + (log(n-2))/2 + (log(n-3))/2 + ...
Upon approximation, the final result would be
=1/2(log(n!))
=1/2(n∗log(n)−n+O(log(n)))

Considering the highest ordered term, the average runtime of max-heapify would then be O(n(log(n)).

Since we call this function for all nodes in the final heapsort function, the runtime would be (n * O(n(log(n))). Calculating the average, upon dividing by n, we'd get a final average runtime of **O(n logn)**

C. WORST CASE TIME COMPLEXITY

The worst case for heap sort might happen when all elements in the list are distinct. Therefore, we would need to call max-heapify every time we remove an element. In such a case, considering there are 'n' number of nodes-

The number of swaps to remove every element would be log(n), as that is the max height of the heap
Considering we do this for every node, the total number of moves would be n * (log(n)).
Therefore, the runtime in the worst case will be **O(n logn).**

7. SPACE COMPLEXITY

Since heapsort is an in-place designed sorting algorithm, the space requirement is constant and therefore, O(1). This is because, in case of any input-

- We arrange all the list items in place using a heap structure
- We put the removed item at the end of the same list after removing the max node from the max-heap.

Therefore, we don't use any extra space when implementing this algorithm. This gives the algorithm a space complexity of **O(1).**

8. UNIQUE CHARACTERISTICS

- No quadratic worst-case run time.
- It is an in-place sorting algorithm and performs sorting in O(1) space complexity.
- Compared to quicksort, it has a better worst-case time complexity — O(nlog n).
- The best-case complexity is the same for both quick sort and heap sort — O(nlog n).
- Unlike merge sort, it does not require extra space.
- The input data being completely or almost sorted doesn't make the complexities suffer.

9. APPLICATIONS

- Finding extremas - Heap sort can easily be used find the maxima and minimum in a given sequence of numbers.
- Job Scheduling - In Linux OS, heapsort is widely used for job scheduling of processes due to it's O(nlogn) time complexity and O(1) space complexity.

- Graph Algorithms - It can used in the implementation of priority queue for Djikstra's algorithm, Prim's algorithm and Huffmann encoding as well.
- Implementation of priority queues
- Security systems
- Embedded systems (for example, Linux Kernel)

10. OPTIMIZATION

BOTTOM-UP HEAP CONSTRUCTION

- Bottom-up Heapsort is a variant in which the heapify() method makes do with fewer comparisons through smart optimization. This is advantageous if, for example, we don't compare int primitives, but objects with a time-consuming compareTo() function.

- In the regular heapify(), we perform two comparisons on each node from top to bottom to find the largest of three elements:

- Parent node with left child
- The larger node from the first comparison with the second child

BOTTOM-UP HEAPSORT ALGORITHM

- Bottom-up Heapsort, on the other hand, only compares the two children and follows the larger child to the end of the tree ("top-down").

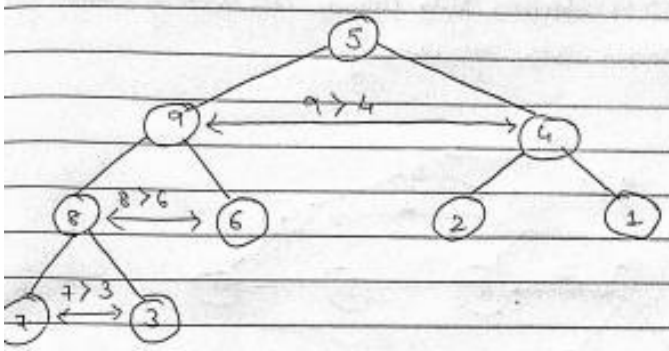
- From there, the algorithm goes back towards the tree root (“bottom-up”) and searches for the first element larger than the root.

- From this position, all elements are moved one position towards the root, and the root element is placed in the field that has become free.

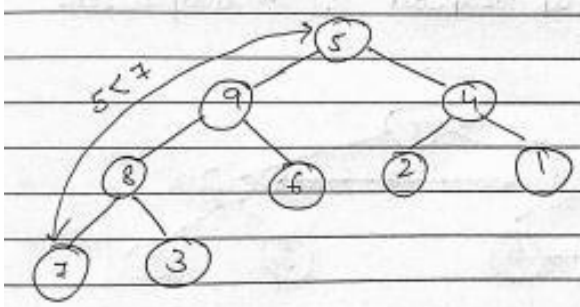
EXAMPLE-

BOTTOM UP HEAPSORT EXAMPLE.

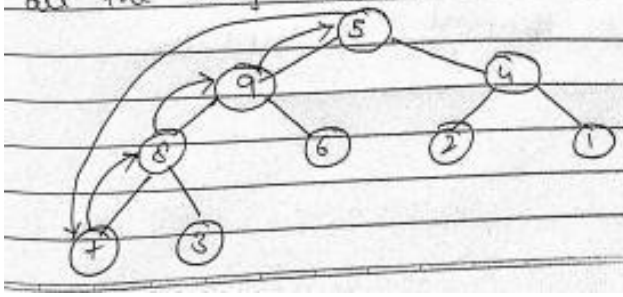
In the example, we compare 9 & 4, then children of 9 i.e 8 & 6 and children of 8 i.e 7 & 3

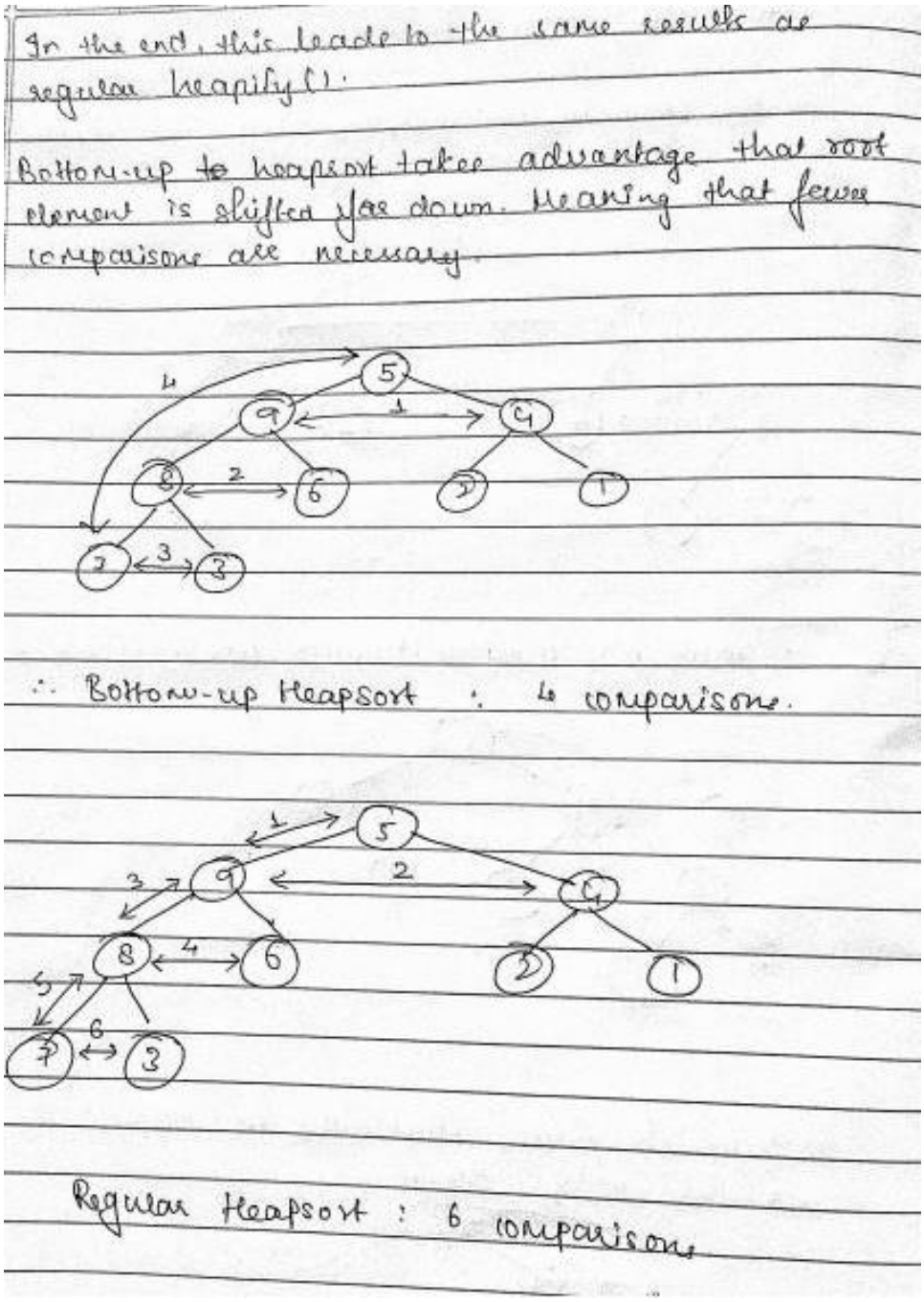


We reach 7 & compare it with tree root i.e 5



5<7, which means that the root element be passed all the way down.





11. ADVANTAGES

- The Heap sort algorithm is widely used because of its efficiency. Heap sort works by transforming the list of items to be sorted into a heap data structure, a binary tree with heap properties.
- While other sorting algorithms may grow exponentially slower as the number of items to sort increase, the time required to perform Heap

sort increases logarithmically. This suggests that Heap sort is particularly suitable for sorting a huge list of items.

- The Heap sort algorithm can be implemented as an in-place sorting algorithm. This means that its memory usage is minimal because apart from what is necessary to hold the initial list of items to be sorted, it needs no additional memory space to work.
- The Heap sort algorithm is simpler to understand than other equally efficient sorting algorithms.

12. DISADVANTAGES

- Heap sort is not a stable sort and requires constant space for sorting.
- Memory management is quite complex due to heap data structure.
- Heap sort is typically not stable since the operations on the heap can change the relative order of equal key items.
- If the input array is very large and doesn't fit into the memory and partitioning the array is faster than maintaining the heap, heap sort isn't an option. In such cases, something like merge sort or bucket sort, where parts of the array can be processed separately and parallelly, works best.

ADVANCED ALGORITHM LAB – 5

Title – BFS and DFS - Analysis and Implementation

BREADTH FIRST SEARCH (BFS)

- Breadth-first search (BFS) is an algorithm that is used to graph data or searching tree or traversing structures.
- The algorithm efficiently visits and marks all the key nodes in a graph in an accurate breadthwise fashion. This algorithm selects a single node (initial or source point) in a graph and then visits all the nodes adjacent to the selected node.

1. WORKING MECHANISM

A standard BFS implementation puts each vertex of the graph into one of two categories:

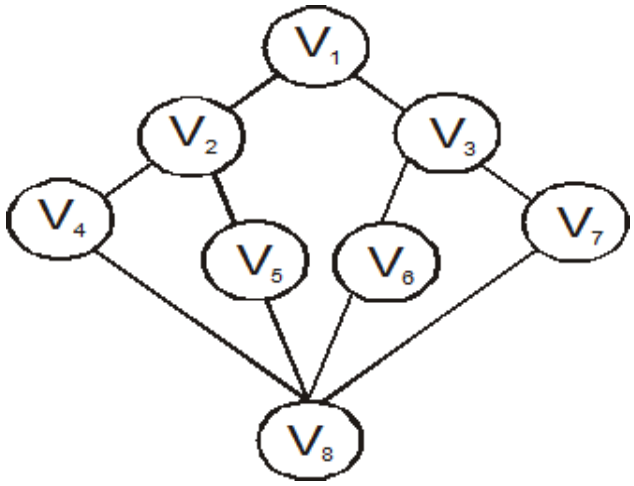
- Visited
- Not Visited

The purpose of the algorithm is to mark each vertex as visited while avoiding cycles.

The algorithm works as follows:

i. Start by putting any one of the graph's vertices at the back of a queue.
ii. Take the front item of the queue and add it to the visited list.
iii. Create a list of that vertex's adjacent nodes. Add the ones which aren't in the visited list to the back of the queue.
iv. Keep repeating steps ii and iii until the queue is empty.

