Algorithms Design and Analysis Lab

ETMA 351

Faculty: **Dr. Deepak Gupta**

Name: **Syeda Reeha Quasar**

Roll No.: **14114802719**

Semester: **5**

Maharaja Agrasen Institute of Technology, PSP Area,

Sector – 22, Rohini, New Delhi – 110085

MAHARAJA AGRASEN INSTITUTE OF TECHNOLOGY

COMPUTER SCIENCE & ENGINEERING DEPARTMENT

VISION

To Produce "Critical thinkers of Innovative Technology"

MISSION

To provide an excellent learning environment across the computer science discipline to inculcate professional behavior, strong ethical values, innovative research capabilities and leadership abilities which enable them to become successful entrepreneurs in this globalized world.

1. To nurture an **excellent learning environment** that helps students to enhance their problem solving skills and to prepare students to be lifelong learners by offering a solid theoretical foundation with applied computing experiences and educating them about their **professional, and ethical responsibilities**.
2. To establish **Industry-Institute Interaction**, making students ready for the industrial environment and be successful in their professional lives.
3. To promote **research activities** in the emerging areas of technology convergence.
4. To build engineers who can look into technical aspects of an engineering solution thereby setting a ground for producing successful **entrepreneur.**

VISION

To nurture young minds in a learning environment of high academic value and im-bibe spiritual and ethical values with technological and management competence.

MISSION

The Institute shall endeavor to incorporate the following basic missions in the teaching methodology:

Engineering Hardware - Software Symbiosis

Practical exercises in all Engineering and Management disciplines shall be carriedout by Hardware equipment as well as the related software enabling deeper un- derstanding of basic concepts and encouraging inquisitive nature.

Life - Long Learning

The Institute strives to match technological advancements and encourage studentsto keep updating their knowledge for enhancing their skills and inculcating theirhabit of continuous learning.

Liberalization and Globalization

The Institute endeavors to enhance technical and management skills of students so that they are intellectually capable and competent professionals with Industrial Aptitude to face the challenges of globalization.

Diversification

The Engineering, Technology and Management disciplines have diverse fields of studies with different attributes. The aim is to create a synergy of the above at- tributes by encouraging analytical thinking.

Digitization of Learning Processes

The Institute provides seamless opportunities for innovative learning in all En- gineering and Management disciplines through digitization of learning processes using analysis, synthesis, simulation, graphics, tutorials and related tools to createa platform for multi-disciplinary approach.

INDEX

Sr. no.	Program Name	Date	R1	R2	R3	R4	R5	Total Marks	Signature
1.	To implement following sorting techniques and analyze their time complexity. a) Bubble sort b) Bucket sort c) Radix sort d) Shell sort e) Selection sort f) Heapsort g) Insertion Sort	21-09-2021 21-09-2021 28-09-2021 28-09-2021 28-09-2021 05-10-2021 05-10-2021							
2.	To implement Linear search and Binary search and analyse its time complexity.	19-10-2021							
3.	To implement divide and conquer techniques and analyse its time Complexity. a) Merge sort b) Quick sort c) Matrix Multiplication and Strassen's Algorithm	26-10-2021							
4.	To implement dynamic programming techniques and analyse its time complexity. a) LCS (Longest Common Subsequence) b) Matrix Chain Multiplication c) Optimal Binary Search d) Binomial Coefficient	09-11-2021							
5.	To implement Algorithms using Greedy Approach and analyse its time complexity. a) Knapsack problem b) Activity Selection c) Huffman Encoding d) Task Scheduling Problem	16-11-2021							
6.	To implement Dijkstra's Algorithm and analyse its time complexity.	30-11-2021							

INDEX

7.	To implement minimum spanning trees algorithm and analyse its time complexity. a) Krushkal's Algorithm b) Prim's Algorithm	07-12-2021							
8.	To implement String matching algorithm and analyse its time complexity. a) Naïve Algorithm b) Rabin Karp Algorithm c) Knuth Morris Pratt Algorithm	14-12-2021							

EXPERIMENT - 1

Algorithms Design and Analysis Lab

Aim

To implement following sorting techniques and analyze their time complexity.

Syeda Reeha Quasar
14114802719
4C7

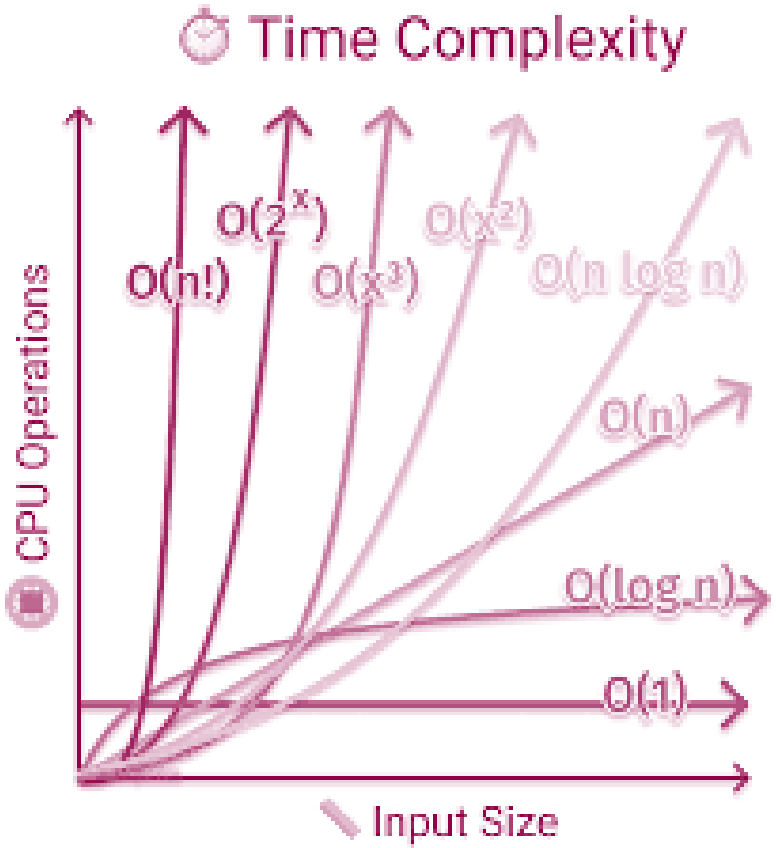
EXPERIMENT – 1

Aim:

To implement following sorting techniques and analyze their time complexity.

Theory:

Time complexity is the amount of time taken by an algorithm to run, as a function of the length of the input. It measures the time taken to execute each statement of code in an algorithm.



Data Structure with their Time Complexity:

Data Structures

Data Structure	Time Complexity								Space Complexity
	Average				Worst				Worst
	Indexing	Search	Insertion	Deletion	Indexing	Search	Insertion	Deletion	
Basic Array	O(1)	O(n)	-	-	O(1)	O(n)	-	-	O(n)
Dynamic Array	O(1)	O(n)	O(n)	O(n)	O(1)	O(n)	O(n)	O(n)	O(n)
Singly-Linked List	O(n)	O(n)	O(1)	O(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Doubly-Linked List	O(n)	O(n)	O(1)	O(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Skip List	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)	O(n)	O(n)	O(n)	O(n log(n))
Hash Table	-	O(1)	O(1)	O(1)	-	O(n)	O(n)	O(n)	O(n)
Binary Search Tree	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)
Cartresian Tree	-	O(log(n))	O(log(n))	O(log(n))	-	O(n)	O(n)	O(n)	O(n)
B-Tree	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Red-Black Tree	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Splay Tree	-	O(log(n))	O(log(n))	O(log(n))	-	O(log(n))	O(log(n))	O(log(n))	O(n)
AVL Tree	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)

Clock/time analysis implementation

Code

```
#include <iostream>
#include <ctime>
using namespace std;

int main() {

  // use time() with NULL argument
  //   cout << time(NULL);

  time_t current_time;

  // stores time in current_time
  time(&current_time);
```

```
  cout << current_time;
  cout << " seconds has passed since 00:00:00 GMT, Jan 1, 1970";
  return 0;
}
```

Output

```
1635196484 seconds has passed since 00:00:00 GMT, Jan 1, 1970

...Program finished with exit code 0
Press ENTER to exit console.
```

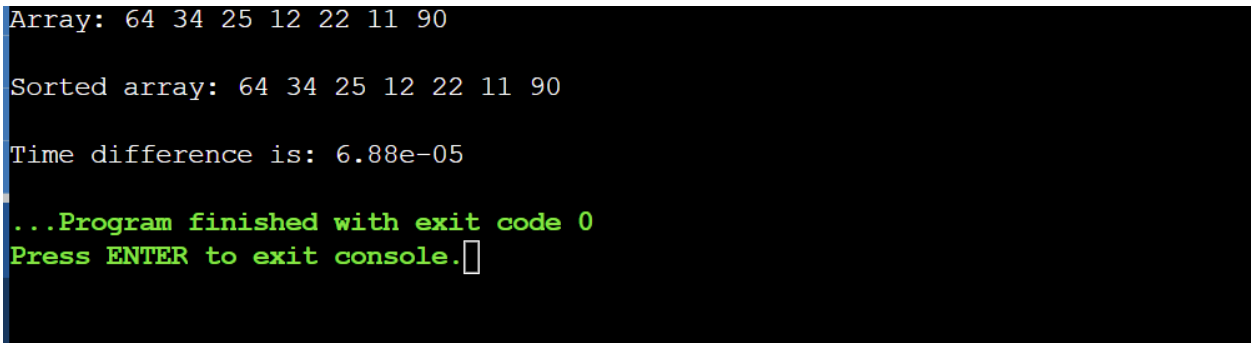
Sorting implementation

Code

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

void printArray(int arr[], int size)    {
    int i;
    for (i = 0; i < size; i++)
        cout << arr[i] << " ";
    cout << endl;
}

int main()  {
```

```
    int arr[] = {64, 34, 25, 12, 22, 11, 90};
    int n = sizeof(arr)/sizeof(arr[0]);
    auto start = chrono::steady_clock::now();
    // unsync the I/O of C and C++.
    ios_base::sync_with_stdio(false);

    cout << "Array: ";
    printArray(arr, n);
    cout << endl;
    cout << "Sorted array: ";
    printArray(arr, n);
    cout << endl;
    auto end = chrono::steady_clock::now();

    double time_taken = chrono::duration_cast<chrono::nanoseconds>(end -
start).count();
    time_taken *= 1e-9;