EXAMPLE-



v	w	Visited	Queue
V_1		V_1	V_1
V_1	V_2 V_3	V_1 V_2 V_3	V_2 V_3
V_2	V_4 V_5	V_1V_2 V_3V_4 V_5	V_3V_4 V_5
V_3	V_6 V_7	V_1V_2 V_3V_4 V_5 V_6 V_7	V_4 V_5V_6 V_7
V_4	V_8	V_1V_2 V_3V_4 V_5 V_6 V_7 V_8	V_5 V_6 V_7 V_8
V_5	Not found	V_1V_2 V_3V_4 V_5 V_6 V_7 V_8	V_6 V_7 V_8
V_6	Not found	V_1V_2 V_3V_4 V_5 V_6 V_7 V_8	V_7 V_8
V_7	Not found	V_1V_2 V_3V_4 V_5 V_6 V_7 V_8	V_8
V_8	Not found	V_1V_2 V_3V_4 V_5 V_6 V_7 V_8	Empty

Hence, BFS = $V_1V_2V_3V_4V_5V_6V_7V_8$

2. PSEUDO CODE

ITERATIVE-

1. create a queue Q
2. mark v as visited and put v into Q
3. while Q is non-empty
4. remove the head u of Q
5. mark and enqueue all (unvisited) neighbours of u

RECURSIVE-

```
1. void recursiveBFS(Graph const &graph, queue<int> &q, vector<bool>
   &discovered)
2. {
3. if (q.empty()) {
4. return;
5. }
6. // dequeue front node and print it
7. int v = q.front();
8. q.pop();
9. cout << v << " ";
10.        // do for every edge (v, u)
11.        for (int u: graph.adjList[v])
12.        {
13.        if (!discovered[u])
14.        {
        a. // mark it as discovered and enqueue it
        b. discovered[u] = true;
        c. q.push(u);
15.        }
16.        }
17.        recursiveBFS(graph, q, discovered);
18.        }
```

3. PROGRAM CODE SNIPPET (both iterative and recursive)

ITERATIVE-

BFS_iter.cpp

```cpp
#include <iostream>
#include <queue>
using namespace std;
// tree class
class Node
{
public:
 int data;
 Node* left;
 Node* right;
 // constructor for a new tree node
 // insert the data
 // initialize the left and the right
 // subtrees with NULL
 Node(int data): data(data), left(NULL), right(NULL) {}
};
// function to build the tree
Node* build_preorder()
{
 // input an element
 int ele;
 cin >> ele;
 // base case
 if(ele == -1)
 return NULL;

 // first insert the data
 // into the root
 Node* root = new Node(ele);
 // now build the left subtree
 root->left = build_preorder();
 // similarly the right subtree
 root->right = build_preorder();
 // finally return the root
 return root;
}
// Breadth-First Search Algorithm
void BFS(Node* root)
{
 cout << "The BFS traversal of the tree is:" << endl;
 // queue to store the nodes
 queue <Node*> q;
 // pushing root and NULL into the queue
 q.push(root);
 q.push(NULL);
```

```
    q.push(NULL);
    // this while loop will run until
    // the queue becomes empty
    while(!q.empty())
    {
    // extract the front most node
    Node *f = q.front();

    q.pop();
    if(f == NULL)
    {
    // print a newline
    cout << endl;
    if(!q.empty())
    q.push(NULL);
    }
    else
    {
    cout << f->data << " ";
    if(f->left)
    q.push(f->left);
    if(f->right)
    q.push(f->right);
    }
    }
   }
   // main function to drive the program
   int main()
   {
    cout << "Enter the elements of the tree in pre-order manner" << endl;

    // building the tree
    Node *root = build_preorder();
    // calling the BFS function
    BFS(root);
   }

```

RECURSIVE-

```
#include <iostream>
#include <queue>
#include <vector>
using namespace std;

// Data structure to store a graph edge
struct Edge {
	int src, dest;
};

// A class to represent a graph object
class Graph
{
public:
	// a vector of vectors to represent an adjacency list
	vector<vector<int>> adjList;

	// Graph  Constructor
	Graph(vector<Edge> const &edges, int n)
	{
		// resize the vector to hold `n` elements of type `vector<int>`
		adjList.resize(n);

		// add edges to the undirected graph
		for (auto &edge: edges)
		{
			adjList[edge.src].push_back(edge.dest);
			adjList[edge.dest].push_back(edge.src);
		}
```

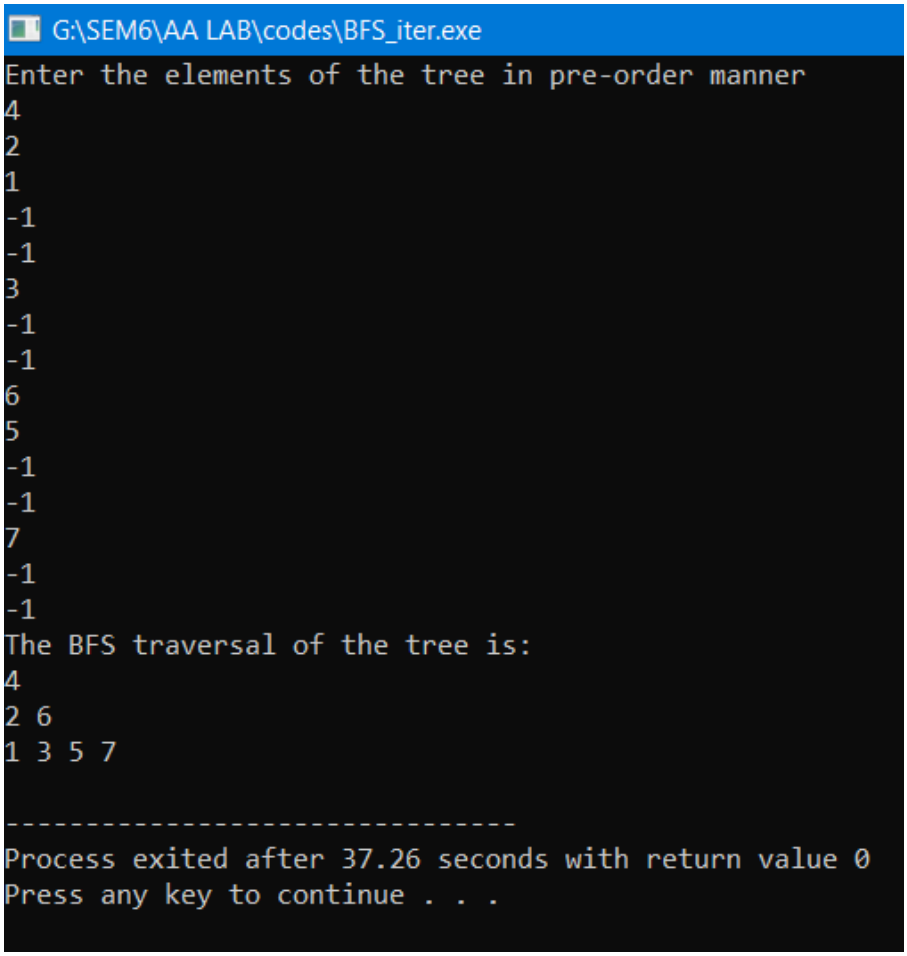
```
        }
};

// Perform BFS recursively on the graph
void recursiveBFS(Graph const &graph, queue<int> &q, vector<bool>
&discovered)
{
        if (q.empty()) {
                return;
        }

        // dequeue front node and print it
        int v = q.front();
        q.pop();
        cout << v << " ";

        // do for every edge (v, u)
        for (int u: graph.adjList[v])
        {
                if (!discovered[u])
                {
                        // mark it as discovered and enqueue it
                        discovered[u] = true;
                        q.push(u);
                }
        }

        recursiveBFS(graph, q, discovered);
}
```

```
int main()
{
	// vector of graph edges as per the above diagram
	vector<Edge> edges = {
		{1, 2}, {1, 3}, {1, 4}, {2, 5}, {2, 6}, {5, 9},
		{5, 10}, {4, 7}, {4, 8}, {7, 11}, {7, 12}
		// vertex 0, 13, and 14 are single nodes
	};

	// total number of nodes in the graph (labelled from 0 to 14)
	int n = 15;

	// build a graph from the given edges
	Graph graph(edges, n);

	// to keep track of whether a vertex is discovered or not
	vector<bool> discovered(n, false);

	// create a queue for doing BFS
	queue<int> q;

	// Perform BFS traversal from all undiscovered nodes to
	// cover all connected components of a graph
	for (int i = 0; i < n; i++)
	{
		if (discovered[i] == false)
		{
			// mark the source vertex as discovered
			discovered[i] = true;
```

```
                // enqueue source vertex
                q.push(i);

                // start BFS traversal from vertex `i`
                recursiveBFS(graph, q, discovered);
            }
        }

        return 0;
}
```

4. O/P SNIPPET (both iterative and recursive)

ITERATIVE-



RECURSIVE-

5. ANALYSIS AND ITS PROOF –

EXPLANATION 1

If V is the number of vertices and E is the number of edges of a graph, then the time complexity for BFS can be expressed as O (|V|+|E|). Having said this, it also depends on the data structure that we use to represent the graph.

If we use the adjacency list (like in our implementation), then the time complexity is O (|V|+|E|).

If we use the adjacency matrix, then the time complexity is O (V^2).

Apart from the data structures used, there is also a factor of whether the graph is densely populated or sparsely populated.

When the number of vertices exceeds the number of edges, then the graph is said to be sparsely connected as there will be many disconnected vertices. In this case, the time complexity of the graph will be O (V).

On the other hand, sometimes the graph may have a higher number of edges than the number of vertices. In such a case, the graph is said to be densely populated. The time complexity of such a graph is O (E).

To conclude, what the expression O (|V|+|E|) means is depending on whether the graph is densely or sparsely populated, the dominating factor i.e. edges or vertices will determine the time complexity of the graph accordingly.

EXPLANATION 2

Let's consider analyzing the running time of the BFS algorithm on an input graph G= (V, E). After initialization, the breadth-first search never whitens a vertex and thus ensures that each vertex is enqueued and dequeued at most once. Time taken by each enqueue and dequeue operations is O(1), and so the total time devoted to queue operations is O(V). The algorithm scans the adjacency list of every vertex at most once and when the vertex is dequeued. The sum of the lengths of all the adjacency lists account to (E), the time spent to scan the adjacency lists is O(E). The overhead for initialization is calculated to be O(V), and thus the total running time complexity of the BFS algorithm is O (V + E).

So, the algorithm runs in linear time concerning the size of the adjacency list representation of graph G.

COMPLEXITY(ITERATIVE)

- Time Complexity using adjacency list = O(V+E)
- Time Complexity using adjacency matrix = O(V^2)
- Space Complexity = O(V)

COMPLEXITY(RECURSIVE)

- Time Complexity using adjacency list = O(V+E)
- Time Complexity using adjacency matrix = O(V^2)
- Space Complexity = O(V)

6. TIME COMPLEXITY

The time complexity of BFS if the entire tree is traversed is O(V) where V

is the number of nodes.

If the graph is represented as adjacency list:

- Here, each node maintains a list of all its adjacent edges. Let's assume that there are V number of nodes and E number of edges in the graph. For each node, we discover all its neighbors by traversing its adjacency list just once in linear time.
- For a directed graph, the sum of the sizes of the adjacency lists of all the nodes is E. So, the time complexity in this case is O(V) +O(E) = **O(V + E).**

- For an undirected graph, each edge appears twice. Once in the adjacency list of either end of the edge. The time complexity for this case will be O(V) + O (2E) = O (V + E).

If the graph is represented as an adjacency matrix (a V x V array):

- For each node, we will have to traverse an entire row of length V in the matrix to discover all its outgoing edges.
- Note that each row in an adjacency matrix corresponds to a node in the graph, and that row stores information about edges emerging from the node. Hence, the time complexity of BFS in this case is O(V * V) = **O(V^2).**

• The time complexity of BFS actually depends on the data structure being used to represent the graph

7. SPACE COMPLEXITY

We use two data structures in Breadth-First Search - the visited array/Set, and the Queue.
A Visited array will have the size of the number of vertices in the graph (Similarly, the maximum size of a Set can also be up to the number of vertices in the graph).

Queue carries the vertices, as it explores them, and the maximum nodes it can have is again as many as the vertices in the Graph. E.g. If the starting vertex is 0 and all other vertices are connected to 0, then when we explore 0, we add all other vertices(as those are neighbors of 0) to Queue. So, the Queue will carry V-1 vertices in the worst case.

Since both the Visited array and Queue can have a max size of V(equal to as many vertices), the overall space complexity will be **O(V)**.

8. UNIQUE CHARACTERISTICS

- A queue (FIFO-First in First Out) data structure is used by BFS.
- You mark any node in the graph as root and start traversing the data from it.
- BFS traverses all the nodes in the graph and keeps dropping them as completed.
- BFS visits an adjacent unvisited node, marks it as done, and inserts it into a queue.
- Removes the previous vertex from the queue in case no adjacent vertex is found.
- BFS algorithm iterates until all the vertices in the graph are successfully traversed and marked as completed.

- There are no loops caused by BFS during the traversing of data from any node.

9. APPLICATIONS

- **Un-weighted Graphs:** BFS algorithm can easily create the shortest path and a minimum spanning tree to visit all the vertices of thegraph in the shortest time possible with high accuracy.
- **P2P Networks:** BFS can be implemented to locate all the nearest or neighboring nodes in a peer to peer network. This will find the required data faster.
- **Web Crawlers:** Search engines or web crawlers can easily build multiple levels of indexes by employing BFS. BFS implementation starts from the source, which is the web page, and then it visits all thelinks from that source.
- **Navigation Systems:** BFS can help find all the neighboring locations from the main or source location.
- **Network Broadcasting:** A broadcasted packet is guided by the BFS algorithm to find and reach all the nodes it has the address for.

10. OPTIMIZATION

- The map representation
  Assume one map representation. There are many ways to do a map representation and the way it is done could impact the usefulness and speed of the various BFS versions.

  ```
  int[] mapConnections = new int[WIDTH * HEIGHT];
  int indexToMapElement = x + WIDTH * y;
  ```

- Adding the children
  We need to add reachable neighbors to the queue. We do this with the following method. There are 4 possible neighbors. Let's look at the first part specifically: this is the top neighbor. We first create an index to the map for this neighbor. It works the same way as I showed before, only now the y coordinate is 1 lower (y-1). We check if the y is larger than 0, because otherwise we're at the upper boundary of the maze. We also check if the connection is allowed for this particular tile. The "8" means "1000" which means the top bit is set and we can connect to the top neighbor. We also check if the visited hash already contains the neighbor. In that case, we can't add it.

```
public void AddChildren()
{
int index = x + WIDTH * (y - 1);
if (y > 0 && (connections & 8) > 0 && !visitedHash.Contains(index))
{
int nConnections =  mapConnections[index];
if ((nConnections & 2) > 0)
{
queue.Enqueue(new Node(x, y - 1, distance + 1, nConnections, this));
visitedHash.Add(index);
}
}
```

- BFS without new (no garbage)
  Another way is to create a global array with enough nodes in it that you keep reusing every time you need to do a BFS. In this way you don't create any garbage for the garbage collector to clean up. Instead of a constructor we have a "set" method and this slightly changes our Add children method as well.

  ```
  static readonly Node[] nodes = new Node[1000];
  static int nodeIndex = 0;
  ```

- BFS without queue
  The queue is an essential part of BFS. Using the queue data structure clearly shows your intent. However, a queue is slower than an array. Moreover, you now have an array + a queue. What if you could do both with the same array? We need to make the following changes:

  Instead of:
  ```
  static int nodeIndex = 0;
  ```

  We'll have:
  ```
  static int queueCount = 0;
  static int queueIndex = 0;
  ```

  The SetChildren method also changes:

  It now has:
  ```
  Node child = nodes[queueCount++];
  ```

  instead of:
  ```
  Node child = nodes[nodeIndex++];
  ```

  We no longer "Enqueue" because we no longer have a queue. If you make this change, you will increase your performance by 50% or so.

11. ADVANTAGES

- The solution will definitely found out by BFS If there is some solution.
- BFS will never get trapped in a blind alley, which means unwanted nodes.
- If there is more than one solution then it will find a solution with minimal steps.
- It does not follow a single unfruitful path for a long time. It finds the minimal solution in case of multiple paths.
- The algorithm is optimal (i.e., admissible) if all operators have the same cost. Otherwise, breadth-first search finds a solution with the shortest path length.

12. DISADVANTAGES

- Memory Constraints As it stores all the nodes of the present level to go for the next level.
- If a solution is far away then it consumes time
- The breadth-first search algorithm cannot be effectively used unless the search space is quite small.

DEPTH FIRST SEARCH (BFS)

The main idea of the DFS Algorithm is to go as deep as possible in the graph and to come back (backtrack) when there is no unvisited vertex remaining such that it is adjacent to the current vertex.

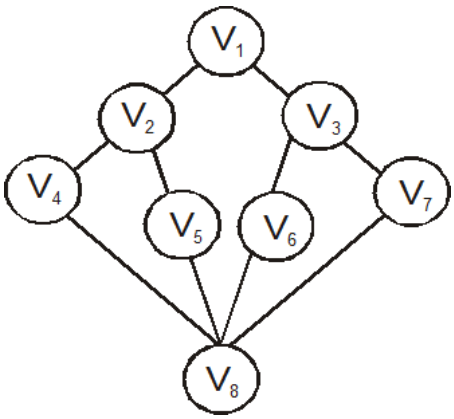
It algorithm starts with the initial node of the graph G, and then goes to deeper and deeper until we find the goal node or the node which has no children. The algorithm, then backtracks from the dead end towards the most recent node that is yet to be completely unexplored.

The data structure which is being used in DFS is stack. The process is similar to BFS algorithm. In DFS, the edges that leads to an unvisited node are called discovery edges while the edges that leads to an already visited node are called block edges.

1. WORKING MECHANISM

Step 1: SET STATUS = 1 (ready state) for each node in G
Step 2: Push the starting node A on the stack and set its STATUS = 2 (waiting state)
Step 3: Repeat Steps 4 and 5 until STACK is empty
Step 4: Pop the top node N. Process it and set its STATUS = 3 (processed state)
Step 5: Push on the stack all the neighbors of N that are in the ready state (whose STATUS = 1) and set their
STATUS = 2 (waiting state)
[END OF LOOP]
Step 6: EXIT

EXAMPLE-



v	w	Visited	Stack	
V_1		V_1	V_1	
V_1	V_2	V_1 V_2	V_1 V_2	
V_2	V_4	V_1V_2 V_4	V_1V_2 V_4	
V_4	V_8	V_1V_2 V_4 V_8	V_1V_2 V_4 V_8	
V_8	V_5	V_1V_2 V_4 V_8 V_5	V_1V_2 V_4 V_8 V_5	
V_5	Not found	V_1V_2 V_4 V_8 V_5	V_1V_2 V_4	Pop,pop
V_8	V_6	V_1V_2 V_4 V_8 V_5 V_6	V_1V_2 $v4$ V_6	
V_6	V_3	V_1V_2 V_4 V_8 V_5 V_6 V_3	V_1V_2 V_4 V_6 V_3	
V_3	V_7	V_1V_2 V_4 V_8 V_5 V_6 V_3 V_7	V_1V_2 V_4 V_6 V_3 V_7	
V_7	Not found	V_1V_2 V_4 V_8 V_5 V_6 V_3 V_7	V_1V_2 V_4 V_6 V_3	Pop
V_7	Not found	V_1V_2 V_4 V_8 V_5 V_6 V_3 V_7	V_1V_2 V_4 V_6	Pop
V_3	Not found	V_1V_2 V_4 V_8 V_5 V_6 V_3 V_7	V_1V_2 V_4	Pop
V_6	Not found	V_1V_2 V_4 V_8 V_5 V_6 V_3 V_7	V_1V_2	Pop
V_4	Not found	V_1V_2 V_4 V_8 V_5 V_6 V_3 V_7	V_1	Pop
V_2	Not found	V_1V_2 V_4 V_8 V_5 V_6 V_3 V_7	Empty	Pop
V_1	Not found	V_1V_2 V_4 V_8 V_5 V_6 V_3 V_7	Empty	

Hence, DFS = $V_1V_2V_3V_4V_5V_6V_7V_8$

2. PSEUDO CODE

ITERATIVE-

```
1. DFS-iterative (G, s): //Where G is graph and s is source
2. vertex
3. let S be stack
4. S.push( s ) //Inserting s in stack
5. mark s as visited.
6. while ( S is not empty):
7. //Pop a vertex from stack to visit next
8. v = S.top( )
9. S.pop( )
10. //Push all the neighbours of v in stack that are not visited
11.for all neighbours w of v in Graph G:
12.if w is not visited :
13.S.push( w )
14.mark w as visited
```

RECURSIVE-

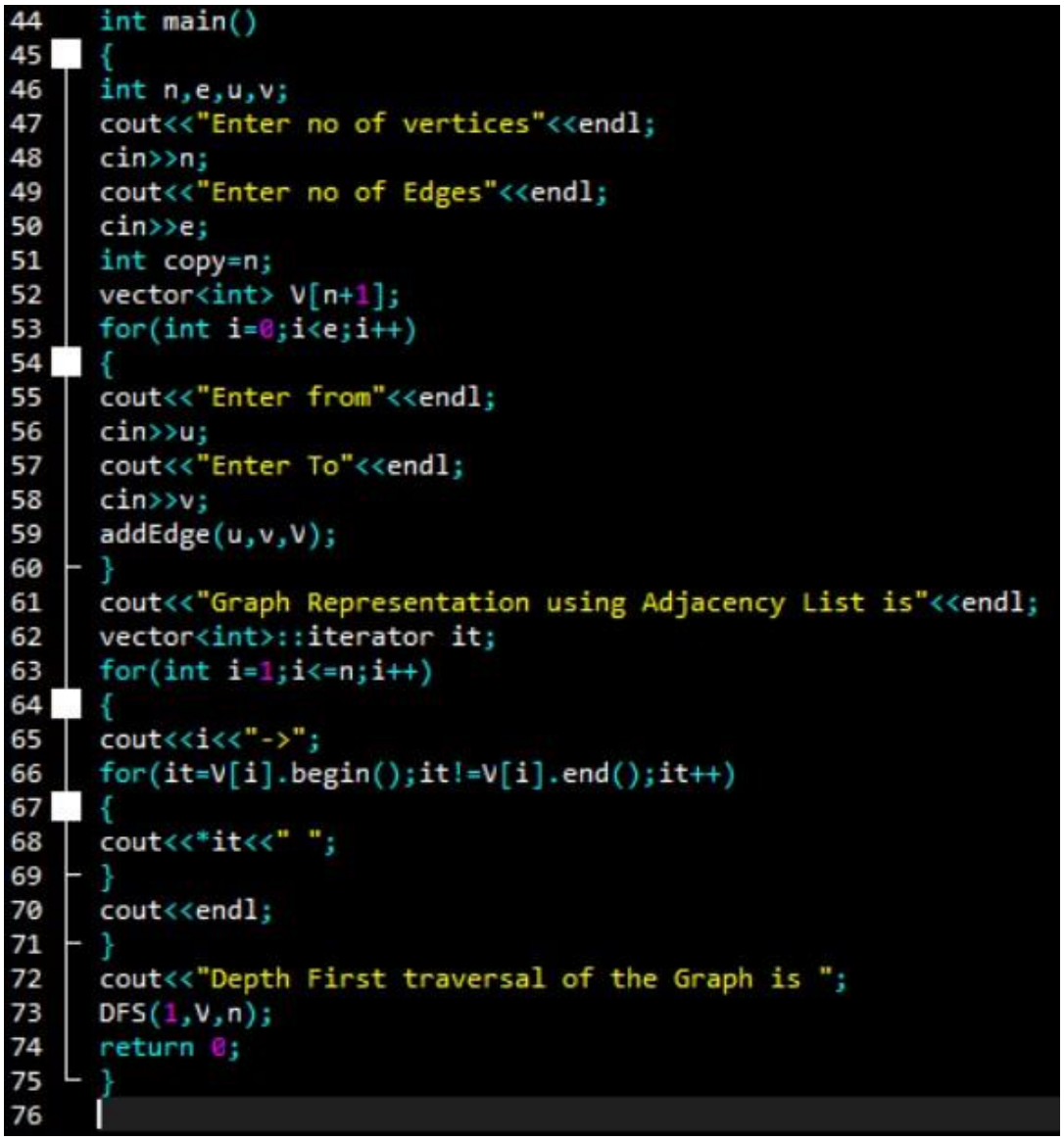
```
1. mark s as visited
2. for all neighbours w of s in Graph G:
3. if w is not visited:
4. DFS-recursive(G, w)
```

3. PROGRAM CODE SNIPPET (both iterative and recursive)

ITERATIVE-

DFS_iter.cpp

```cpp
#include<iostream>
#include<vector>
#include<stack>
using namespace std;
void addEdge(int u,int v,vector<int>* V)
{
V[u].push_back(v);
}
void DFS(int s,vector<int>* adj,int n)
{
// Initially mark all verices as not visited
 int visited[n+1]={0};

 // Create a stack for DFS
 stack<int> stack;

 // Push the current source node.
 stack.push(s);

 vector<int>::iterator i;
 while (!stack.empty())
 {
 // Pop a vertex from stack and print it
 s = stack.top();
 stack.pop();

 // Stack may contain same vertex twice. So
 // we need to print the popped item only
 // if it is not visited.
 if (!visited[s])
 {
 cout << s << " ";
 visited[s] = true;
 }

 // Get all adjacent vertices of the popped vertex s
 // If a adjacent has not been visited, then push it
 // to the stack.
 for (i = adj[s].begin(); i != adj[s].end(); ++i)
 if (!visited[*i])
 stack.push(*i);
 }
}
int main()
```



```
int main()
{
int n,e,u,v;
cout<<"Enter no of vertices"<<endl;
cin>>n;
cout<<"Enter no of Edges"<<endl;
cin>>e;
int copy=n;
vector<int> V[n+1];
for(int i=0;i<e;i++)
{
cout<<"Enter from"<<endl;
cin>>u;
cout<<"Enter To"<<endl;
cin>>v;
addEdge(u,v,V);
}
cout<<"Graph Representation using Adjacency List is"<<endl;
vector<int>::iterator it;
for(int i=1;i<=n;i++)
{
cout<<i<<"->";
for(it=V[i].begin();it!=V[i].end();it++)
{
cout<<*it<<" ";
}
cout<<endl;
}
cout<<"Depth First traversal of the Graph is ";
DFS(1,V,n);
return 0;
}
```

RECURSIVE-

DFS_rec.cpp

```cpp
#include <iostream>
#include <list>
#include <stack>
#include <iterator>
#define Max 10 ;
using namespace std;
class Vertex{
public:
 list<Vertex*> neighbors;
 char name;
 bool visited;
 Vertex(): name{'\0'},visited{false}{}
 Vertex(char ch): name{ch},visited{false}{}
 void addNeighbor(Vertex *v){
 neighbors.push_back(v);
 }
};
class DFS{
    public:
    void dfs(Vertex *currentVertex){
     cout << currentVertex->name <<" ";
     list<Vertex*> nbr = currentVertex->neighbors;
     list<Vertex*> :: iterator it;
     for(it=nbr.begin(); it != nbr.end();it++ ){
         if(!(*it)->visited){
             (*it)->visited = true;
             dfs(*it);
         }
     }
    }
};
int main(){
 //Adding vertex to the graph
 Vertex v1('A'), v2('B'),v3('C'),v4('D'),v5('E');
 //Connecting vertex OR assigning neighbor
 v1.addNeighbor(&v2);
 v1.addNeighbor(&v4);
 v4.addNeighbor(&v5);
 v2.addNeighbor(&v3);
 DFS b;
 //Assuming Vertex v1 as root
 b.dfs(&v1);
 return 0;
}
```

4. O/P SNIPPET (both iterative and recursive)

ITERATIVE-

```
G:\SEM6\AA LAB\codes\DFS_iter.exe
Enter no of vertices
5
Enter no of Edges
4
Enter from
1
Enter To
2
Enter from
1
Enter To
3
Enter from
2
Enter To
4
Enter from
2
Enter To
5
Graph Representation using Adjacency List is
1->2 3
2->4 5
3->
4->
5->
Depth First traversal of the Graph is 1 3 2 5 4
--------------------------------
Process exited after 23.74 seconds with return value 0
Press any key to continue . . .
```

RECURSIVE-

```
G:\SEM6\AA LAB\codes\DFS_rec.exe
A B C D E
--------------------------------
Process exited after 0.0387 seconds with return value 0
Press any key to continue . . .
```

5. ANALYSIS AND ITS PROOF –

Runtime for the standard DFS algorithm.

```
DFS(G) \\ DFS on an entire graph G
init(G);
t ← 1
foreach v ∈ G do
if color(v) = white then
t ← DFS(v, t)
t++
```

Everything above the loop runs in O (1) time per node visit. Excluding the recursive call, everything inside of the for loop takes O (1) time every time anedge is scanned. Everything after the for loop also runs in O (1) time per node visit. We can express the runtime of DFS as O (of node visits + of edge scans). Assume we have a graph with n nodes and m edges. We know that the # of node visits is ≤ n, since we only visit white nodes and whenever we visit a node we change its color from white to grey and never change it back to white again.