    cout << "Time difference is: " << time_taken << setprecision(6);
    return 0;
}
```

Output



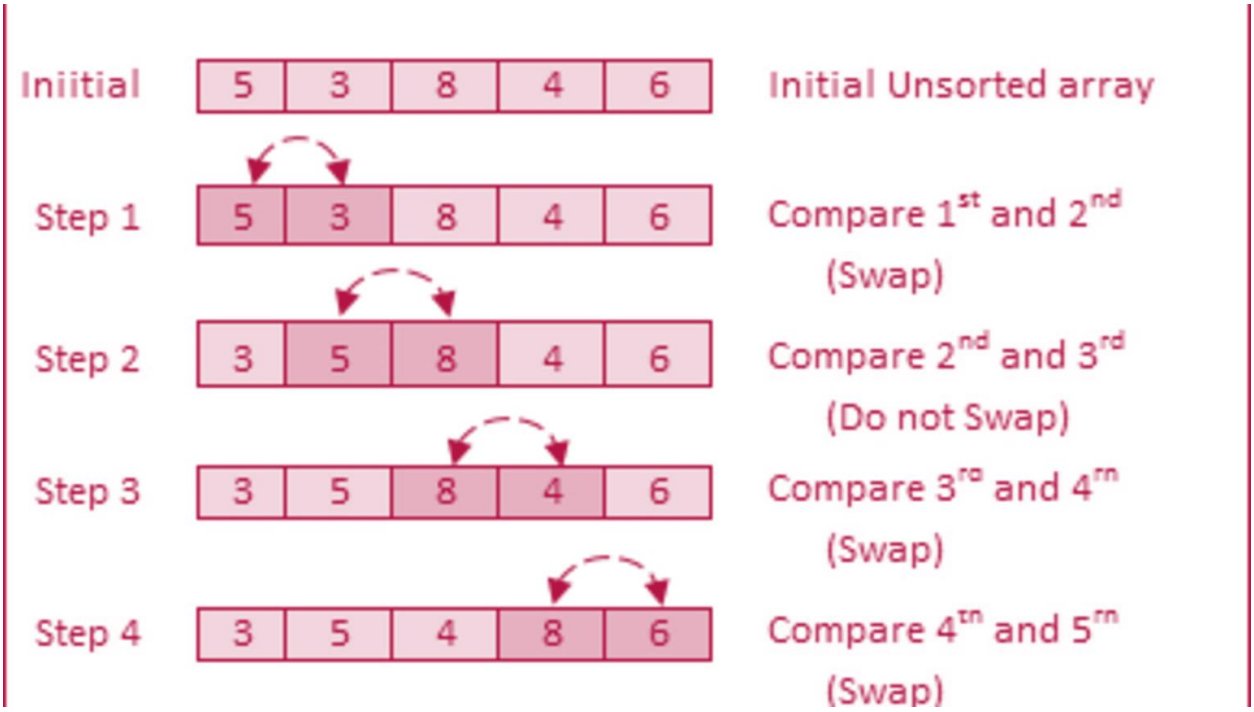
1.1 Bubble Sort:

This sorting algorithm is comparison-based algorithm in which each pair of adjacent elements is compared, and the elements are swapped if they are not in order.

Pseudo code

```
We assume list is an array of n elements. We further assume that swap
function swaps the values of the given array elements.
begin BubbleSort (list)
   for all elements of list
      if list[i] > list[i+1]
         swap (list[i], list [i+1])
      end if
   end for
      return list
   end BubbleSort
```

Example:



Result and Analysis

Worst and Average Case Time Complexity: O (n*n). Worst case occurs when array is reverse sorted and its time complexity is O (n*n) and Best case occurs when array is already sorted and its time complexity is O (n) (linear time). This sort takes just O (1) extra space.

Source Code:

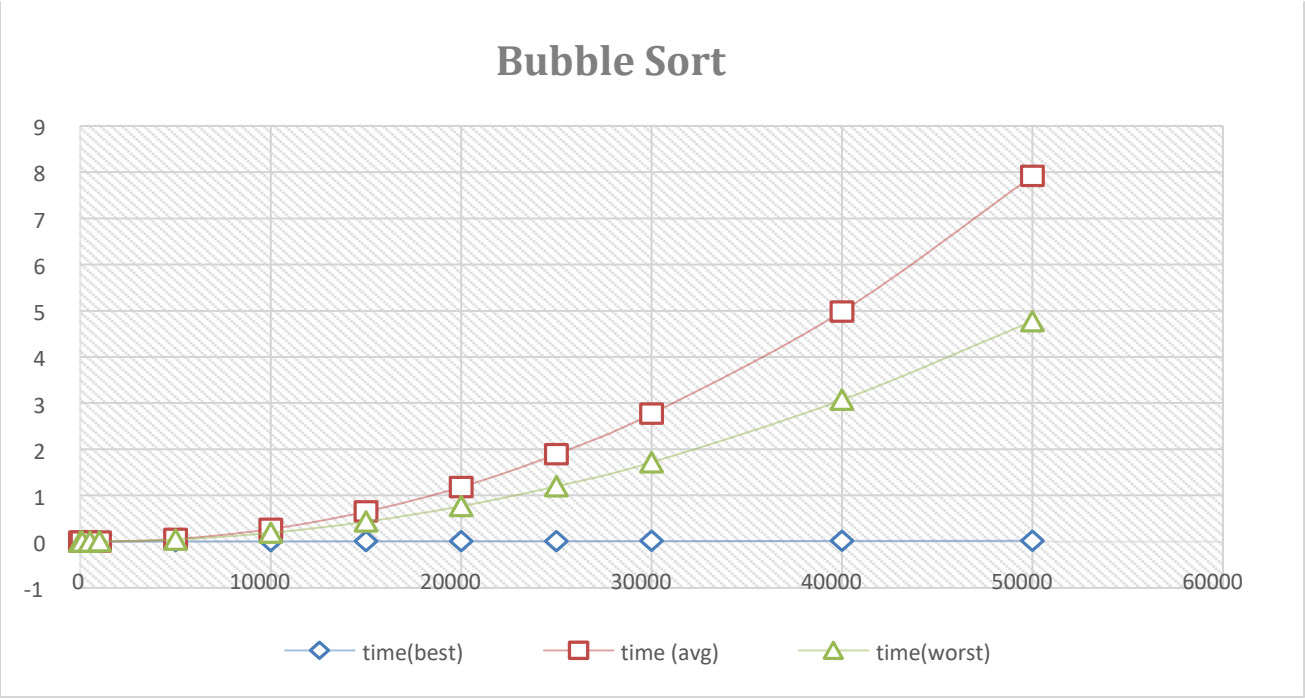
```
#include <bits/stdc++.h>
#include <chrono>
using namespace std;

void swap(int *xp, int *yp) {
    int temp = *xp;
    *xp = *yp;
    *yp = temp;
}

void bubbleSort(int arr[], int n)   {
   int i, j;
   bool swapped;
   for (i = 0; i < n - 1; i++)  {
     swapped = false;
     for (j = 0; j < n - i - 1; j++)    {
        if (arr[j] > arr[j+1])  {
           swap(&arr[j], &arr[j + 1]);
           swapped = true;
        }
     }

     if (swapped == false)  {// no swap means array sorted so break out
        break;
     }
   }
}

void printArray(int arr[], int size)    {
    int i;
    for (i = 0; i < size; i++)
```

```
        cout << arr[i] << " ";
    cout << endl;
}

int main()  {
    // int arr[] = {64, 34, 25, 12, 22, 11, 90};
    // int n = sizeof(arr)/sizeof(arr[0]);

    int n = rand() % 100;
    int arr[n];

    for (int i = 0; i < n; i++) {
        arr[i] = rand() % 100;
    }

    cout << "Array: ";
    printArray(arr, n);
    cout << endl;

    auto start = chrono::high_resolution_clock::now();

    // unsync the I/O of C and C++.
    ios_base::sync_with_stdio(false);

    bubbleSort(arr, n);

    auto end = chrono::high_resolution_clock::now();

    cout << "Sorted array: ";
    printArray(arr, n);
    cout << endl;

    double time_taken = chrono::duration_cast<chrono::nanoseconds>(end -
start).count();
    time_taken *= 1e-9;

    cout << "Time difference is: " << time_taken << setprecision(6);

    return 0;
}
```

Output:

input

```
Array: 86 77 15 93 35 86 92 49 21 62 27 90 59 63 26 40 26 72 36 11 68 67 29
82 30 62 23 67 35 29 2 22 58 69 67 93 56 11 42 29 73 21 19 84 37 98 24 15 70
 13 26 91 80 56 73 62 70 96 81 5 25 84 27 36 5 46 29 13 57 24 95 82 45 14 67
 34 64 43 50 87 8 76 78

Sorted array: 2 5 5 8 11 11 13 13 14 15 15 19 21 21 22 23 24 24 25 26 26 26
27 27 29 29 29 29 30 34 35 35 36 36 37 40 42 43 45 46 49 50 56 56 57 58 59 6
2 62 62 63 64 67 67 67 67 68 69 70 70 72 73 73 76 77 78 80 81 82 82 84 84 86
 86 87 90 91 92 93 93 95 96 98

Time difference is: 5.3534e-05

...Program finished with exit code 0
Press ENTER to exit console.
```



Batch Analysis:

Source Code:

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

void swap(int *xp, int *yp)
{
    int temp = *xp;
    *xp = *yp;
    *yp = temp;
}

void bubbleSort(int arr[], int n)
{
    int i, j;
    bool swapped;
    for (i = 0; i < n - 1; i++)
    {
        swapped = false;
        for (j = 0; j < n - i - 1; j++)
        {
            if (arr[j] > arr[j + 1])
            {
                swap(&arr[j], &arr[j + 1]);
                swapped = true;
            }
        }
```

```
        if (swapped == false)
        { // no swap means array sorted so break out
            break;
        }
    }
}

void printArray(double arr[], int size)
{
    for (int i = 0; i < size; i++)
        cout << arr[i] << ", ";
    cout << endl;
}

void printArray(int arr[], int size)
{
    for (int i = 0; i < size; i++)
        cout << arr[i] << ", ";
    cout << endl;
}

void printArray(float arr[], int size)
{
    for (int i = 0; i < size; i++)
        cout << arr[i] << ", ";
    cout << endl;
}

double sortApp(int n)
{
```

```
    int arr[n];
    for (int i = 0; i < n; i++)
    {
        arr[i] = rand() % 100;
    }

    auto start = chrono::high_resolution_clock::now();
    ios_base::sync_with_stdio(false);

    bubbleSort(arr, n);

    auto end = chrono::high_resolution_clock::now();

    double time_taken = chrono::duration_cast<chrono::nanoseconds>(end -
start).count();
    time_taken *= 1e-9;

    return time_taken;
}

int main()
{
    double times[10];
    int ns[10];
    for (int x = 0; x < 10; x++)
    {
        int n = rand() % 100;
        ns[x] = n;
        times[x] = sortApp(n);
    }
    cout << "value of n's: " << endl;
```

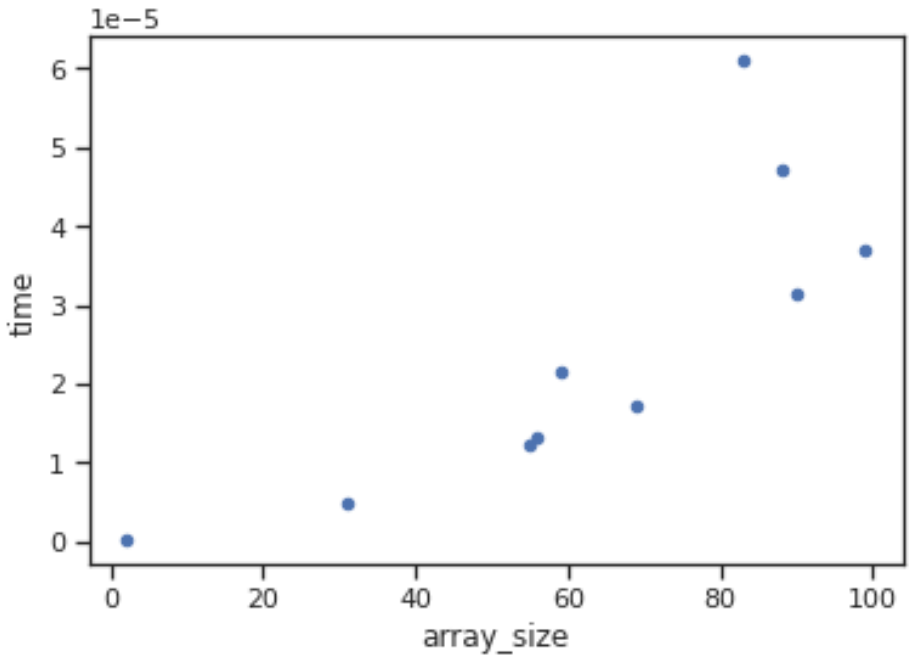
```
    printArray(ns, 10);
    cout << "time for each n: " << endl;
    printArray(times, 10);
}
```

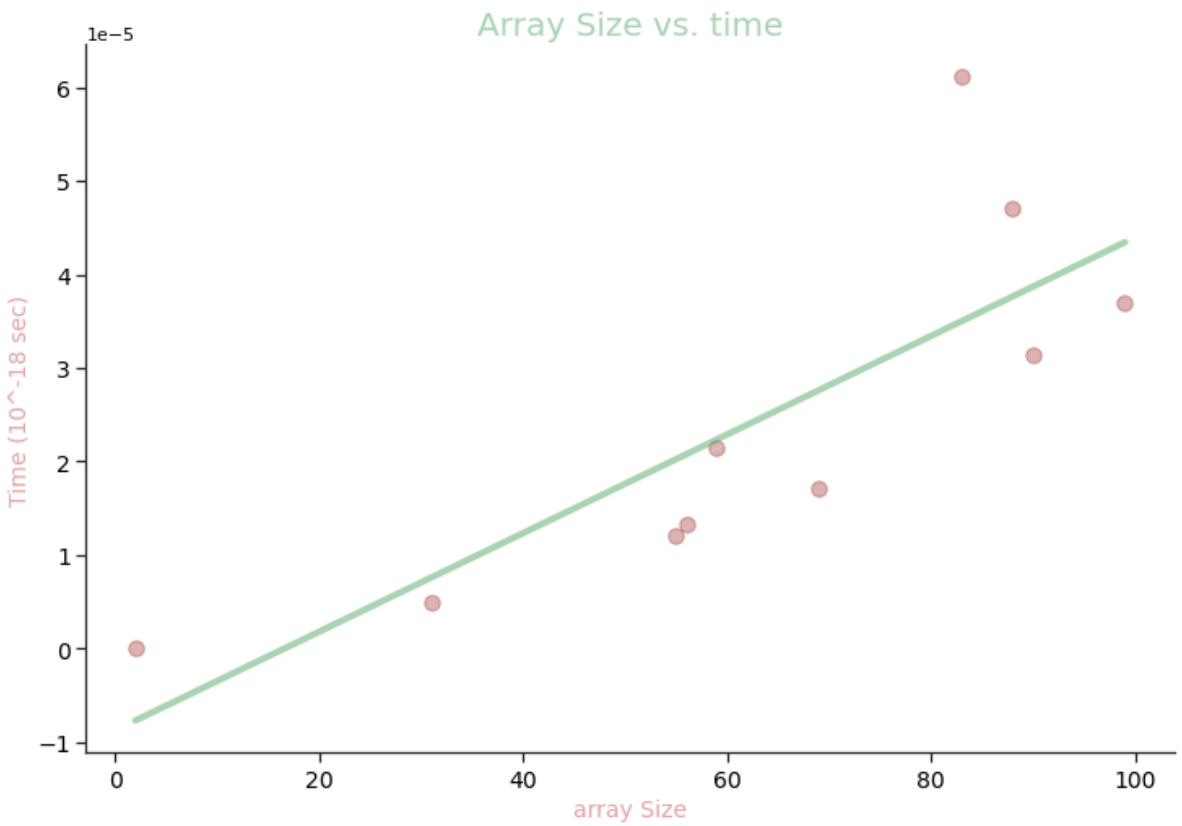
Output:

```
input
value of n's:
83, 88, 56, 69, 90, 59, 2, 99, 55, 31,
time for each n:
8.4473e-05, 3.8244e-05, 1.6018e-05, 2.1333e-05, 6.078e-05, 2.466e-05, 1.39e-
07, 4.5619e-05, 1.4549e-05, 6.06e-06,


...Program finished with exit code 0
Press ENTER to exit console.
```

arraySize = [83, 88, 56, 69, 90, 59, 2, 99, 55, 31]

time = [6.1129e-05, 4.7125e-05, 1.3298e-05, 1.7092e-05, 3.147e-05, 2.1538e-05, 8.6e-08, 3.6987e-05, 1.2126e-05, 4.909e-06]




Array Size vs. time
1e−5
6
5
4
3
2
1
0
−1
Time (10^-18 sec)
0
20
40
60
80
100
array Size

Bubble Sort

Viva Questions

1. How can the best-case efficiency of bubble sort be improved?

Ans.

Some iterations can be skipped if the list is sorted, hence efficiency improves to O(n).

A better version of bubble sort, known as modified bubble sort, includes a flag that is set if an exchange is made after an entire pass over the array. If no exchange is made, then it should be clear that the array is already in order because no two elements need to be switched.

2. Is bubble sorting a stable sort?

Ans.

Yes

sorting algorithm is said to be stable if two objects with equal keys appear in the same order in sorted output as they appear in the input array to be sorted.

3. How much time will bubble sort take if all the elements are same?

Ans.

O(n)

Bubble sort has a worst-case and average complexity of O(n2), where n is the number of items being sorted. Most practical sorting algorithms have substantially better worst-case or average complexity, often O(n log n).

4. What would happen if bubble sort didn't keep track of the number of swaps made on each pass through the list?

Ans.

The algorithm wouldn't know when to terminate as it would have no way of knowing when the list was in sorted order.

5. List main properties to be considered in bubble sort?

Ans.

- Large values are always sorted first.
- It only takes one iteration to detect that a collection is already sorted.
- The best time complexity for Bubble Sort is O(n). ...
- The space complexity for Bubble Sort is O(1), because only single additional memory space is required.

1.2 Bucket Sort:

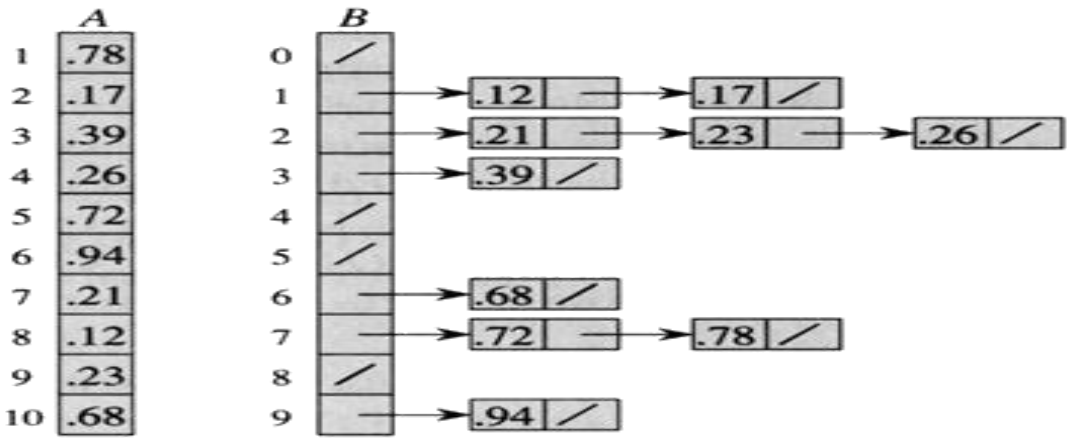
Bucket sort, or **bin sort**, is a sorting algorithm that works by distributing the elements of an array into a number of buckets. Each bucket is then sorted individually, either using a different sorting algorithm, or by recursively applying the bucket sorting algorithm.

It is a distribution sort, a generalization of pigeonhole sort, and is a cousin of radix sort in the most-to-least significant digit flavour. Bucket sort can be implemented with comparisons and therefore can also be considered a comparison sort algorithm.

It works as follows:

1. Set up an array of initially empty "buckets".
2. **Scatter**: Go over the original array, putting each object in its bucket.
3. Sort each non-empty bucket.
4. **Gather**: Visit the buckets in order and put all elements back into the original array.

. This will be the sorted list.



Pseudo code

The code assumes that input is an n-element array A and each element in A satisfies 0 ≤ A[i] ≤ 1. We also need an auxiliary array B [0 . . n -1] for linked-lists (buckets).

```
BUCKET SORT (A)
 1. n ← length [A]
2. For i =1 to n do
3. Insert A[i]into list B [A[i]/b] where b is the bucket size
4. For i =0 to n-1do
5. Sort list B with Insertion sort
6. Concatenate the lists B [0], B [1] ... B [n-1] together in order.
```

Example:

Result and Analysis

In the above Bucket sort algorithm, we observe in the *best case*, the algorithm distributes the elements uniformly between buckets, a few elements are placed on each bucket and sorting the buckets is **O (1)**. Rearranging the elements is one more run through the initial list.

T (n) = [time to insert *n* elements in array *A*] + [time to go through auxiliary array *B* [0 . . . *n*-1] * (Sort by INSERTION_SORT)
= O (*n*) + (*n*-1) (*n*)
= O (*n*)

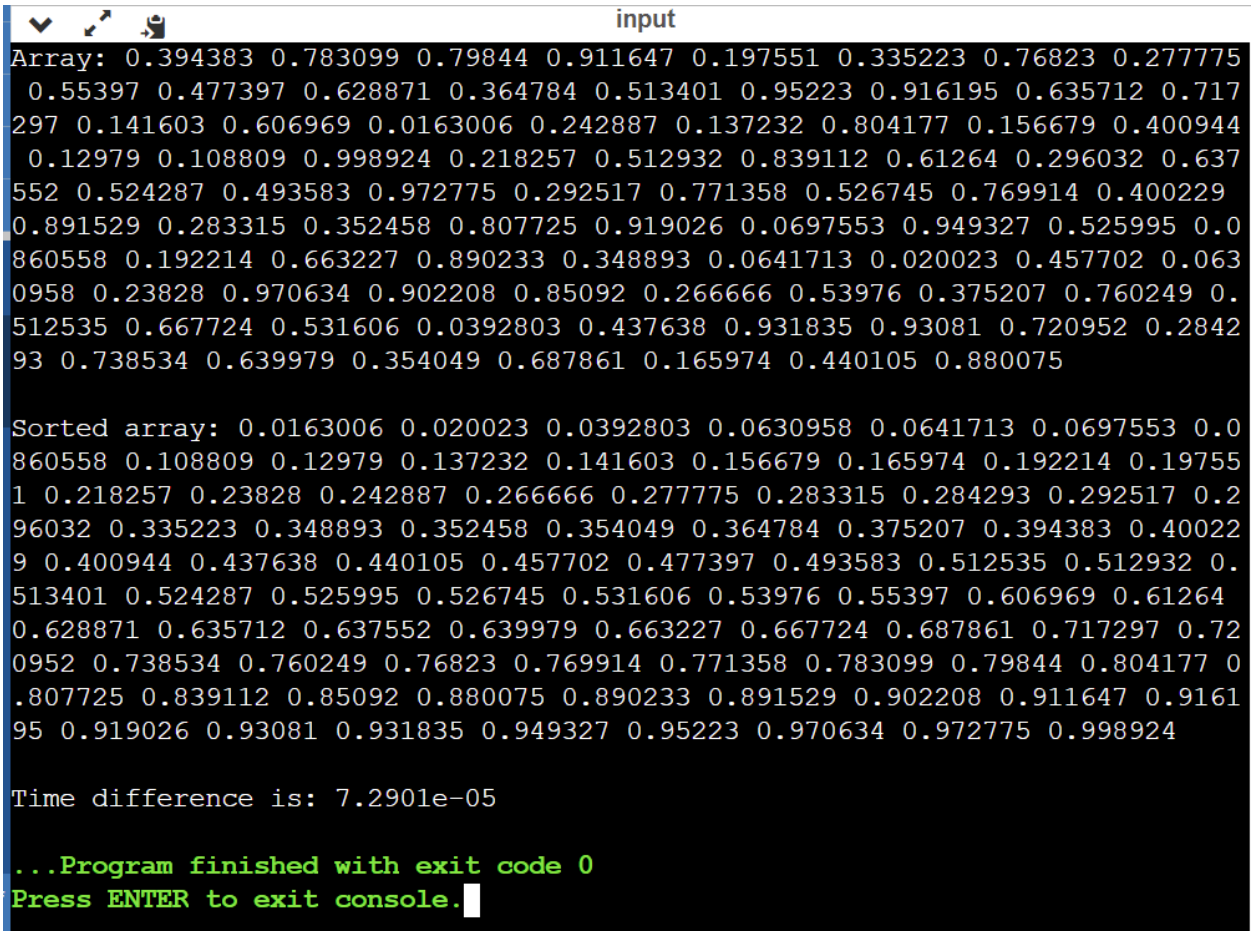
In the *worst case*, the elements are sent all to the same bucket, making the process take **O (n^2)**.