We also know that an edge (u, v) is scanned only when u or v is visited. Since every node is visited at most once, we know that an edge (u, v) is scanned at most twice (or only once for directed graphs). Thus, of edges scanned is O(m), and the overall runtime of DFS is O (m + n).

- If we represent the graph G by adjacency matrix then the running time of DFS algorithm is O (n),
  where n is the number of nodes.
- If we represent the graph G by link lists then the running time of DFS algorithm is O (m + n),
  where m is the number of edges and n is the number of nodes.

COMPLEXITY(ITERATIVE)

- Time Complexity using adjacency list = O(V+E)
- Time Complexity using adjacency matrix = O(V^2)
- Space Complexity = O(V)

COMPLEXITY(RECURSIVE)

- Time Complexity using adjacency list = O(V+E)
- Time Complexity using adjacency matrix = O(V^2)
- Space Complexity = O(V)

6. TIME COMPLEXITY

The time complexity of DFS if the entire tree is traversed is O(V) where V

is the number of nodes.

If the graph is represented as adjacency list:

- Here, each node maintains a list of all its adjacent edges. Let's assume that there are V number of nodes and E number of edges in the graph.
- For each node, we discover all its neighbors by traversing its adjacency list just once in linear time.
- For a directed graph, the sum of the sizes of the adjacency lists of all the nodes is E. So, the time complexity in this case is O(V) + O(E) = **O (V + E).**
- For an undirected graph, each edge appears twice. Once in the adjacency list of either end of the edge. The time complexity for this case will be O(V) + O (2E) = O (V + E).

If the graph is represented as an adjacency matrix (a V x V array):

- For each node, we will have to traverse an entire row of length V in the matrix to discover all its outgoing edges.
- Note that each row in an adjacency matrix corresponds to a node inthe graph, and that row stores information about edges emerging fromthe node. Hence, the time complexity of DFS in this case is **O (V * V) = O(V^2).**

• The time complexity of DFS actually depends on the data structure being

used to represent the graph

7. SPACE COMPLEXITY

Since we are maintaining a stack to keep track of the last visited node, in worst case, the stack could take up to the size of the nodes(or vertices) in thegraph. Hence, the space complexity is **O(V).**

8. UNIQUE CHARACTERISTICS

- Depth First Search or DFS is a graph traversal algorithm.
- It is used for traversing or searching a graph in a systematic fashion.
- DFS uses a strategy that searches "deeper" in the graph whenever possible.
- Stack data structure is used in the implementation of depth first search.

9. APPLICATIONS

- Depth-first search is used in topological sorting, scheduling problems, cycle detection in graphs, and solving puzzles with only one solution, such as a maze or a sudoku puzzle.
- Other applications involve analyzing networks, for example, testing if a graph is bipartite. Depth-first search is often used as a subroutine in network flow algorithms such as the Ford-Fulkerson algorithm.
- DFS is also used as a subroutine in matching algorithms in graph theory such as the Hopcroft–Karp algorithm.
- Depth-first searches are used in mapping routes, scheduling, and finding spanning trees.
- Finding strongly connected components in the directed graph.
- To check if the given graph is Bipartite in nature.

10. OPTIMIZATION

Depth-First Branch-and-Bound (DFBB)

- DFS technique in which upon finding a solution, the algorithm updates current best solution.
- DFBB does not explore paths that are guaranteed to lead to poorer solutions than the current best solution.
- On termination, the current best solution is a globally optimal solution.

Iterative Deepening Search

- Often, the solution may exist close to the root, but on an alternative branch.
- Simple backtracking might explore a large space before finding this.
- Iterative deepening sets a depth bound on the space it searches (using DFS).
- If no solution is found, the bound is increased and the process repeated.

11. ADVANTAGES

- DFS consumes very less memory space.
- It will reach at the goal node in a less time period than BFS if it traverses in a right path.
- It may find a solution without examining much of search because we may get the desired solution in the very first go.
- The memory requirement is Linear WRT Nodes.

12. DISADVANTAGES

- It is possible that may states keep reoccurring. There is no guarantee of finding the goal node.
- Sometimes the states may also enter into infinite loops.
- May find a sub-optimal solution (one that is deeper or more costly than the best solution)

DIFFERENCE BETWEEN BFS AND DFS

Sr. No.	Key	BFS	DFS
1	Definition	BFS, stands for Breadth First Search.	DFS, stands for Depth First Search.
2	Data structure	BFS uses Queue to find the shortest path.	DFS uses Stack to find the shortest path.
3	Source	BFS is better when target is closer to Source.	DFS is better when target is far from source.
4	Suitabilit yfor decision tree	As BFS considers all neighbor so it is not suitable for decision tree used in puzzle games.	DFS is more suitable for decision tree. As with one decision, we need to traverse further to augment the decision. If we reach the conclusion, we won.
5	Speed	BFS is slower than DFS.	DFS is faster than BFS.
6	Time Complexity	Time Complexity of BFS = O(V+E) where V is vertices and E is edges.	Time Complexity of DFS is also O(V+E) where V is vertices and E is edges.

THANK YOU !

ADVANCED ALGORITHM LAB – 6

Title - Analysis and Implementation of Prim's and Kruskal's MST Algorithm

PRIM'S MST

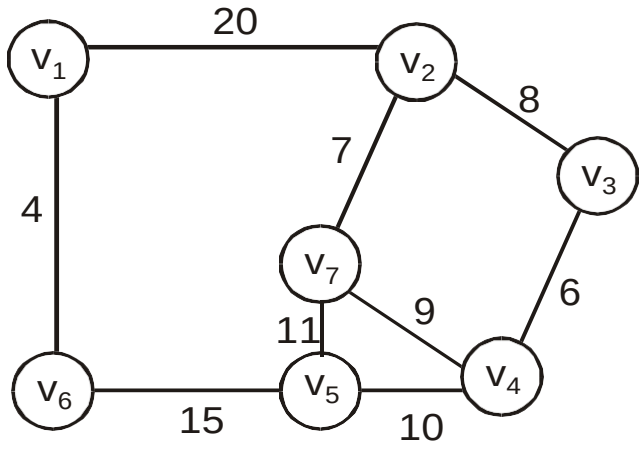
1. WORKING MECHANISM

Prim's technique works on the simple premise that all vertices in a spanning tree must be connected. To build a Spanning Tree, the two distinct subsets of vertices must be joined. To make it a Minimum Spanning Tree, they must be connected with the minimum weight edge.

ALGORITHM

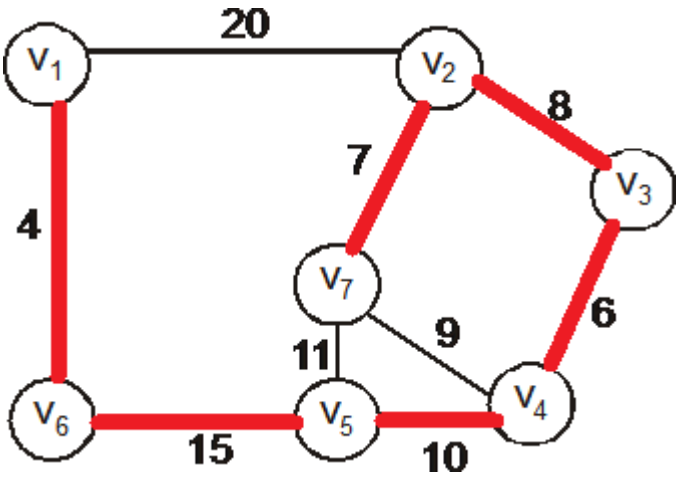
Step 1: Pick a starting vertex.
Step 2: Repeat Steps 3 and 4 until all of the vertices are fringed.
Step 3: Select an edge 'e' connecting the tree vertex and fringe vertex that has minimum weight
Step 4: To the minimum spanning tree T, add the selected edge and vertex.
[THE END OF THE LOOP]
Step 5: EXIT

EXAMPLE-



Step	Consider	Select	Comments
1.	(V_1, V_6)	(V_1, V_6)	Tree
2.	(V_1, V_2), (V_5, V_6)	(V_5, V_6)	Tree
3.	(V_1, V_2), (V_5, V_7), (V_4, V_5)	(V_4, V_5)	Tree
4.	(V_1, V_2), (V_5, V_7), (V_3, V_4), (V_4, V_7)	(V_3, V_4)	Tree
5.	(V_1, V_2), (V_5, V_7), (V_4, V_7), (V_2, V_3)	(V_2, V_3)	Tree
6.	(V_1, V_2), (V_5, V_7), (V_4, V_7), (V_2, V_7)	(V_2, V_7)	Tree
7.	(V_1, V_2), (V_5, V_7), (V_4, V_7)	(V_4, V_7)	Forms a cycle

8.	(V_5, V_7), (V_1, V_2)	(V_5, V_7)	Forms a cycle
9.	(V_1, V_2)	(V_1, V_2)	Forms a cycle



TOTAL COST = 50

2. PSEUDO CODE

```
Prim()
        S = new empty set
        for i = 1 to n
                d[i] = inf
        while S.size() < n
                x = inf
                v = -1
                for each i in V - S // V is the set of vertices
                        if x >= d[v]
                                then x = d[v], v = i
                d[v] = 0
                S.insert(v)
                for each u in adj[v]
                        do d[u] = min(d[u], w(v,u))
```

3. PROGRAM CODE SNIPPET

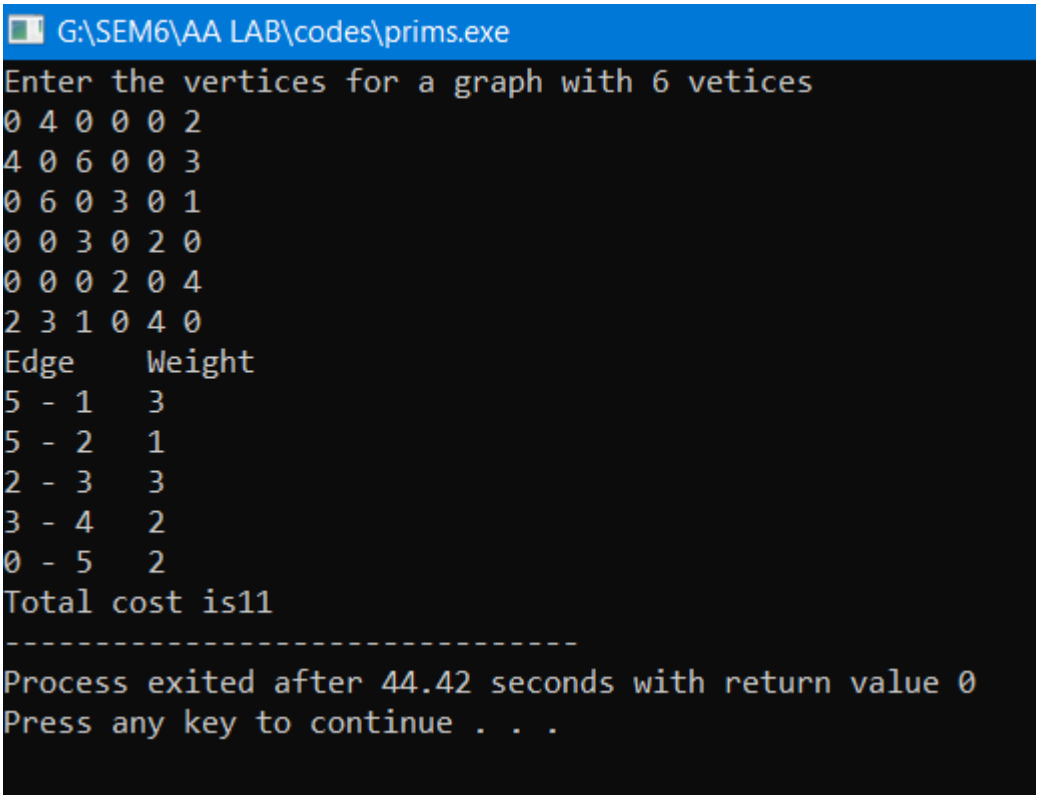
prims.cpp

```cpp
#include<iostream>

using namespace std;

// Number of vertices in the graph
const int V=6;

// Function to find the vertex with minimum key value
int min_Key(int key[], bool visited[])
{
    int min = 999, min_index;  // 999 represents an Infinite value

    for (int v = 0; v < V; v++) {
        if (visited[v] == false && key[v] < min) {
            // vertex should not be visited
            min = key[v];
            min_index = v;
        }
    }
    return min_index;
}

// Function to print the final MST stored in parent[]
int print_MST(int parent[], int cost[V][V])
{
    int minCost=0;
    cout<<"Edge \tWeight\n";
    for (int i = 1; i< V; i++) {
        cout<<parent[i]<<" - "<<i<<" \t"<<cost[i][parent[i]]<<" \n";
        minCost+=cost[i][parent[i]];
    }
    cout<<"Total cost is"<<minCost;
}

// Function to find the MST using adjacency cost matrix representation
void find_MST(int cost[V][V])
{
    int parent[V], key[V];
    bool visited[V];

    // Initialize all the arrays
    for (int i = 0; i< V; i++) {
        key[i] = 999;    // 99 represents an Infinite value
        visited[i] = false;
        parent[i]=-1;
```