Source Code:

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

void bucketSort(float arr[], int n) {
    vector<float> bucket[n];

 // put elemets in respective buckets
    for (int i = 0; i < n; i++) {
        int bi = n * arr[i]; // Index in bucket
        bucket[bi].push_back(arr[i]);
    }

    for (int i = 0; i < n; i++) // sorting each buckets
        sort(bucket[i].begin(), bucket[i].end());

    int index = 0;
    for (int i = 0; i < n; i++) // Concatenate all buckets into arr[]
        for (int j = 0; j < bucket[i].size(); j++)
            arr[index++] = bucket[i][j];
}
```

```
void printArray(float arr[], int size)    {
    for (int i = 0; i < size; i++)
        cout << arr[i] << " ";
    cout << endl;
}

int main()  {
    // int arr[] = {64, 34, 25, 12, 22, 11, 90};
    // int n = sizeof(arr)/sizeof(arr[0]);
    int n = rand() % 100;
    float arr[n];
    for (int i = 0; i < n; i++) {
        arr[i] = (float(rand())/float((RAND_MAX)));
    }
    cout << "Array: ";
    printArray(arr, n);
    cout << endl;

    auto start = chrono::high_resolution_clock::now();
    // unsync the I/O of C and C++.
    ios_base::sync_with_stdio(false);

    bucketSort(arr, n);  // sort arr

    auto end = chrono::high_resolution_clock::now();

    cout << "Sorted array: ";
    printArray(arr, n);
    cout << endl;

    double time_taken = chrono::duration_cast<chrono::nanoseconds>(end -
start).count();
    time_taken *= 1e-9;

    cout << "Time difference is: " << time_taken << setprecision(6);
    return 0;
}
```

Output:


input
Array: 0.394383 0.783099 0.79844 0.911647 0.197551 0.335223 0.76823 0.277775
0.55397 0.477397 0.628871 0.364784 0.513401 0.95223 0.916195 0.635712 0.717
297 0.141603 0.606969 0.0163006 0.242887 0.137232 0.804177 0.156679 0.400944
0.12979 0.108809 0.998924 0.218257 0.512932 0.839112 0.61264 0.296032 0.637
552 0.524287 0.493583 0.972775 0.292517 0.771358 0.526745 0.769914 0.400229
0.891529 0.283315 0.352458 0.807725 0.919026 0.0697553 0.949327 0.525995 0.0
860558 0.192214 0.663227 0.890233 0.348893 0.0641713 0.020023 0.457702 0.063
0958 0.23828 0.970634 0.902208 0.85092 0.266666 0.53976 0.375207 0.760249 0.
512535 0.667724 0.531606 0.0392803 0.437638 0.931835 0.93081 0.720952 0.2842
93 0.738534 0.639979 0.354049 0.687861 0.165974 0.440105 0.880075
Sorted array: 0.0163006 0.020023 0.0392803 0.0630958 0.0641713 0.0697553 0.0
860558 0.108809 0.12979 0.137232 0.141603 0.156679 0.165974 0.192214 0.19755
1 0.218257 0.23828 0.242887 0.266666 0.277775 0.283315 0.284293 0.292517 0.2
96032 0.335223 0.348893 0.352458 0.354049 0.364784 0.375207 0.394383 0.40022
9 0.400944 0.437638 0.440105 0.457702 0.477397 0.493583 0.512535 0.512932 0.
513401 0.524287 0.525995 0.526745 0.531606 0.53976 0.55397 0.606969 0.61264
0.628871 0.635712 0.637552 0.639979 0.663227 0.667724 0.687861 0.717297 0.72
0952 0.738534 0.760249 0.76823 0.769914 0.771358 0.783099 0.79844 0.804177 0
.807725 0.839112 0.85092 0.880075 0.890233 0.891529 0.902208 0.911647 0.9161
95 0.919026 0.93081 0.931835 0.949327 0.95223 0.970634 0.972775 0.998924
Time difference is: 7.2901e-05
...Program finished with exit code 0
Press ENTER to exit console.

Batch Analysis:

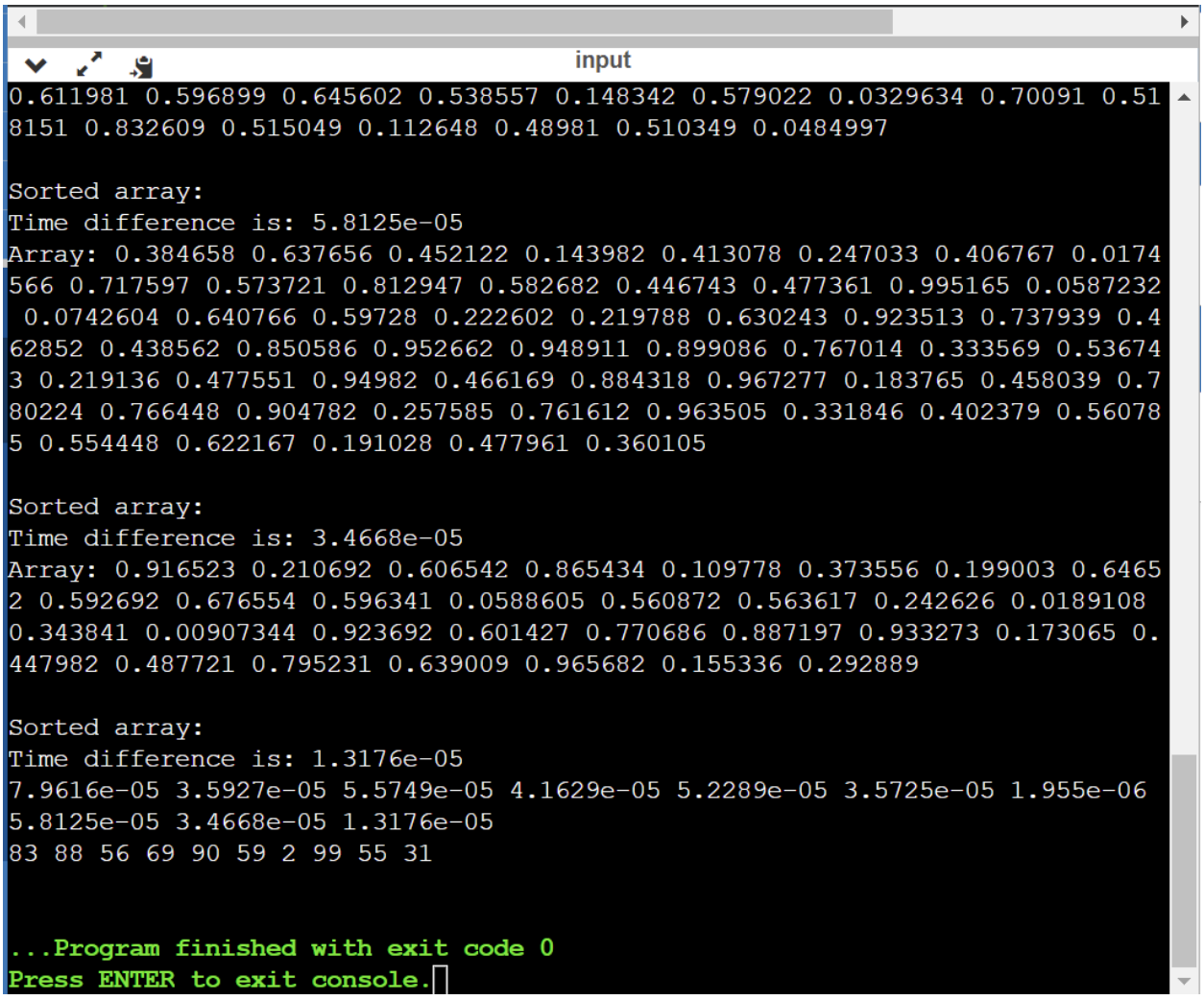
Source Code:

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

void bucketSort(float arr[], int n)
{
    vector<float> bucket[n];

    // put elemets in respective buckets
    for (int i = 0; i < n; i++)
    {
        int bi = n * arr[i]; // Index in bucket
        bucket[bi].push_back(arr[i]);
    }

    for (int i = 0; i < n; i++) // sorting each buckets
        sort(bucket[i].begin(), bucket[i].end());

    int index = 0;
    for (int i = 0; i < n; i++) // Concatenate all buckets into arr[]
        for (int j = 0; j < bucket[i].size(); j++)
            arr[index++] = bucket[i][j];
}

void printArray(double arr[], int size)
{
    for (int i = 0; i < size; i++)
```

```
        cout << arr[i] << " ";
    cout << endl;
}

void printArray(int arr[], int size)
{
    for (int i = 0; i < size; i++)
        cout << arr[i] << " ";
    cout << endl;
}

void printArray(float arr[], int size)
{
    for (int i = 0; i < size; i++)
        cout << arr[i] << " ";
    cout << endl;
}

int main()
{
    double times[10];
    int ns[10];
    for (int x = 0; x < 10; x++)
    {
        int n = rand() % 100;
        ns[x] = n;
        float arr[n];
        for (int i = 0; i < n; i++)
        {
            arr[i] = (float(rand()) / float((RAND_MAX)));
```

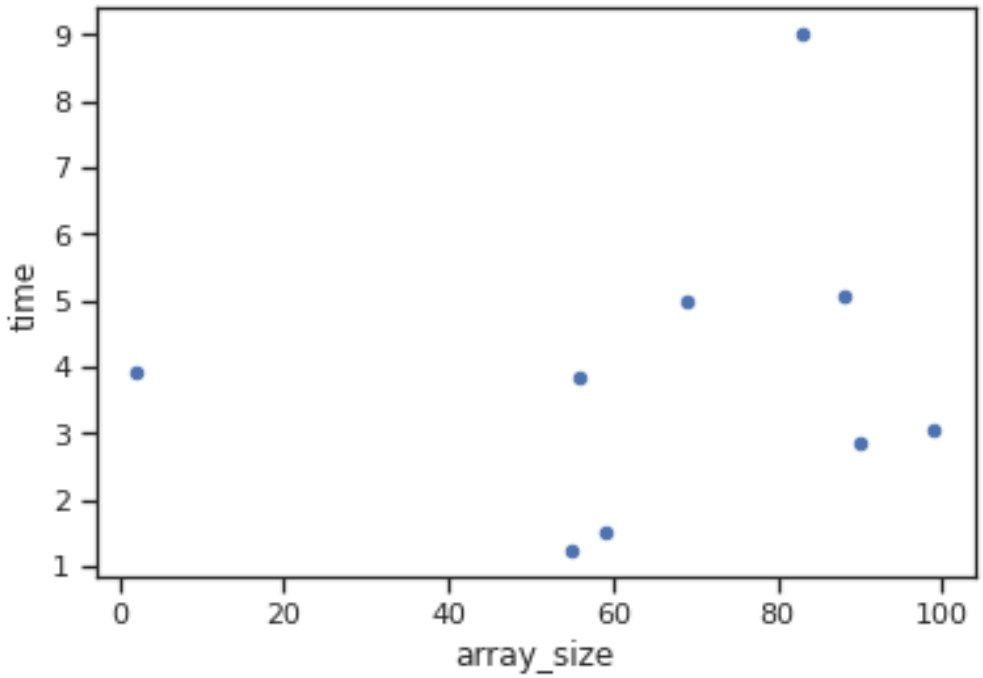
```
        }
        cout << "Array: ";
        printArray(arr, n);
        cout << endl;

        auto start = chrono::high_resolution_clock::now();
        // unsync the I/O of C and C++.
        ios_base::sync_with_stdio(false);

        bucketSort(arr, n); // sort arr

        auto end = chrono::high_resolution_clock::now();

        cout << "Sorted array: ";
        // printArray(arr, n);
        cout << endl;

        double time_taken = chrono::duration_cast<chrono::nanoseconds>(end -
start).count();
        time_taken *= 1e-9;

        times[x] = time_taken;
        cout << "Time difference is: " << time_taken << setprecision(6) << endl;
    }
    printArray(times, 10);
    printArray(ns, 10);
    return 0;
}
```

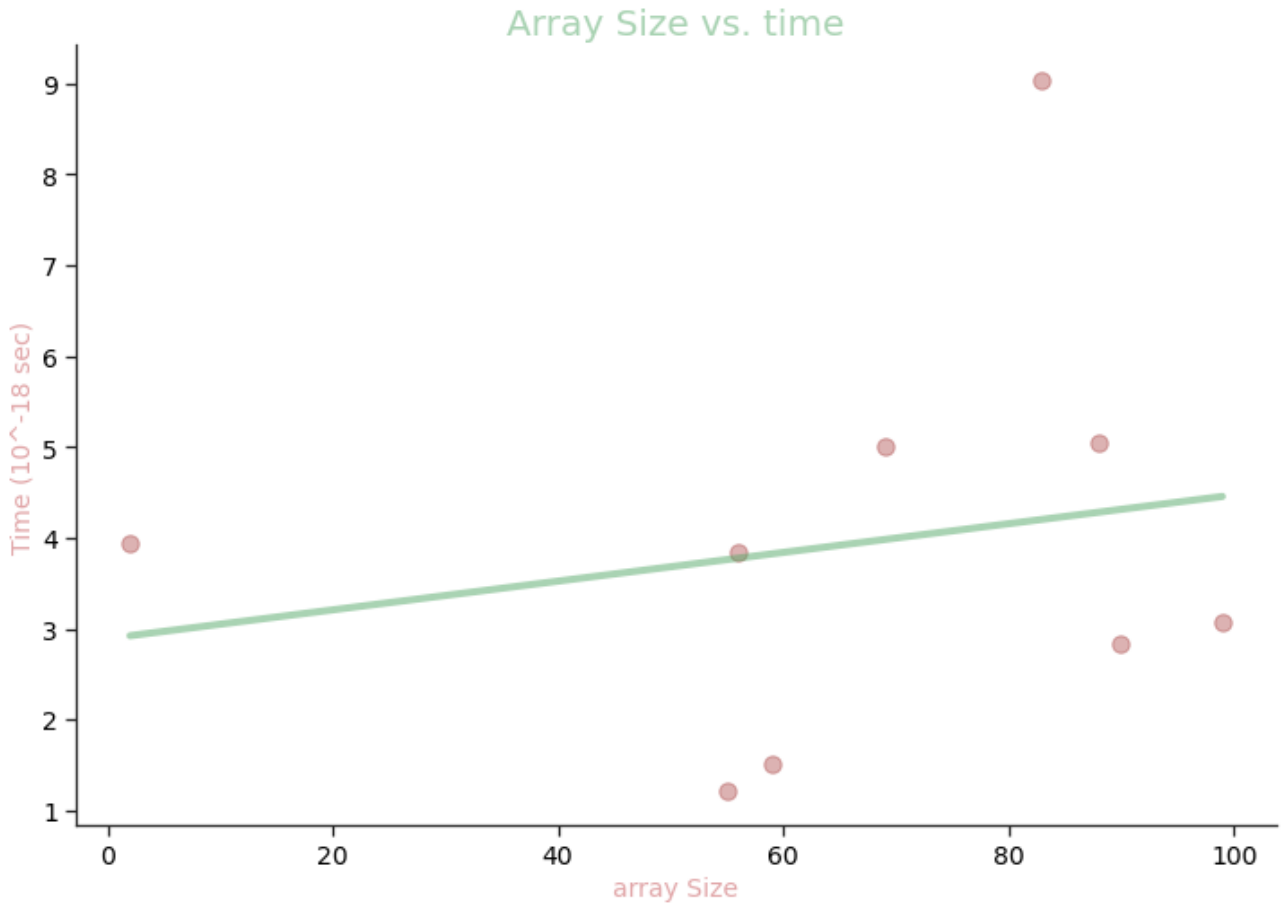
Output:



arraySize = [83, 88, 56, 69, 90, 59, 2, 99, 55, 31]

time = [9.0298, 5.0532, 3.8491, 5.0068, 2.8416, 1.505, 3.9323, 3.0733, 1.2236]


time
array_size
1
2
3
4
5
6
7
8
9
0
20
40
60
80
100


Array Size vs. time
Time (10^-18 sec)
array Size
1
2
3
4
5
6
7
8
9
0
20
40
60
80
100

Bucket Sort

Viva Questions

1. Is the bucket sort in place sort? Why or why not?

Ans.

No, it's not an in-place sorting algorithm. The whole idea is that input sorts themselves as they are moved to the buckets.

2. Is bucket sort a stable sort?

Ans.

Bucket sort is stable, if the underlying sort is also stable, as equal keys are inserted in order to each bucket. Counting sort works by determining how many integers are behind each integer in the input array A. Using this information, the input integer can be directly placed in the output array B.

3. Why bucket sort is good for large size arrays?

Ans.

Yes

The advantage of bucket sort is **that once the elements are distributed into buckets, each bucket can be processed independently of the others**. This means that you often need to sort much smaller arrays as a follow-up step than the original array.
Bucket sort works by distributing the array elements into a number of buckets. So bucket sort is most efficient in the case **when the input is uniformly distributed**.

4. State any disadvantage of using this sort.

Ans.

1. The problem is that if the buckets are distributed incorrectly, you may wind up spending a lot of extra effort for no or very little gain. As a result, bucket

sort works best when the data is more or less evenly distributed, or when there is a smart technique to pick the buckets given a fast set of heuristics based on the input array.

2. Can't apply it to all data types since a suitable bucketing technique is required. Bucket sort's efficiency is dependent on the distribution of the input values, thus it's not worth it if your data are closely grouped.In many situations, you might achieve greater performance by using a specialized sorting algorithm like radix sort, counting sort, or burst sort instead of bucket sort.
3. Bucket sort's performance is determined by the number of buckets used, which may need some additional performance adjustment when compared to other algorithms.

5. Can this sort be used for sorting negative numbers?

Ans.

Using Bucket sort for negative values simply **requires mapping each element to a bucket proportional to its** a distance from the minimal value to be sorted.

```
sortMixed(arr[], n)
1) Split array into two parts
   create two Empty vector Neg[], Pos[]
   (for negative and positive element respectively)
   Store all negative element in Neg[] by converting
   into positive (Neg[i] = -1 * Arr[i] )
   Store all +ve in pos[]  (pos[i] =  Arr[i])
2) Call function bucketSortPositive(Pos, pos.size())
   Call function bucketSortPositive(Neg, Neg.size())

bucketSortPositive(arr[], n)
3) Create n empty buckets (Or lists).
4) Do following for every array element arr[i].
       a) Insert arr[i] into bucket[n*array[i]]
5) Sort individual buckets using insertion sort.
6) Concatenate all sorted buckets.
```

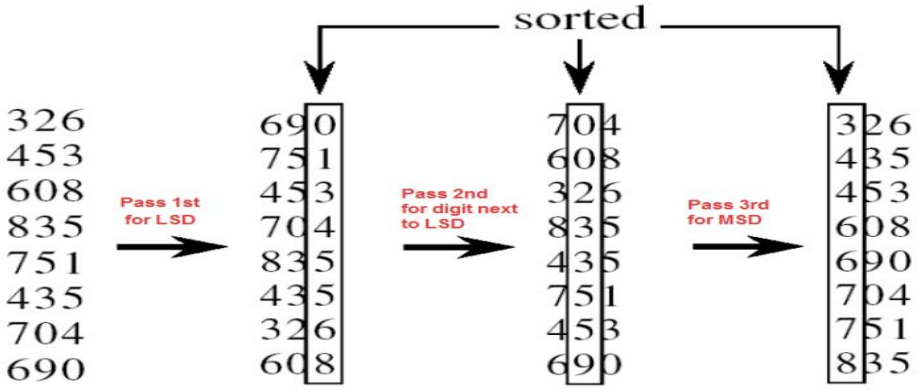
1.3 Radix Sort:

Radix Sort is a non-comparative sorting algorithm. It is one of the most efficient and fastest linear sorting algorithms. In radix sort, we first sort the elements based on last digit (least significant digit). Then the result is again sorted by second digit, continue this process for all digits until we reach most significant digit. We use counting sort to sort elements of every digit.

Pseudo code

```
Radix-Sort(A, d)
//It works same as counting sort for d number of passes.
//Each key in A [1...n] is a d-digit integer.
// (Digits are numbered 1 to d from right to left.)
    for j = 1 to d do
        //A[]-- Initial Array to Sort
        intcount[10] = {0};
        //Store the count of "keys" in count[]
        //key- it is number at digit place j
        for i = 0 to n do
         count[key of(A[i]) in pass j]++

        fork = 1 to 10 do
         count[k] = count[k] + count[k-1]

        //Build the resulting array by checking
        //new position of A[i] from count[k]
        for i = n-1 down to 0 do
         result[ count[key of(A[i])] ] = A[j]
         count[key of(A[i])]--

        //Now main array A[] contains sorted numbers
        //according to current digit place
        for i=0 to n do
          A[i] = result[i]

    end for(j)
end func
```

Example:



In the above example:
For 1st pass: We sort the array on basis of least significant digit (1s place) using counting sort. Notice that 43$\underline{5}$ is below 83$\underline{5}$, because 43$\underline{5}$ occurred below 83$\underline{5}$ in the original list.
For 2nd pass: We sort the array on basis of next digit (10s place) using counting sort. Notice that here 6$\underline{0}$8 is below 7$\underline{0}$4, because 6$\underline{0}$8 occurred below 7$\underline{0}$4 in the previous list, and similarly for (8$\underline{3}$5, 4$\underline{3}$5) and (7$\underline{5}$1, 4$\underline{5}$3).
For 3rd pass: We sort the array on basis of most significant digit (100s place) using counting sort. Notice that here $\underline{4}$35 is below $\underline{4}$53, because $\underline{4}$35 occurred below $\underline{4}$53 in the previous list, and similarly for ($\underline{6}$08, $\underline{6}$90) and ($\underline{7}$04, $\underline{7}$51).