prims.cpp

```
45          parent[i]=-1;
46      }
47
48      key[0] = 0;  // Include first vertex in MST by setting its key vaue to 0.
49      parent[0] = -1; // First node is always root of MST
50
51      // The MST will have maximum V-1 vertices
52      for (int x = 0; x < V - 1; x++)
53      {
54          // Finding the minimum key vertex from the
55          //set of vertices not yet included in MST
56          int u = min_Key(key, visited);
57          visited[u] = true;  // Add the minimum key vertex to the MST
58          // Update key and parent arrays
59          for (int v = 0; v < V; v++)
60          {
61
62              if (cost[u][v]!=0 && visited[v] == false && cost[u][v] < key[v])
63              {
64                  parent[v] = u;
65                  key[v] = cost[u][v];
66              }
67          }
68      }
69
70      // print the final MST
71      print_MST(parent, cost);
72  }
73
74  int main()
75  {
76      int cost[V][V];
77      cout<<"Enter the vertices for a graph with 6 vetices";
78      for (int i=0;i<V;i++)
79      {
80          for(int j=0;j<V;j++)
81          {
82              cin>>cost[i][j];
83          }
84      }
85      find_MST(cost);
86
87      return 0;
88  }
```

4. O/P SNIPPET



5. ANALYSIS AND ITS PROOF –

a. RUNNING TIME OF PRIM'S ALGORITHM

Let's say we have a graph with V vertices and E edges for which we need to find an MST.

We delete the min node from Min-Heap and add a number of edge weights to Min-Heap to complete one loop.
We eliminate V nodes from Min-Heap in total since we have V nodes in the graph and 1 edge is deleted in every iteration, totaling V-1 edges in MST, with each deletion taking O(log(V) complexity. And then we add all of the E edges together, with each addition having a complexity of O(log(V)).

As a result, the total complexity is O((V+E)Log(V)).

b. PRIM'S BEST AND WORST CASES

When the given graph is a tree, and each node has the fewest number of adjacent nodes, Prim's has the best time complexity.
When it comes to time complexity, the worst case scenario is a graph with V2 edges.

6. TIME COMPLEXITY

DATA STRUCTURE USED FOR THE MINIMUM EDGE WEIGHT	TIME COMPLEXITY
Adjacency matrix, linear searching	$O(\|V\|^2)$
Adjacency list and binary heap	$O(\|E\| \log \|V\|)$
Adjacency list and Fibonacci heap	$O(\|E\|+ \|V\| \log \|V\|)$

Prim's technique is simple to implement using an adjacency matrix or an adjacency list graph representation, and adding the edge with the lowest weight requires linearly searching an array of weights. It takes O(|V|2) time to run. It can be improved even further by using heap implementation to discover the minimal weight edges in the algorithm's inner loop.

The prim's algorithm has a temporal complexity of O(E logV) or O(V logV), where E is the number of edges and V is the number of vertices.

7. SPACE COMPLEXITY

To determine whether a node is in MST or not, we require an array.s (V).

To keep track of Min-Heap, we'll need an array. Space O(E).

As a result, the total space complexity is of the order of O(V+E).

8. UNIQUE CHARACTERISTICS

- Prim's algorithm finds a minimal cost spanning tree for a linked weighted undirected graph using a greedy approach.
- The technique creates a tree with all vertex and a subset of the edges in such a way that the total weight of all the edges in the tree is the smallest.
- The shortest path first algorithm is used to find the smallest spanning tree for the prims algorithm.
- To generate a minimum spanning tree, this approach adds new nodes from the Graph with the smallest edge weight.

9. APPLICATIONS

- A road and rail network that connects all of the cities.
- Irrigation channels and microwave tower placement
- Creating a fiber-optic grid or integrated circuits.
- The Problem of the Traveling Salesman
- Cluster analysis
- Pathfinding algorithms are utilised in AI in a number of ways
- Developing a Game
- Cognitive Science
- Optimization

10. OPTIMIZATION

Priority queue is provided by STL, although it does not support the decrease key operation. We also need a priority queue in Prim's algorithm, as well as the following operations on the priority queue:

ExtractMin: We need to get a vertex with the smallest key value from all the vertices that haven't been included in MST yet.
ReduceKey: After removing a vertex, we must update the keys of its surrounding vertices, and if the new key is smaller, we must update the data structure as well.

The goal is to put only those vertices in the priority queue that are not MST and have been visited by a vertex that is included in MST. In MST[, we keep track of the vertices that are included in MST in a separate boolean array.

11. ADVANTAGES

- Prim's algorithm returns a connected component and only works with connected graphs.
- In dense graphs, Prim's algorithm performs better.

12. DISADVANTAGES

- As additional edges are added, the list of edges must be searched from the beginning.
- If there are multiple edges with the same weight, all possible spanning trees must be found for the final minimum tree.

KRUSKAL'S MST

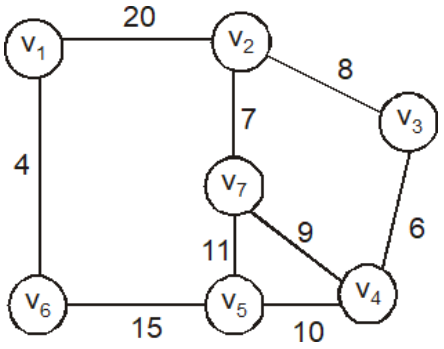
1. WORKING MECHANISM

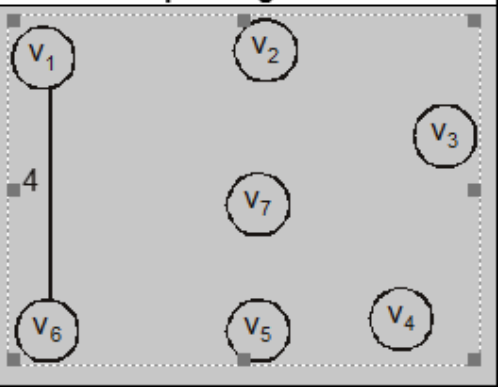
For a connected weighted graph, Kruskal's Algorithm is used to determine the smallest spanning tree. The algorithm's main goal is to locate a subset of edges that may be used to visit every vertex of the graph. Instead of focusing on a global optimum, it uses a greedy method to discover an optimal solution at each stage.

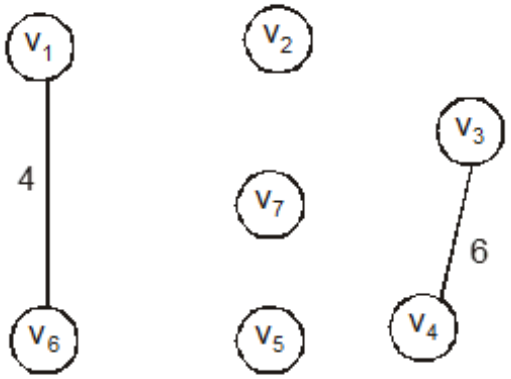
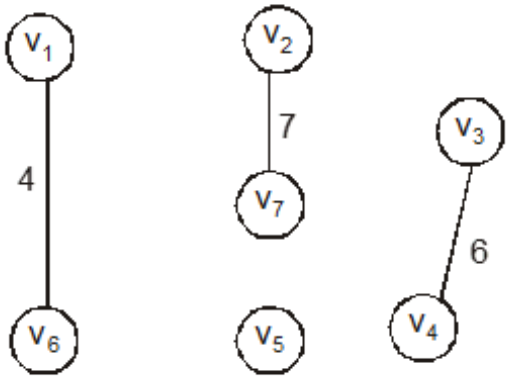
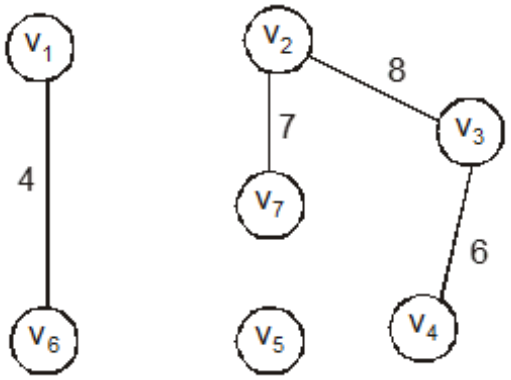
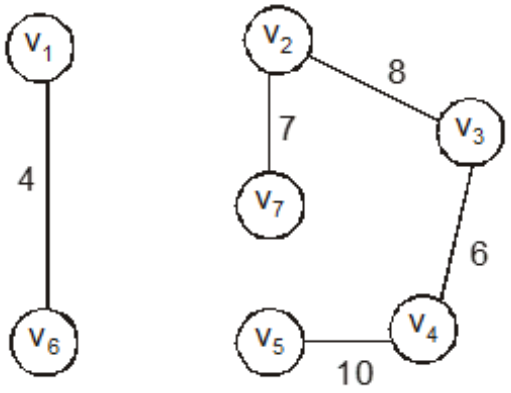
ALGORITHM

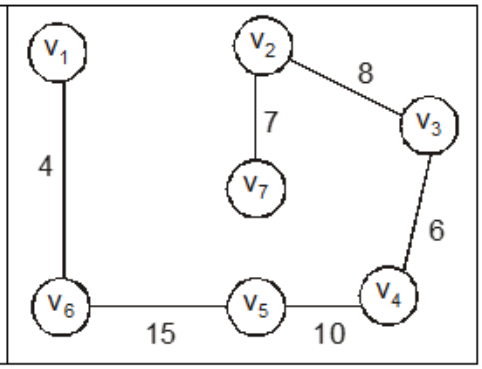
Step 1: Make a forest F with each vertex of graph representing a single tree.
Step 2: Make a set E that contains all of the graph's edges.
Step 3: While E is NOT EMPTY and F is not spanning, repeat Steps 4 and 5.
Step 4: Remove an edge from E with minimum weight
Step 5: Add the edge obtained in Step 4 to the forest F if it connects two different trees (for combining two trees into one tree).
ELSE, Delete the edge.
STEP 6: END

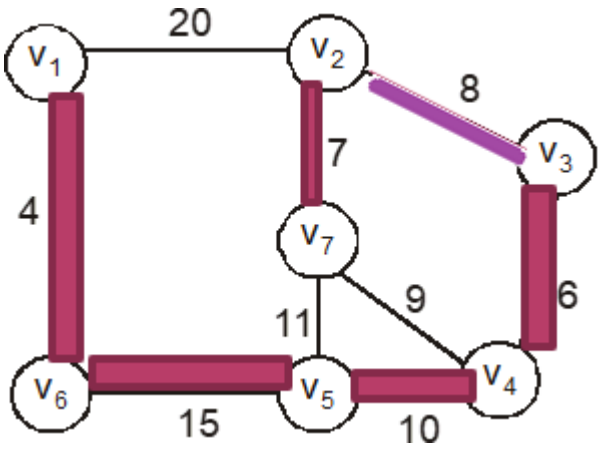
EXAMPLE-



Step	Consider	Spanning Tree
1	(V_1, V_6)	

2.	(V_3, V_4)	
3.	(V_2, V_7)	
4.	(V_2, V_3)	
5.	(V_4, V_7)	Forms a Cycle, Reject
6.	(V_4, V_5)	
7.	(V_5, V_7)	Forms a Cycle, Reject

8.	(V_5, V_6)	