Result and Analysis

In this algorithm running time depends on intermediate sorting algorithm which is counting sort. If the range of digits is from 1 to k, then counting sort time complexity is O (n+k). There are d passes i.e. counting sort is called d time, so total time complexity is O (nd+nk) =O (nd). As k=O (n) and d is constant, so radix sort runs in linear time. It performs the same way for best case as well

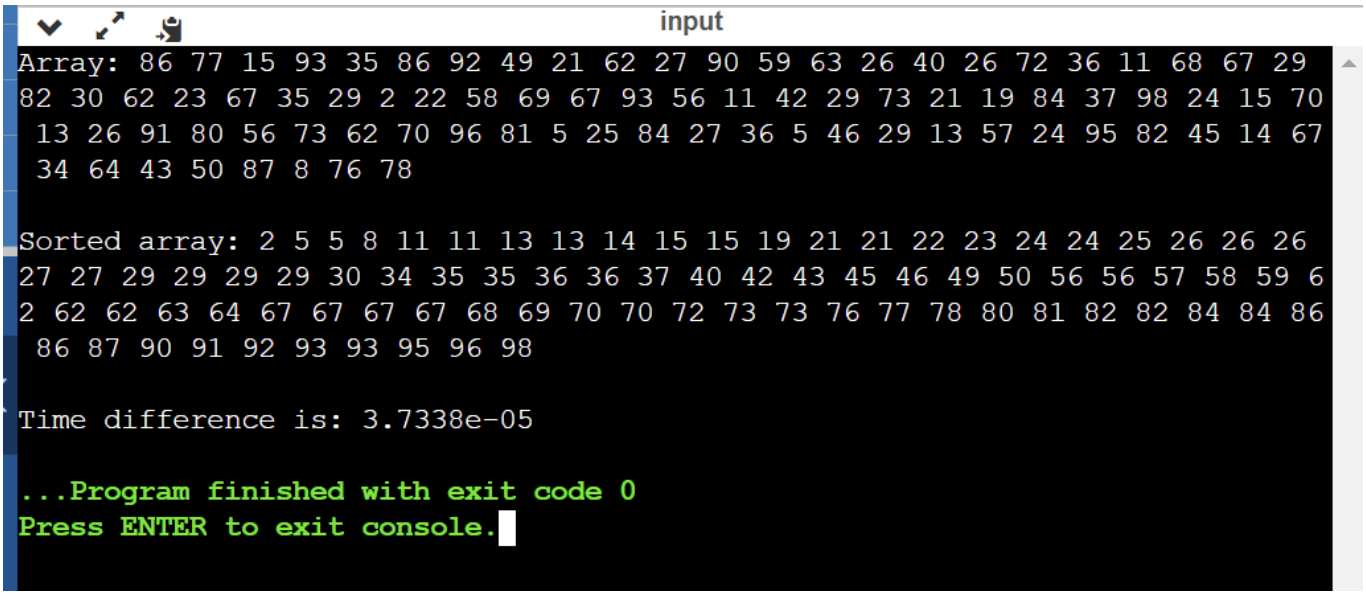
Initial Array	10	21	17	34	44	11	654	123
Sorted based on One's Place	10	21	11	123	34	44	654	17
Sorted based on Ten's Place	10	11	17	21	123	34	44	654
Sorted based on Hundred's Place	010	011	017	021	034	044	123	654

Source Code:

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

int getMax(int arr[], int n)    {
    int maxEle = arr[0];
    for (int i = 1; i < n; i++)
        if (arr[i] > maxEle)
            maxEle = arr[i];
    return maxEle;
}

void countSort(int arr[], int n, int exp)   {
    int output[n]; // output array
    int count[10] = { 0 };

    // Store count of occurrences in count[]
    for (int i = 0; i < n; i++)
        count[(arr[i] / exp) % 10]++;

    // Change count[i] so that count[i] now contains actual position of this
digit in output[]
    for (int i = 1; i < 10; i++)
        count[i] += count[i - 1];

    for (int i = n - 1; i >= 0; i--) {
        output[count[(arr[i] / exp) % 10] - 1] = arr[i];
        count[(arr[i] / exp) % 10]--;
    }

    // Copy the output array to arr[], so that arr[] now contains sorted numbers
according to current digit
    for (int i = 0; i < n; i++)
        arr[i] = output[i];
}

void radixsort(int arr[], int n)    {
    // Find the maximum number to know number of digits
    int m = getMax(arr, n);

    // Do counting sort for every digit. Note that instead of passing digit
number, exp is passed. exp is 10^i where i is current digit number
```

```
    for (int exp = 1; m / exp > 0; exp *= 10)
        countSort(arr, n, exp);
}

void printArray(int arr[], int size)    {
    for (int i = 0; i < size; i++)
        cout << arr[i] << " ";
    cout << endl;
}

int main()  {
    // int arr[] = {64, 34, 25, 12, 22, 11, 90};
    // int n = sizeof(arr)/sizeof(arr[0]);
    int n = rand() % 100;
    int arr[n];
    for (int i = 0; i < n; i++) {
        arr[i] = rand() % 100;
    }
    cout << "Array: ";
    printArray(arr, n);
    cout << endl;

    auto start = chrono::high_resolution_clock::now();
    // unsync the I/O of C and C++.
    ios_base::sync_with_stdio(false);

    radixsort(arr, n);  // sort arr

    auto end = chrono::high_resolution_clock::now();

    cout << "Sorted array: ";
    printArray(arr, n);
    cout << endl;

    double time_taken = chrono::duration_cast<chrono::nanoseconds>(end -
start).count();
    time_taken *= 1e-9;

    cout << "Time difference is: " << time_taken << setprecision(6);
    return 0;
}
```

Output:


input
Array: 86 77 15 93 35 86 92 49 21 62 27 90 59 63 26 40 26 72 36 11 68 67 29
82 30 62 23 67 35 29 2 22 58 69 67 93 56 11 42 29 73 21 19 84 37 98 24 15 70
13 26 91 80 56 73 62 70 96 81 5 25 84 27 36 5 46 29 13 57 24 95 82 45 14 67
34 64 43 50 87 8 76 78
Sorted array: 2 5 5 8 11 11 13 13 14 15 15 19 21 21 22 23 24 24 25 26 26 26
27 27 29 29 29 29 30 34 35 35 36 36 37 40 42 43 45 46 49 50 56 56 57 58 59 6
2 62 62 63 64 67 67 67 67 68 69 70 70 72 73 73 76 77 78 80 81 82 82 84 84 86
86 87 90 91 92 93 93 95 96 98
Time difference is: 3.7338e-05
...Program finished with exit code 0
Press ENTER to exit console.

Batch Analysis:

Source Code:

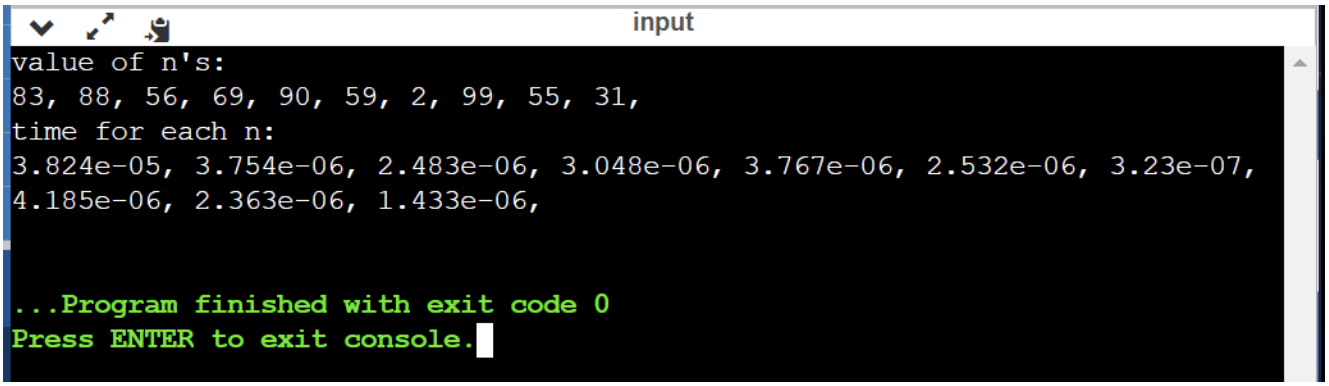
```
#include <bits/stdc++.h>
#include <chrono>
using namespace std;

int getMax(int arr[], int n)    {
    int maxEle = arr[0];
    for (int i = 1; i < n; i++)
        if (arr[i] > maxEle)
            maxEle = arr[i];
    return maxEle;
}

void countSort(int arr[], int n, int exp)   {
    int output[n]; // output array
    int count[10] = { 0 };

    // Store count of occurrences in count[]
    for (int i = 0; i < n; i++)
        count[(arr[i] / exp) % 10]++;

    // Change count[i] so that count[i] now contains actual position of this
digit in output[]
    for (int i = 1; i < 10; i++)
        count[i] += count[i - 1];

    for (int i = n - 1; i >= 0; i--) {
        output[count[(arr[i] / exp) % 10] - 1] = arr[i];
        count[(arr[i] / exp) % 10]--;
```

```
    }


    // Copy the output array to arr[], so that arr[] now contains sorted numbers
according to current digit
    for (int i = 0; i < n; i++)
        arr[i] = output[i];
}


void radixsort(int arr[], int n)    {
    // Find the maximum number to know number of digits
    int m = getMax(arr, n);


    // Do counting sort for every digit. Note that instead of passing digit
number, exp is passed. exp is 10^i where i is current digit number
    for (int exp = 1; m / exp > 0; exp *= 10)
        countSort(arr, n, exp);
}


void printArray(double arr[], int size)
{
    for (int i = 0; i < size; i++)
        cout << arr[i] << ", ";
    cout << endl;
}


void printArray(int arr[], int size)
{
    for (int i = 0; i < size; i++)
        cout << arr[i] << ", ";
    cout << endl;
}
```

```
void printArray(float arr[], int size)
{
    for (int i = 0; i < size; i++)
        cout << arr[i] << ", ";
    cout << endl;
}

double sortApp(int n)  {
    int arr[n];
    for (int i = 0; i < n; i++) {
        arr[i] = rand() % 100;
    }

    auto start = chrono::high_resolution_clock::now();
    ios_base::sync_with_stdio(false);

      radixsort(arr, n);  // sort arr

    auto end = chrono::high_resolution_clock::now();

    double time_taken = chrono::duration_cast<chrono::nanoseconds>(end -
start).count();
    time_taken *= 1e-9;

      return time_taken;
}

int main() {
    double times[10];
    int ns[10];
```

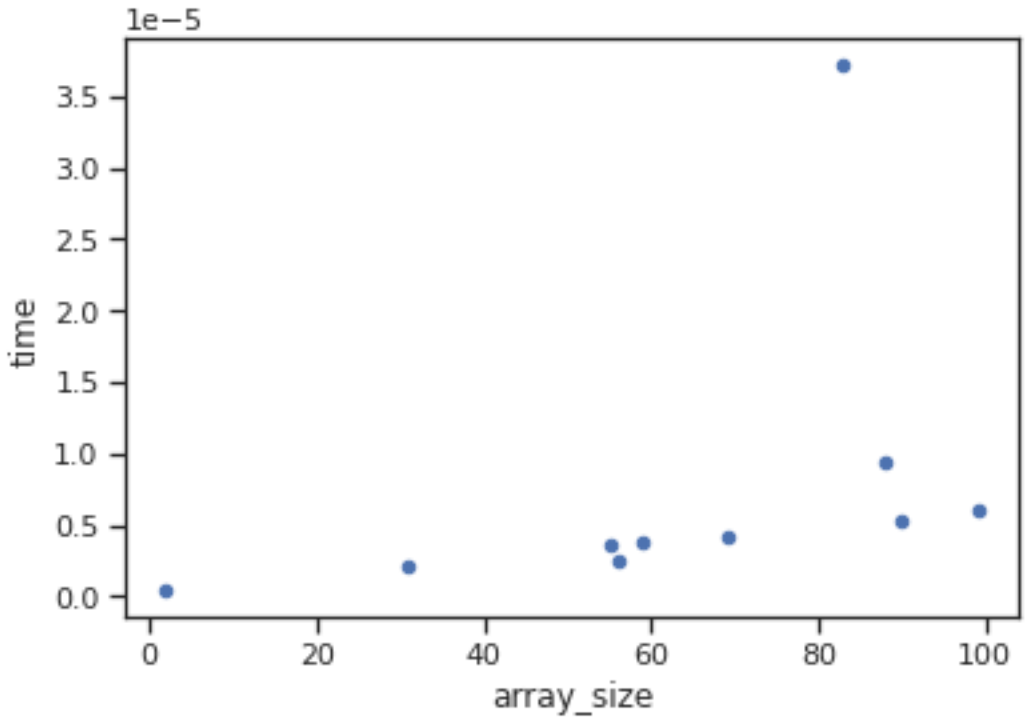
```
    for (int x = 0; x < 10; x++)
    {
        int n = rand() % 100;
        ns[x] = n;
        times[x] = sortApp(n);
    }
    cout << "value of n's: " << endl;
    printArray(ns, 10);
    cout << "time for each n: " << endl;
    printArray(times, 10);
}
```

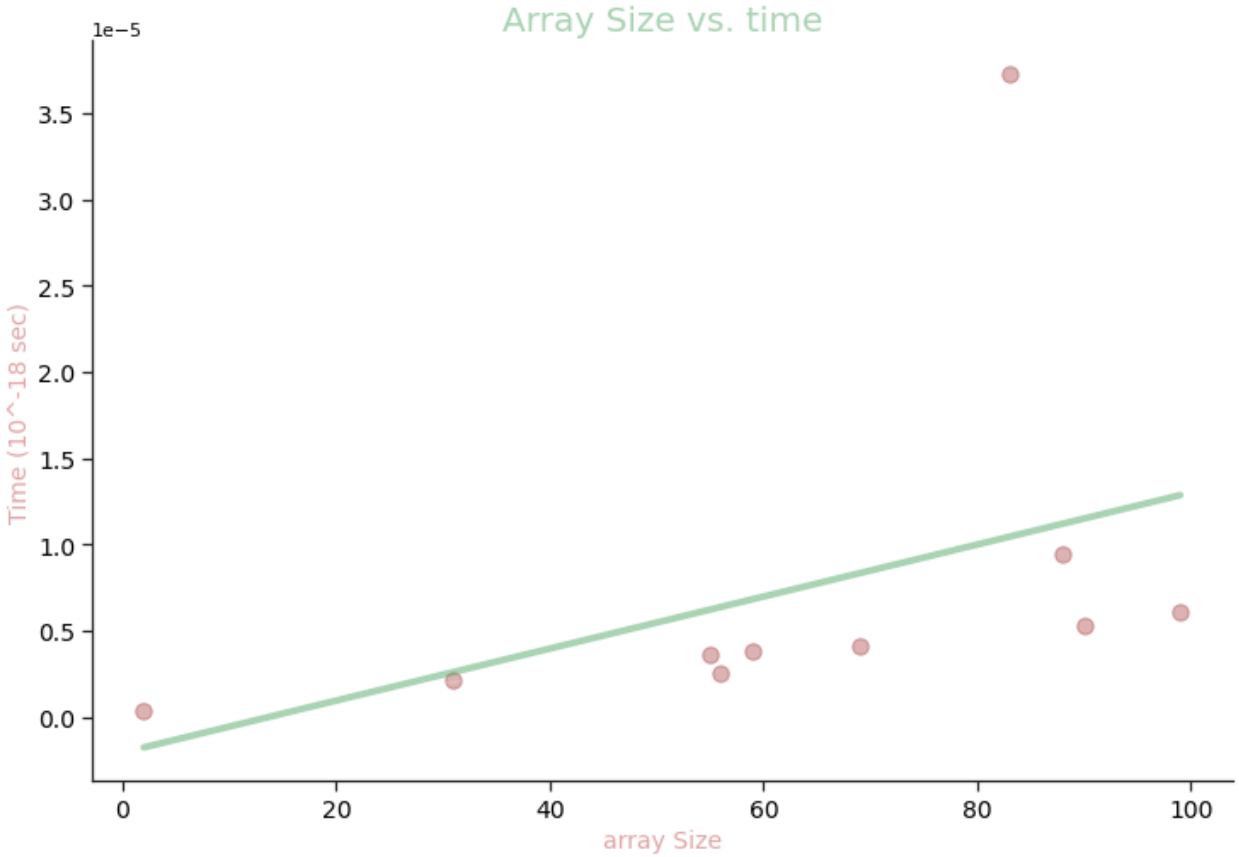
Output:



arraySize = [83, 88, 56, 69, 90, 59, 2, 99, 55, 31]

time = [3.7257e-05, 9.437e-06, 2.511e-06, 4.097e-06, 5.246e-06, 3.785e-06, 3.63e-07, 6.097e-06, 3.63e-06, 2.104e-06]


1e−5
3.5
3.0
2.5
2.0
1.5
1.0
0.5
0.0
time
0
20
40
60
80
100
array_size


Array Size vs. time
1e−5
3.5
3.0
2.5
2.0
1.5
1.0
0.5
0.0
Time (10^-18 sec)
0
20
40
60
80
100
array Size

Radix Sort

Viva Questions

1. Is Radix Sort preferable to Comparison based sorting algorithms like Quick-Sort?

Ans.
If we have $\log_2 n$ bits for every digit, the **running time of Radix appears to be better than Quick Sort** for a wide range of input numbers. The constant factors hidden in asymptotic notation are higher for Radix Sort and Quick-Sort uses hardware caches more effectively.

Radix sort is slower for (most) real world use cases.

One reason is the complexity of the algorithm:

If items are unique, k >= log(n). Even with duplicate items, the set of problems where k < log(n) is small.

Another is the implementation:

The additional memory requirement (which in it self is a disadvantage), affects cache performance negatively.

2. Is this sort stable?

Ans.

No

MSD sorts are not necessarily stable if the original ordering of duplicate keys must always be maintained. Other than the traversal order, MSD and LSD sorts differ in their handling of variable length input. LSD sorts can group by length, radix sort each group, then concatenate the groups in size order.

3. How much additional space is taken by this sort?

Ans.

O(k + n)

Radix sort's **space complexity is bound to the sort it uses to sort each radix**. In best case, that is counting sort. As you can see, counting sort creates multiple arrays, one based on the size of K, and one based on the size of N. B is the output array which is size n.

4. State any disadvantage of using this sort?

Ans.

- Since Radix Sort depends on digits or letters, Radix Sort is much less flexible than other sorts. Hence , for every different type of data it needs to be rewritten.
- The constant for Radix sort is greater compared to other sorting algorithms.
- It takes more space compared to Quicksort which is inplace sorting.

5. How can the performance of this sort be improved?

Ans.

Bucket sort is mainly useful when input is uniformly distributed over a range. For example, consider the following problem.
Sort a large set of floating point numbers which are in range from 0.0 to 1.0 and are uniformly distributed across the range. How do we sort the numbers efficiently?
A simple way is to apply a comparison based sorting algorithm. The lower bound for Comparison based sorting algorithm (Merge Sort, Heap Sort, Quick-Sort .. etc) is Ω(n Log n), i.e., they cannot do better than nLogn.
Can we sort the array in linear time? Counting sort can not be applied here as we use keys as index in counting sort. Here keys are floating point numbers.
The idea is to use bucket sort.

1.4 Shell Sort:

It is a generalized version of insertion sort. It is an in–place comparison sort. It is also known as **diminishing increment sort**; it is one of the oldest sorting algorithms invented by Donald L. Shell (1959.)This algorithm uses insertion sort on the large interval of elements to sort. Then the interval of sorting keeps on decreasing in a sequence until the interval reaches 1. These intervals are known as **gap sequence.**

Increment Sequences:

Shell's original sequence: N/2 , N/4 , …, 1 (repeatedly divide by 2);

Hibbard's increments: 1, 3, 7, …, $2^k - 1$;

Knuth's increments: 1, 4, 13, …, $(3^k - 1) / 2$;

Sedge wick's increments: 1, 5, 19, 41, 109…

Here interval is calculated based on Knuth's formula as −

Knuth's Formula

h = h * 3 + 1, where ->h is interval with initial value 1

Pseudo code

```
Procedure shellSort ()
A: array of items
   /* calculate interval*/
   while interval < A. length /3 do:
      interval = interval * 3 + 1
   end while

   while interval > 0 do:
      for outer = interval; outer < A. length; outer ++ do:
      /* select value to be inserted */
ValueToInsert = A[outer]
      inner = outer;
         /*shift element towards right*/
         while inner > interval -1 &&A [inner - interval] >=
valueToInsert do:
```