TOTAL COST = 50

2. PSEUDOCODE

```
# an empty set for keeping final MST
S = { }
if edges not sorted:
    Sort edges of the given graph per ascending order
for every vertex (v) in given graph:
     create subset(v)
while number of edges < V-1:
    select edge having least weight
    if adding this edge doesn't create cycle:
        add edge to S
    increment the index to go to subsequent edge
```

return final MST created

return S

3. PROGRAM CODE SNIPPET-

kruskal.cpp

```cpp
#include <bits/stdc++.h>
using namespace std;

#define V 5
int parent[V];

// Find set of vertex i
int find(int i)
{
    while (parent[i] != i)
        i = parent[i];
    return i;
}

void union1(int i, int j)
{
    int a = find(i);
    int b = find(j);
    parent[a] = b;
}

// Finds MST using Kruskal's algorithm
void kruskalMST(int cost[][V])
{
    int mincost = 0; // Cost of min MST.

    // Initialize sets of disjoint sets.
    for (int i = 0; i < V; i++)
        parent[i] = i;

    // Include minimum weight edges one by one
    int edge_count = 0;
    while (edge_count < V - 1) {
        int min = INT_MAX, a = -1, b = -1;
        for (int i = 0; i < V; i++) {
            for (int j = 0; j < V; j++) {
                if (find(i) != find(j) && cost[i][j] < min) {
                    min = cost[i][j];
                    a = i;
                    b = j;
                }
            }
        }

        union1(a, b);
```

```
45              union1(a, b);
46              printf("Edge %d:(%d, %d) cost:%d \n",
47                  edge_count++, a, b, min);
48              mincost += min;
49          }
50          printf("\n Minimum cost= %d \n", mincost);
51      }
52
53      int main()
54      {
55
56          int cost[][V] = {
57              { INT_MAX, 2, INT_MAX, 6, INT_MAX },
58              { 2, INT_MAX, 3, 8, 5 },
59              { INT_MAX, 3, INT_MAX, INT_MAX, 7 },
60              { 6, 8, INT_MAX, INT_MAX, 9 },
61              { INT_MAX, 5, 7, 9, INT_MAX },
62          };
63
64          kruskalMST(cost);
65
66          return 0;
67      }
68
```

4. O/P SNIPPET

```
G:\SEM6\AA LAB\codes\kruskal.exe
Edge 0:(0, 1) cost:2
Edge 1:(1, 2) cost:3
Edge 2:(1, 4) cost:5
Edge 3:(0, 3) cost:6

 Minimum cost= 16

--------------------------------
Process exited after 0.02847 seconds with return value 0
Press any key to continue . . .
```

5. ANALYSIS AND ITS PROOF –

a. RUNNING TIME OF KRUSKAL'S

Let's assume we're using Kruskal's to find the MST of a network with N vertices.
To check edges, we must first sort the given edges according to their weights. The most efficient sorting method has an order of O (N log(N)).To see if an edge has to be in MST, we use Union-Find to see if it makes a circle with the edges present, then add it to MST exactly once and use the order log Union-Find method (E).
Because we do at most N checks for a graph, the overall complexity of the checks is O(Nlog(E)).

As a result, the overall complexity is O(Nlog(E)+Nlog(N)).

b. KRUSKAL'S BEST AND WORST CASES

In all circumstances, the time complexity of regular Kruskal's will be O(Nlog(E)+Nlog(N)). In the case of Kruskal's, we have N no. cycles in thebest case scenario, and we must execute N-1 iterations to calculate MST.

In this situation, the time complexity will be *O((N-1)log(E)+Elog(E)).

In the worst-case scenario, we'll have to double-check all E edges. In this example, the time complexity would be O(Elog(E)+Nlog(N)).

6. TIME COMPLEXITY

BEST CASE

In the best case, we assume the edges E are already sorted and we have a graph having minimum number of edges as possible. A graph with minimal number of edges is called a Sparse graph.

In such a case, we won't need to sort edges and the impact of edges on the rest of the algorithm will be minimal in practice but same as worst case in theory.

Another case could be when the first V-1 edges form the MST. That way, the number of operations for all edges would be minimum.

In either of the above two cases-

The make_set function would have a runtime same as the worst case, O(V).

The union and find functions would have a constant runtime in practice but Log(E) runtime in theory. Considering that we do this for all vertices and valid edges, the total complexity for each would be O(E(log(E)).

Therefore, the total time complexity in the best case becomes O(E(log(E)), since E is V-1.

AVERAGE CASE

Considering all forms of graphs we might come across as input for this algorithm-

The sorting and make_set function would have the same runtime on average as the worst, if we don't consider the constants.

In functions union and find, we either consider two vertices (parent and the current node) or two subtrees at a time. So the complexity would be a constant in practice, but (log E) in theory, divided by two on average per operation. Ignoring the constants, the runtime would be the same as the worst case.

Therefore, the total runtime on average would also be O(E(log(E)).

WORST CASE

The worst case might happen when all the edges are not sorted, the graph contains maximum number of edges, and we have included the edge with maximum weight in our MST. So in this case, we would need to check all the edges at each step to make sure our MST includes minimal weighted edges. The maximum possible number of edges in an undirected graph is given by n(n-1)/2. Ignoring constants, this results in n^2 total maximum number of edges. Such a graph having maximum number of edges is called a Dense graph. In such a case, the graph with V number of vertices would have V^2 number of edges.

- **WORST-CASE TIME COMPLEXITY = O(E(LOG(E)).**
- **AVERAGE-CASE TIME COMPLEXITY = O(E(LOG(E)).**
- **BEST-CASE TIME COMPLEXITY = O(E(LOG(E)).**

7. SPACE COMPLEXITY

The special structure that we use to keep track of all vertices, edges in a tree form called the Disjoint set structure, requires-

- A space requirement of O(V) to keep track of all vertices in the beginning and the respective subsets.
- A space requirement of O(E) to keep track of all the valid sorted edges to be included in the final MST.

Therefore, the total space complexity turns out to be O(E+V).

8. UNIQUE CHARACTERISTICS

- Kruskal's Algorithm builds a Minimum Spanning Tree for a linked, weighted, and undirected graph using the Greedy Algorithm. The graph is treated as a forest, and the vertices are treated as individual trees in this technique.
- The goal of this technique is to discover a subset of edges that builds a tree that contains every vertex with the smallest number of edges.
- Many different spanning trees can exist in a single graph. A spanning tree with a total of edges weight less than or equal to the sum of edges weight of every other spanning tree is a minimum spanning tree for a weighted, linked, and undirected graph.

9. APPLICATIONS

- Landing cables
- TV Network
- Tour Operations
- LAN Networks
- A network of pipes for drinking water or natural gas.
- An electric grid
- Single-link Cluster

10. Optimization

By following the approach, the Kruskal runtime can be reduced to O(VlogV). It takes advantage of the fact that MST (minimum spanning tree) is a subgraph of Delaunay Triangulation, and that Delaunay Triangulation can be calculated in O. (VlogV). Delaunay Triangulation's size is linear, i.e. O(V), thus the edges you'll sort are also linear (V). Kruskal can attain the time bound O(VlogV) since sorting is dominating in the algorithm.

11. Advantages

- As it uses disjoint sets and smaller data structures, Kruskal performs better in typical scenarios (sparse graphs) and is easier to implement.
- By adding the next cheapest edge to the existing tree / forest, Kruskal's Algorithm builds a solution from the cheapest edge.

12. Disadvantages

- It's possible that the minimum spanning tree isn't unique.
- The minimum spanning tree will be unique if the weights assigned to all of the graph's edges are unique. There could be a lot of minimum spanning trees if this isn't the case.

Difference between Prim's and Kruskal's

Prim's Algorithm	Kruskal's Algorithm
The tree that we are making or growing always remains connected.	The tree that we are making or growing usually remains disconnected.
Prim's Algorithm grows a solution from a random vertex by adding the next cheapest vertex to the existing tree.	Kruskal's Algorithm grows a solution from the cheapest edge by adding the next cheapest edge to the existing tree / forest.
Prim's Algorithm is faster for dense graphs.	Kruskal's Algorithm is faster for sparse graphs.

Thank you!

ADVANCED ALGORITHM LAB – 7

Title - Analysis and Implementation of Dijkstra's Algorithm

1. WORKING MECHANISM

Dijkstra's algorithm allows us to find the shortest path between any two vertices of a graph.

It differs from the minimum spanning tree because the shortest distance between two vertices might not include all the vertices of the graph.

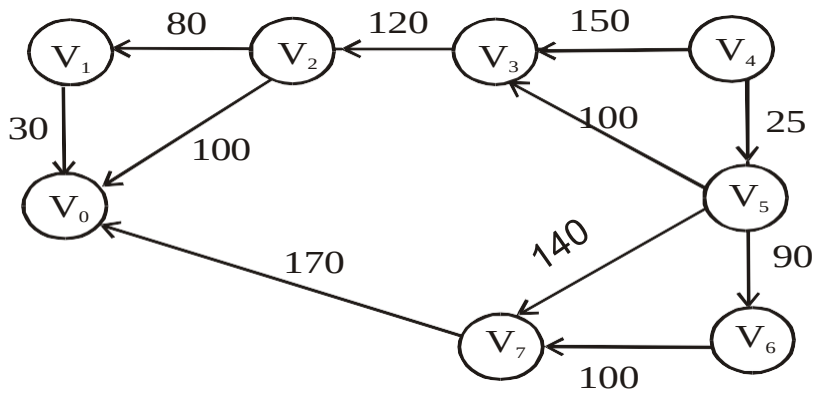
ALGORITHM -

1. V is the starting vertex
2. Initialize visited array to 0.
3. Initialize all elements of distance array as
   dist [i]= cost [v] [i]
4. visited[v] = 1
5. num = 1
6. while (num < n)
   {
   u = choose(dist, n) ; num = num + 1;

   /* choose is a function which returns u such that dist[u] = min{ dist [w]}
   where visited[w] is false */

   for w = 1 to n
   { if (! visited[w])
   if (dist[u] + cost[u][w]<dist[w])
   dist[w] = dist[u]+cost[u][w]
   }
   }
7. dist array contains the shortest paths from V to all other destinations.
8. Stop

EXAMPLE-



Starting vertex = V_4 . All paths are calculated with respect to V_4

Step	Visited	u	Dist Array								Remarks
			0	1	2	3	4	5	6	7	
0	-	-	∞	∞	∞	150	0	25	∞	∞	
1	4	5	∞	∞	∞	125	0	25	115	165	Shorter dist to 3, 6 , 7 via 5
2	4,5	6	∞	∞	∞	125	0	25	115	165	No change via 6
3	4,5,6	3	∞	∞	245	125	0	25	115	165	Shorter dist to 2 via 3
4	4,5,6,3	7	335	∞	245	125	0	25	115	165	Shorter dist to 0 via 7
5	4,5,6,3,7	2	335	325	245	125	0	25	115	165	Shorter dist to 1 via 2
6	4,5,6,3,7,2	1	335	325	245	125	0	25	115	165	No change via 1
7	4,5,6,3,7,2,1	0	335	325	245	125	0	25	115	165	No change via 0
8	4,5,6,3,7,2,1,0	-	335	325	245	125	0	25	115	165	

2. PSEUDO CODE

```
function dijkstra(G, S)
    for each vertex V in G
        distance[V] <- infinite
        previous[V] <- NULL
        If V != S, add V to Priority Queue Q
    distance[S] <- 0

    while Q IS NOT EMPTY
        U <- Extract MIN from Q
        for each unvisited neighbour V of U
            tempDistance <- distance[U] + edge_weight(U, V)
            if tempDistance < distance[V]
                distance[V] <- tempDistance
                previous[V] <- U
    return distance[], previous[]
```

3. PROGRAM CODE SNIPPET

dj.cpp

```cpp
#include <iostream>
#include <vector>

#define INT_MAX 10000000

using namespace std;

void DijkstrasTest();

int main() {
  DijkstrasTest();
  return 0;
}

class Node;
class Edge;

void Dijkstras();
vector<Node*>* AdjacentRemainingNodes(Node* node);
Node* ExtractSmallest(vector<Node*>& nodes);
int Distance(Node* node1, Node* node2);
bool Contains(vector<Node*>& nodes, Node* node);
void PrintShortestRouteTo(Node* destination);

vector<Node*> nodes;
vector<Edge*> edges;

class Node {
   public:
  Node(char id)
    : id(id), previous(NULL), distanceFromStart(INT_MAX) {
    nodes.push_back(this);
  }

   public:
  char id;
  Node* previous;
  int distanceFromStart;
};

class Edge {
   public:
  Edge(Node* node1, Node* node2, int distance)
    : node1(node1), node2(node2), distance(distance) {
    edges.push_back(this);
```

dj.cpp

```
46    }
47    bool Connects(Node* node1, Node* node2) {
48      return (
49        (node1 == this->node1 &&
50         node2 == this->node2) ||
51        (node1 == this->node2 &&
52         node2 == this->node1));
53    }
54
55     public:
56    Node* node1;
57    Node* node2;
58    int distance;
59  };
60
61  ////////////////////
62  void DijkstrasTest() {
63    Node* a = new Node('A');
64    Node* b = new Node('B');
65    Node* c = new Node('C');
66    Node* d = new Node('D');
67    Node* e = new Node('E');
68    Node* f = new Node('F');
69    Node* g = new Node('G');
70
71    Edge* e1 = new Edge(a, c, 1);
72    Edge* e2 = new Edge(a, d, 2);
73    Edge* e3 = new Edge(b, c, 2);
74    Edge* e4 = new Edge(c, d, 1);
75    Edge* e5 = new Edge(b, f, 3);
76    Edge* e6 = new Edge(c, e, 3);
77    Edge* e7 = new Edge(e, f, 2);
78    Edge* e8 = new Edge(d, g, 1);
79    Edge* e9 = new Edge(g, f, 1);
80
81    a->distanceFromStart = 0;  // set start node
82    Dijkstras();
83    PrintShortestRouteTo(f);
84  }
85
86  ////////////////////
87  void Dijkstras() {
88    while (nodes.size() > 0) {
89      Node* smallest = ExtractSmallest(nodes);
90      vector<Node*>* adjacentNodes =
```

dj.cpp

```
91          AdjacentRemainingNodes(smallest);
92      const int size = adjacentNodes->size();
93        for (int i = 0; i < size; ++i) {
94          Node* adjacent = adjacentNodes->at(i);
95          int distance = Distance(smallest, adjacent) +
96                         smallest->distanceFromStart;
97
98          if (distance < adjacent->distanceFromStart) {
99            adjacent->distanceFromStart = distance;
100           adjacent->previous = smallest;
101         }
102       }
103       delete adjacentNodes;
104     }
105   }
106   // Find the node with the smallest distance,
107   // remove it, and return it.
108   Node* ExtractSmallest(vector<Node*>& nodes) {
109     int size = nodes.size();
110     if (size == 0) return NULL;
111     int smallestPosition = 0;
112     Node* smallest = nodes.at(0);
113     for (int i = 1; i < size; ++i) {
114       Node* current = nodes.at(i);
115       if (current->distanceFromStart <
116         smallest->distanceFromStart) {
117         smallest = current;
118         smallestPosition = i;
119       }
120     }
121     nodes.erase(nodes.begin() + smallestPosition);
122     return smallest;
123   }
124   // Return all nodes adjacent to 'node' which are still
125   // in the 'nodes' collection.
126   vector<Node*>* AdjacentRemainingNodes(Node* node) {
127     vector<Node*>* adjacentNodes = new vector<Node*>();
128     const int size = edges.size();
129     for (int i = 0; i < size; ++i) {
130       Edge* edge = edges.at(i);
131       Node* adjacent = NULL;
132       if (edge->node1 == node) {
133         adjacent = edge->node2;
134       } else if (edge->node2 == node) {
135         adjacent = edge->node1;
```

dj.cpp

```
136       }
137       if (adjacent && Contains(nodes, adjacent)) {
138         adjacentNodes->push_back(adjacent);
139       }
140     }
141     return adjacentNodes;
142   }
143   // Return distance between two connected nodes
144   int Distance(Node* node1, Node* node2) {
145     const int size = edges.size();
146     for (int i = 0; i < size; ++i) {
147       Edge* edge = edges.at(i);
148       if (edge->Connects(node1, node2)) {
149         return edge->distance;
150       }
151     }
152     return -1;  // should never happen
153   }
154   // Does the 'nodes' vector contain 'node'
155   bool Contains(vector<Node*>& nodes, Node* node) {
156     const int size = nodes.size();
157     for (int i = 0; i < size; ++i) {
158       if (node == nodes.at(i)) {
159         return true;
160       }
161     }
162     return false;
163   }
164   ////////////////////
165   void PrintShortestRouteTo(Node* destination) {
166     Node* previous = destination;
167     cout << "Distance from start: "
168          << destination->distanceFromStart << endl;
169     while (previous) {
170       cout << previous->id << " ";
171       previous = previous->previous;
172     }
173     cout << endl;
174   }
175   // these two not needed
176   vector<Edge*>* AdjacentEdges(vector<Edge*>& Edges, Node* node);
177   void RemoveEdge(vector<Edge*>& Edges, Edge* edge);
178   vector<Edge*>* AdjacentEdges(vector<Edge*>& edges, Node* node) {
179     vector<Edge*>* adjacentEdges = new vector<Edge*>();
180     const int size = edges.size();
```

```
175   // these two not needed
176   vector<Edge*>* AdjacentEdges(vector<Edge*>& Edges, Node* node);
177   void RemoveEdge(vector<Edge*>& Edges, Edge* edge);
178   vector<Edge*>* AdjacentEdges(vector<Edge*>& edges, Node* node) {
179     vector<Edge*>* adjacentEdges = new vector<Edge*>();
180     const int size = edges.size();
181     for (int i = 0; i < size; ++i) {
182       Edge* edge = edges.at(i);
183       if (edge->node1 == node) {
184         cout << "adjacent: " << edge->node2->id << endl;
185         adjacentEdges->push_back(edge);
186       } else if (edge->node2 == node) {
187         cout << "adjacent: " << edge->node1->id << endl;
188         adjacentEdges->push_back(edge);
189       }
190     }
191     return adjacentEdges;
192   }void RemoveEdge(vector<Edge*>& edges, Edge* edge) {
193     vector<Edge*>::iterator it;
194     for (it = edges.begin(); it < edges.end(); ++it) {
195       if (*it == edge) {
196         edges.erase(it);
197         return;
198       }
199     }
200   }
201
```

4. O/P SNIPPET

```
G:\SEM6\AA LAB\codes\dj.exe
Distance from start: 4
F G D A

--------------------------------
Process exited after 0.02493 seconds with return value 0
Press any key to continue . . .
```

5. ANALYSIS AND ITS PROOF –

The given graph G= (V, E) is represented as an adjacency matrix. Here w [u, v] stores the weight of edge (u, v).

The priority queue Q is represented as an unordered list.

Let |E| and |V| be the number of edges and vertices in the graph, respectively. Then the time complexity is calculated:

- Adding all |V| vertices to Q takes O(|V|) time.
- Removing the node with minimal dist. takes O(|V|) time, and we only need O(1) to recalculate dist [u] and update Q. Since we use an adjacency matrix here, we'll need to loop for |V| vertices to update the dist array.
- The time taken for each iteration of the loop is O(|V|), as one vertex is deleted from Q per loop.
- Thus, total time complexity becomes O(|V|) + O(|V|) \times O(|V|) = O(|V|^2).

The given graph G= (V, E) is represented as an adjacency list.

The priority queue Q is represented as a binary heap or a Fibonacci heap.

Time complexity using a binary heap. In this case, the time complexity is:

- It takes O(|V|) time to construct the initial priority queue of |V| vertices.
- With adjacency list representation, all vertices of the graph can be traversed using BFS. Therefore, iterating over all vertices' neighbors and updating their dist values over the course of a run of the algorithm takes O(|E|) time.
- The time taken for each iteration of the loop is O(|V|), as one vertex is removed from Q per loop.
- The binary heap data structure allows us to extract-min (remove the node with minimal dist) and update an element (recalculate dist[u]) in O(log|V|) time.
- Therefore, the time complexity becomes O(|V|) + O(|E| * log|V|) + O(|V| * log|V|), which is O((|E|+|V|) * log|V|) = O(|E| * log|V|), since |E| >=|V| - 1 as G is a connected graph.

Time complexity using a Fibonacci heap. The Fibonacci heap allows us to insert a new element in O(1) and extract the node with minimal dist in O(log|V). Therefore, the time complexity will be:

- The time taken for each iteration of the loop and extract-min is O(|V|), as one vertex is removed from Q per loop.

- Iterating over all vertices' neighbors and updating their dist values for a run of the algorithm takes O(|E|) time. Since each priority value update takes O(log|V|) time, the total of all dist calculation and priority value updates takes O(|E| * log|V|) time.
- So the overall time complexity becomes O(|V| + |E|* log|V|).

6. TIME COMPLEXITY

CASE 1: NAIVE IMPLEMENTATION

BEST CASE TIME COMPLEXITY

The same situation occurs in best case since again the array is unsorted:

- V calculations
- O(V) time

Total: O(V^2)

AVERAGE CASE TIME COMPLEXITY

The average case doesn't change the steps we have to take since the array isn't sorted, we do not know the costs between each node. Therefore it will remain O(V^2) since

- V calculations
- O(V) time

Total: O(V^2)

WORST CASE TIME COMPLEXITY

As stated above this is the worst case complexity for Dijkstra's algorithm with O(V^2) when implementing using an unsorted array and no priority queue. This is because for each vertex (V), we need to relax the connected edges in order to find the minimum cost edge that connects a vertex to V. We need to do V number of calculations and each operation takes O(V) times, therefore leaving us with the complexity of O(V^2)

- V calculations
- O(V) time

Total: O(V^2)

CASE 2: BINARY HEAP + PRIORITY QUEUE

Worst Case Time Complexity

Our inner loop statements occur O(V + E) times, where V is number of vertices and E is number of edges, with the decrease key operation taking O(logV) meaning the total time complexity for our implementation is O((V + E)*logV) as E -> V this simplifies to O(ElogV). Where we have the largest number of decrease key operations (which take logV).

- Inner loop operations O(V + E) times
- Decrese key takes O(logV)

Total O(ElogV) where decrease key happens the most amount of times

Average Case Time Complexity

The average running time in this case will be O(EVlog(E/V)logV). We know this since our inner loop still takes O(V + E) times, the only difference is the amount of decrease key operations is bounded by O(Vlog(E/V)) meaning that there is a constant on the calculations.

- Inner loop operations O(V + E) times
- Decrease key takes O(logV)

Total O(EVlog(E/V)logV) where decrease key happens O(log(E/V) times

Best Case Time Complexity

Our best case is identical to our worst case however it is when the number of key operations are the smallest. this means we will have a complexity of O(ElogV) still, just will the number of logV operations reduced.

- Inner loop operations O(V + E) times
- Decrease key takes O(logV)

Total O(ElogV) where decrease key happens the least amount of times

CASE 3: FIBONACCI HEAP + PRIORITY QUEUE

Worst Case Time Complexity

The same steps occur in this algorithm as in the binary heap, however, the fibonacci heap can reduce our running time further since to increment a nodes priority now only takes O(1) time, instead of O(logV) like when using

the binary heap. Meaning that our new worst case time would be O(E+VlogV), with the largest number of decrease key calculations.

- Inner loop operations O(V + E) times
- Decrese key takes O(1)

Total O(E+VlogV) where decrease key happens the most amount of times

Average Case Time Complexity

In the average case the fibonacci similar to the binary heap, we limit our number of key decreases by O(Vlog(E/V)). Multiplying this by our original complexity therefore gives us the average of O(E+Vlog(E/V)VlogV).

- Inner loop operations O(V + E) times
- Decrese key takes O(1)

Total O(E+Vlog(E/V)VlogV) where decrease key happens O(Vlog(E/V) amount of times

Best Case Time Complexity

Similarly, to our binary heap implementation, the worst and best case scenarios for fibonacci heap have the same base complexity, the difference being the amount of decrease key operations. This means we will have a complexity of O(E+VlogV), with the smallest amount of O(1) operations for the graph.

- Inner loop operations O(V + E) times
- Decrese key takes O(1)

Total O(E+VlogV) where decrease key happens the least amount of times

7. SPACE COMPLEXITY

Space complexity of Dijkstra's algorithm is O(V2) where V denotes the number of vertices (or nodes) in the graph.

8. UNIQUE CHARACTERISTICS

- Dijkstra's Algorithm finds the shortest path between a given node (which is called the "source node") and all other nodes in a graph.
- This algorithm uses the weights of the edges to find the path that minimizes the total distance (weight) between the source node and all other nodes.

9. APPLICATIONS

- To find the shortest path
- In social networking applications
- In a telephone network
- To find the locations in the map
- Telephone network: In a telephone network the lines have bandwidth, BW. We want to route the phone call via the highest BW.
- Flight: A travel agent requests software for making an agenda of flights for clients. The agent has access to a data base with all airports and flights. Besides the flight number, origin airport and destination, the flights have departure and arrival time. Specifically, the agent wants to determine the earliest arrival time for the destination given an origin airport and start time.
- File Server: We want to designate a file server in a local area network. Now, we consider that most of time transmitting files from one computer to another computer is the connect time. So we want to minimize the number of “hops” from the file server to every other computer on the network.

10. OPTIMIZATION

Many times the range of weights on edges is in a small range (i.e. all edge weights can be mapped to 0, 1, 2. w where w is a small number). In that case, Dijkstra’s algorithm can be modified by using different data structures, and buckets, which is called dial implementation of Dijkstra’s algorithm. time complexity is O(E + WV) where W is the maximum weight onany edge of the graph, so we can see that, if W is small then this implementation runs much faster than the traditional algorithm. The following are important observations.

The maximum distance between any two nodes can be at max w(V – 1) (w is maximum edge weight and we can have at max V-1 edges between two vertices).

In the Dijkstra algorithm, distances are finalized in non-decreasing, i.e., the distance of the closer (to given source) vertices is finalized before the distant vertices.

11. ADVANTAGES

- One of the main advantages of it is its little complexity which is almost linear.
- It can be used to calculate the shortest path between a single node to all other nodes and a single source node to a single destination node by stopping the algorithm once the shortest distance is achieved for the destination node.
- It only works for directed-, weighted graphs and all edges should have non-negative values.

12. DISADVANTAGES

- It do blind search so wastes lot of time while processing.
- It cannot handle negative edges.
- This leads to acyclic graphs and most often cannot obtain the right shortest path.

THANK YOU !

Advanced Algorithm Lab Assignment No: 8

Parallel Prefix Sum

Definition

The prefix-sum algorithm computes all the partial sums of an array of numbers. It is called prefix-sum because it computes sums over all prefixes of the array.

Given an array arr[] of size n, its prefix sum array is another array prefixSum[] of the same size, such that the value of prefixSum[i] is arr[0] + arr[1] + arr[2] … arr[i]. To fill the prefix sum array, we run through index 1 to last and keep on adding the present element with the previous value in the prefix sum array.

Sequential Implementation

It might seem that computing the partial sums is an inherently serial process, because one must add up the first k elements before adding in the element k+1. Indeed, with only a single processor, one might as well do it that way. Algorithm 1 shows the pseudo-code for such an implementation.

1: s ←x[0]

2: **for** i ←1 **to** n−1 **do**

3: s ←s+x[i]

4: x[i] ← s

Algorithm 1: Sequential algorithm that computes the prefix-sum of an array x containing n elements by doing n-1 additions.

Since we are scanning through the array once the time complexity of getting the prefix sum is O(n).

Parallel Implementation

There are two key algorithms for computing a prefix sum in parallel. The first offers a shorter span and more parallelism but is not work-efficient. The second is work-efficient but requires double the span and offers less parallelism.

A Naive Parallel Scan

This algorithm is based on the scan algorithm presented by Hillis and Steele (1986) and demonstrated for GPUs by Horn (2005). The algorithm performs $O(n \log_2 n)$ addition operations. Remember that a sequential scan performs $O(n)$ adds. Therefore, this naive implementation is not work-efficient. The factor of $\log_2 n$ can have a large effect on performance.

1: for d = 1 to $\log_2 n$ **do**
2: **for all** k in parallel **do**
3: **if** $k >= 2^{d}$ **then**
4: $x[k] = x[k - 2^{d-1}] + x[k]$

This algorithm assumes that there are as many processors as data elements. For large this is not usually the case. Instead, the programmer must divide the computation among several *thread blocks* that each scans a portion of the array on a single multiprocessor. Even still, the number of processors in a multiprocessor is typically much smaller than the number of threads per block, so the hardware automatically partitions the "for all" statement into small parallel batches (called *warps*) that are executed sequentially on the multiprocessor.

To solve this problem, we need to double-buffer the array we are scanning using two temporary arrays. Note that this code will run on

only a single thread block, and so the size of the arrays it can process is limited.

Example 2. A Double-Buffered Version of the Sum Scan from Algorithm 1

1: **for** d = 1 to $\log_2 n$ **do**
2: **for all** k in parallel **do**
3: **if** k $\geq 2^{d}$ **then**
4: x[out][k] = x[in][$k - 2^{d-1}$] + x[in][k]
5: **else**
6: x[out][k] = x[in][k]

Parallel prefix sum - Analysis

Time:

- all additions of one round run in parallel
- [log n] rounds
- O(log n) time best possible!

Work:

- >= n/2 additions in all rounds (except maybe last round)
- O(n log n) work
- more than the O(n) sequential algorithm!

A Work-Efficient Parallel Scan

standard trick to improve work: compute small blocks sequentially

1. Set b := [log n]

2. For blocks of b consecutive indices, i.e., A[0..b), A[b..2b), . do in parallel: compute local prefix sums sequentially

3. Use previous work-inefficient algorithm only on rightmost elements of block, i.e., to compute prefix sums of A[b - 1], A[2b - 1], A[3b - 1], ….

4. For blocks A[0..b), A[b..2b), . do in parallel:

Add block-prefix sums to local prefix sums

Analysis:

Time:

- 2. & 4.: O(b) = O(logn)time
- 3. O(log(n/b)) = O(log n) times

Work:

- 2. & 4.: O(b) per block x [n/b] blocks = O(n)
- 3. O(n/b log(n/b)) = O(n)

The naïve parallel implementation of scan would probably perform very badly on large arrays due to its work-inefficiency. The work-efficient scan algorithm avoids the extra factor of $\log_2 n$ work performed by the naive algorithm.

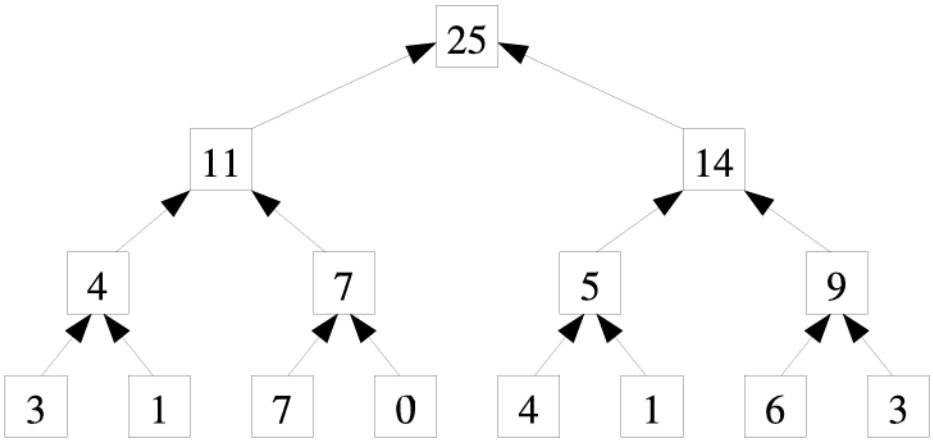
To do this we will use an algorithmic pattern that arises often in parallel computing: *balanced trees*. The idea is to build a balanced binary tree on the input data and sweep it to and from the root to compute the prefix sum. A binary tree with n leaves has $d = \log_2 n$ levels, and each level d has 2^d nodes. If we perform one add per node, then we will perform $O(n)$ adds on a single traversal of the tree.

The tree we build is not an actual data structure, but a concept we use to determine what each thread does at each step of the traversal. In this work-efficient scan algorithm, we perform the operations in place on an array in shared memory.

The algorithm consists of two phases:

1. the *reduce phase* (also known as the *up-sweep phase*)
2. the *down-sweep phase*.

In the reduce phase, we traverse the tree from leaves to root computing partial sums at internal nodes of the tree. This is also known as a parallel reduction, because after this phase, the root node (the last node in the array) holds the sum of all nodes in the array. Pseudocode for the reduce phase is given in Algorithm 3.



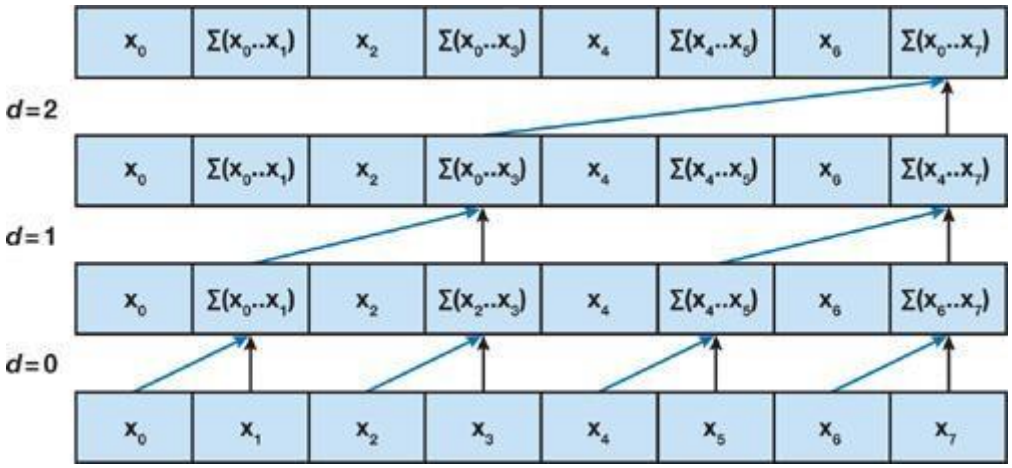
Reduce Step Illustration

On an EREW PRAM, each level of the tree can be executed in parallel, so the implementation can step from the leaves to the root of the tree (see Figure 1.2b); we call this an up-sweep. Since the tree is of depth ⌈logn⌉, and one processor is needed for every pair of elements, the algorithm requires O(log n) time and n/2 processors. If we assume a fixed number of processors p, with n > p, then each processor can sum an n/p section of the vector to generate a processor sum; the tree technique can then be used to reduce the processor sums (see Figure 1.3). The time taken to generate the processor sums is ⌈n/p⌉, so the total time required on an EREW PRAM is:

$$TR(n,p) = \lceil n/p \rceil + \lceil \lg p \rceil = O(n/p + \log p).$$

When $n/p \geq \lg p$ the complexity is $O(n/p)$. This time is an optimal speedup over the sequential algorithm given in Figure 1.1. We now return to the scan operation. We show how to implement the pre-scan operation; the scan is then determined by shifting the result and putting the sum at the end. If we look at the tree generated by the reduce operation, it contains many partial sums over regions of the vector. It turns out that these partial sums can be used to generate all

the prefix sums. This requires executing another sweep of the tree with one step per level, but this time starting at the root and going to the leaves (a down-sweep). Initially, the identity element is inserted at the root of the tree. On each step, each vertex at the current level passes to its left child its own value, and it passes to its right child, summation applied to the value from the left child from the up-sweep and its own value.

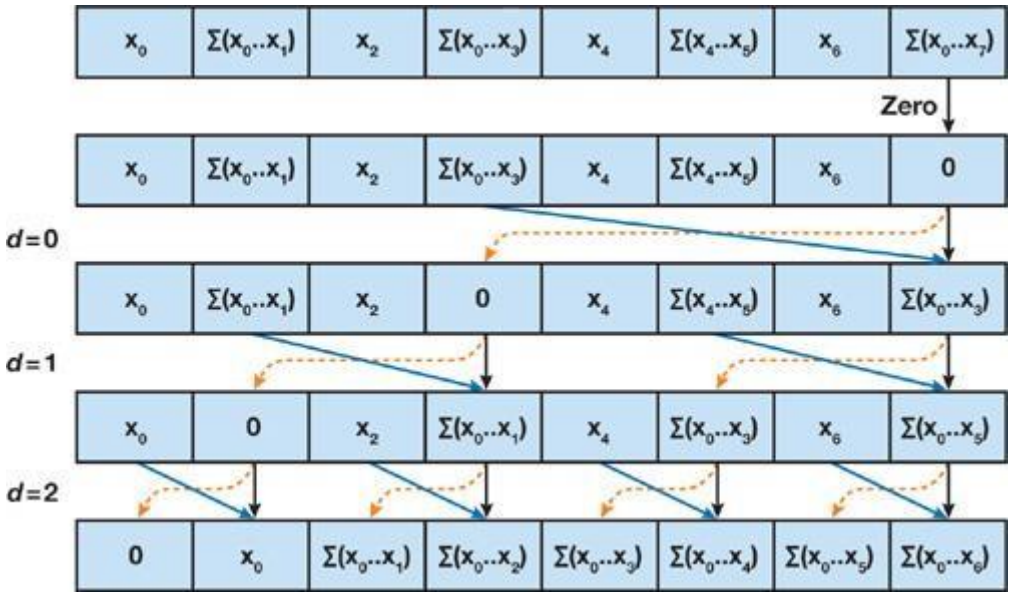


An Illustration of the Up-Sweep, or Reduce, Phase of a Work-Efficient Sum Scan Algorithm

Example 3. The Up-Sweep (Reduce) Phase of a Work-Efficient Sum Scan Algorithm (After Blelloch 1990)

1: **for** $d = 0$ to $\log_2 n - 1$ **do**
2: **for all** $k = 0$ to $n - 1$ by 2^{d+1} in parallel **do**
3: $x[k + 2^{d+1} - 1] = x[k + 2^{d} - 1] + x[k + 2^{d} + 1 - 1]$

In the down-sweep phase, we traverse back down the tree from the root, using the partial sums from the reduce phase to build the scan in place on the array. We start by inserting zero at the root of the tree, and on each step, each node at the current level passes its own value to its left child, and the sum of its value and the former value of its left child to its right child. Like the naive scan code, the code will run on only a single thread block.



An Illustration of the Down-Sweep Phase of the Work-Efficient Parallel Sum Scan Algorithm

The Down-Sweep Phase of a Work-Efficient Parallel Sum Scan Algorithm

1: $x[n-1] \leftarrow 0$
2: **for** $d = \log_2 n - 1$ down to 0 **do**
3: **for all** $k = 0$ to $n - 1$ by $2^{d}+1$ in parallel **do**
4: $t = x[k + 2^{d} - 1]$
5: $x[k + 2^{d} - 1] = x[k + 2^{d}+1 - 1]$
6: $x[k + 2^{d}+1 - 1] = t + x[k + 2^{d}+1 - 1]$

The scan algorithm in Algorithm 4 performs $O(n)$ operations (it performs 2 x $(n - 1)$ adds and $n - 1$ swaps); therefore, it is work-efficient and, for large arrays, should perform much better than the naive algorithm from the previous section. Algorithmic efficiency is not enough; we must also use the hardware efficiently.

Applications

1. **Counting sort**: Counting sort is an integer sorting algorithm that uses the prefix sum of a histogram of key frequencies to calculate the position of each key in the sorted output array. It runs in linear time for integer keys that are smaller than the number of items,

and is frequently used as part of radix sort, a fast algorithm for sorting integers that are less restricted in magnitude.

2. **List ranking**: List ranking, the problem of transforming a linked list into an array that represents the same sequence of items, can be viewed as computing a prefix sum on the sequence 1, 1, 1, ... and then mapping each item to the array position given by its prefix sum value; by combining list ranking, prefix sums, and Euler tours, many important problems on trees may be solved by efficient parallel algorithms.
3. **Binary Adders**: An early application of parallel prefix sum algorithms was in the design of binary adders, Boolean circuits that can add two n-bit binary numbers. In this application, the sequence of carry bits of the addition can be represented as a scan operation on the sequence of pairs of input bits, using the majority function to combine the previous carry with these two bits. Each bit of the output number can then be found as the exclusive or of two input bits with the corresponding carry bit.
4. **Gray codes**: In the construction of gray codes, sequences of binary values with the property that consecutive sequence values differ from each other in a single bit position, a number n can be converted into the gray code value at position n of the sequence simply by taking the exclusive or of n and n/2 (the number formed by shifting n right by a single bit position). The reverse operation, decoding a Gray-coded value x into a binary number, is more complicated, but can be expressed as the prefix sum of the bits of x, where each summation operation within the prefix sum is performed modulo two.