```
            A[inner] = A [inner - interval]
            inner = inner - interval
         end while
      /* insert the number at hole position */
      A[inner] = valueToInsert
      end for
   /* calculate interval*/
   interval = (interval -1) /3;
   end while
end procedure
```

Example:

	a_1	a_2	a_3	a_4	a_5	a_6	a_7	a_8	a_9	a_{10}	a_{11}	a_{12}
Input data	62	83	18	53	07	17	95	86	47	69	25	28
After 5-sorting	17	28	18	47	07	25	83	86	53	69	62	95
After 3-sorting	17	07	18	47	28	25	69	62	53	83	86	95
After 1-sorting	07	17	18	25	28	47	53	62	69	83	86	95

The first pass, 5-sorting, performs insertion sort on five separate sub arrays (a_1, a_6, a_{11}), (a_2, a_7, a_{12}), (a_3, a_8), (a_4, a_9), (a_5, a_{10}). For instance, it changes the sub array (a_1, a_6, a_{11}) from (62, 17, 25) to (17, 25, 62). The next pass, 3-sorting, performs insertion sort on the three sub arrays (a_1, a_4, a_7, a_{10}), (a_2, a_5, a_8, a_{11}), (a_3, a_6, a_9, a_{12}). The last pass, 1-sorting, is an ordinary insertion sort of the entire array (a_1... a_{12}).

Result and Analysis

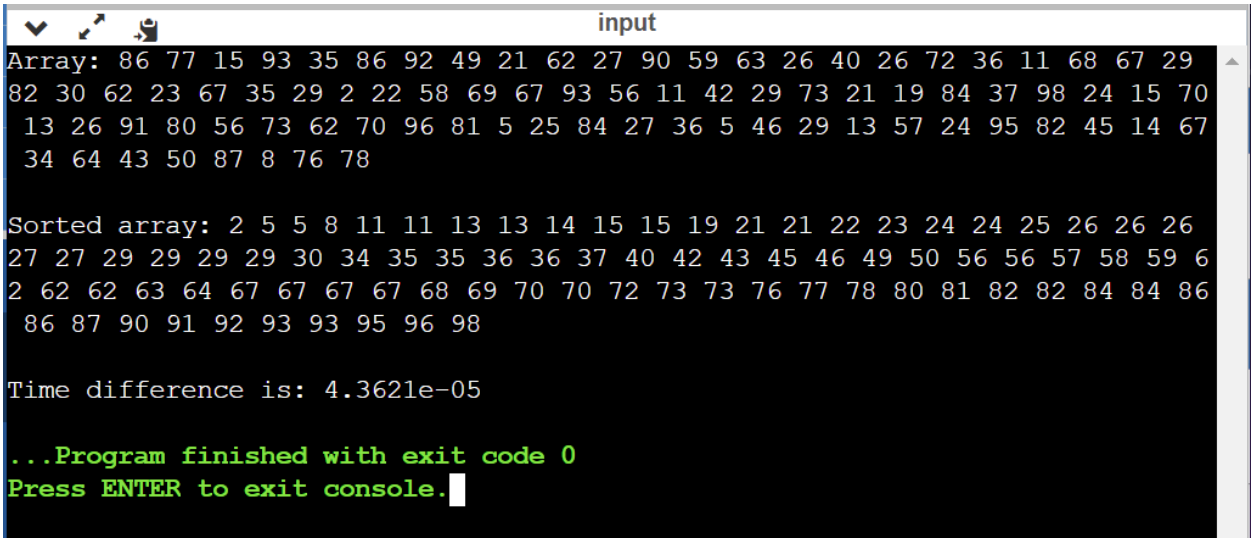
Since in this algorithm insertion sort is applied in the large interval of elements and then interval reduces in a sequence, therefore the running time of Shell sort is heavily dependent on the gap sequence it uses. So in worst case time complexity is $O(n^2)$ and in Average Case time complexity depends on gap sequence while in Best Case Time complexity is O(n*log n)

Source Code:

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

int shellSort(int arr[], int n) {
    for (int gap = n/2; gap > 0; gap /= 2)  {
        for (int i = gap; i < n; i += 1)    {
            int temp = arr[i];

            // shift gap sorted elements to find position for ith element
            int j = i;
            for (j; j >= gap && arr[j - gap] > temp; j -= gap)
                arr[j] = arr[j - gap];
            arr[j] = temp; // place i th element in correct position

        }
    }
    return 0;
}

void printArray(int arr[], int size)    {
    for (int i = 0; i < size; i++)
        cout << arr[i] << " ";
    cout << endl;
}

int main()  {
    // int arr[] = {64, 34, 25, 12, 22, 11, 90};
    // int n = sizeof(arr)/sizeof(arr[0]);

    int n = rand() % 100;
    int arr[n];

    for (int i = 0; i < n; i++) {
        arr[i] = rand() % 100;
    }

    cout << "Array: ";
    printArray(arr, n);
    cout << endl;

    auto start = chrono::high_resolution_clock::now();
```

```
    // unsync the I/O of C and C++.
    ios_base::sync_with_stdio(false);

    shellSort(arr, n);  // sort arr

    auto end = chrono::high_resolution_clock::now();

    cout << "Sorted array: ";
    printArray(arr, n);
    cout << endl;

    double time_taken = chrono::duration_cast<chrono::nanoseconds>(end -
start).count();
    time_taken *= 1e-9;

    cout << "Time difference is: " << time_taken << setprecision(6);
    return 0;
}
```

Output:



Batch Analysis:

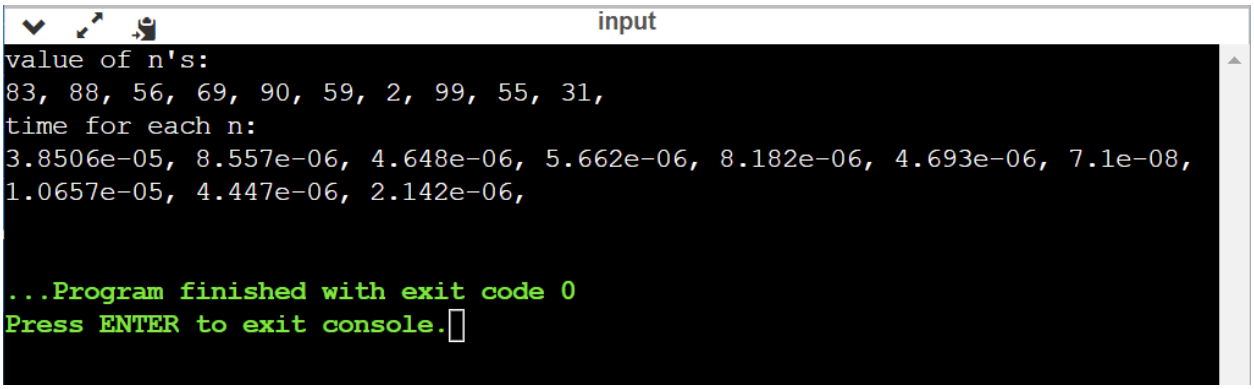
Source Code:

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

int shellSort(int arr[], int n) {
    for (int gap = n/2; gap > 0; gap /= 2)  {
        for (int i = gap; i < n; i += 1)    {
            int temp = arr[i];

            // shift gap sorted elements to find position for ith element
            int j = i;
            for (j; j >= gap && arr[j - gap] > temp; j -= gap)
                arr[j] = arr[j - gap];
            arr[j] = temp; // place i th element in correct position
        }
    }
    return 0;
}

void printArray(double arr[], int size)
{
    for (int i = 0; i < size; i++)
        cout << arr[i] << ", ";
    cout << endl;
}

void printArray(int arr[], int size)
```

```
{
    for (int i = 0; i < size; i++)
        cout << arr[i] << ", ";
    cout << endl;
}

void printArray(float arr[], int size)
{
    for (int i = 0; i < size; i++)
        cout << arr[i] << ", ";
    cout << endl;
}

double sortApp(int n)  {
    int arr[n];
    for (int i = 0; i < n; i++) {
        arr[i] = rand() % 100;
    }

    auto start = chrono::high_resolution_clock::now();
    ios_base::sync_with_stdio(false);

      shellSort(arr, n);  // sort arr

    auto end = chrono::high_resolution_clock::now();

    double time_taken = chrono::duration_cast<chrono::nanoseconds>(end -
start).count();
    time_taken *= 1e-9;

      return time_taken;
```

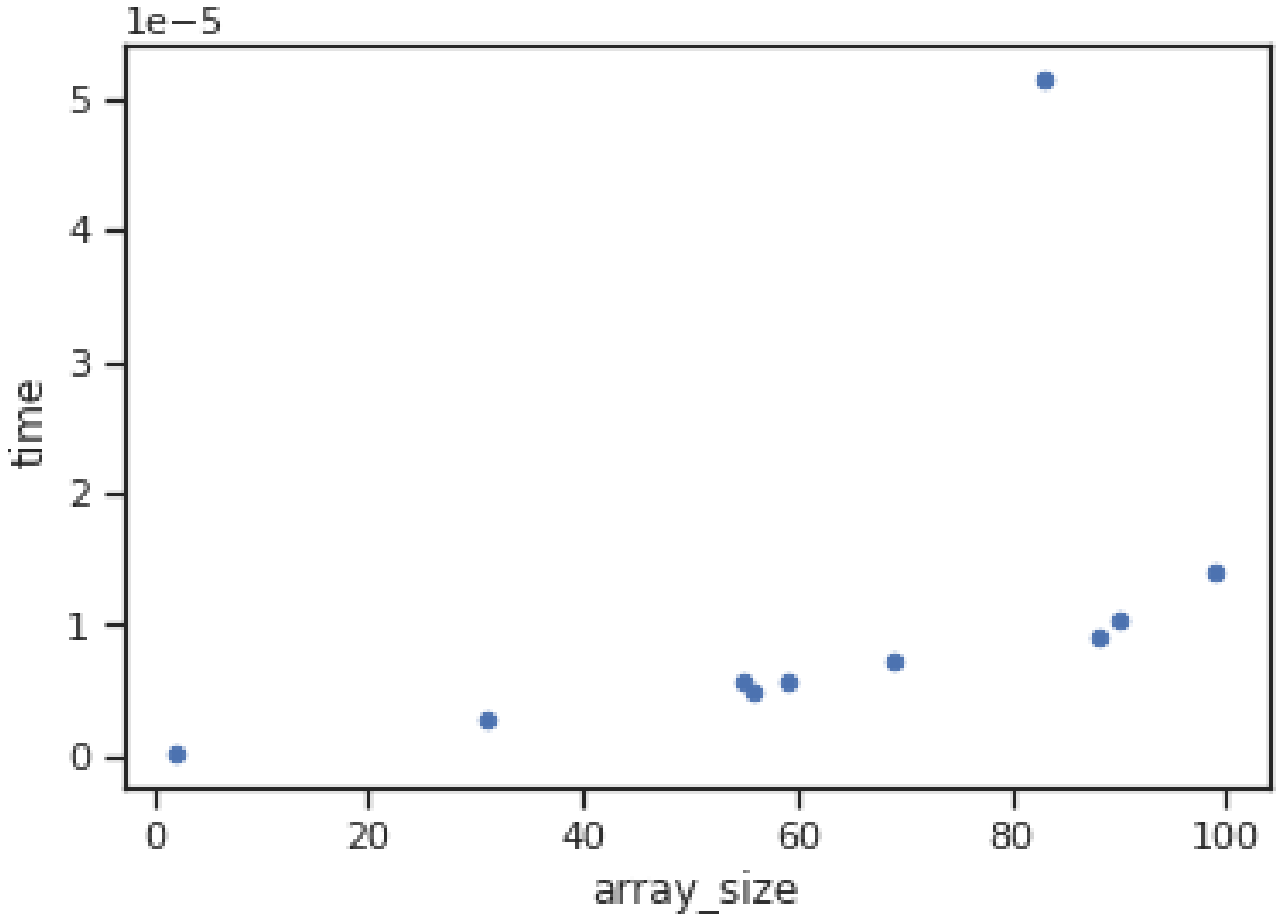
```
}

int main() {
    double times[10];
    int ns[10];
    for (int x = 0; x < 10; x++)
    {
        int n = rand() % 100;
        ns[x] = n;
        times[x] = sortApp(n);
    }
    cout << "value of n's: " << endl;
    printArray(ns, 10);
    cout << "time for each n: " << endl;
    printArray(times, 10);
}
```

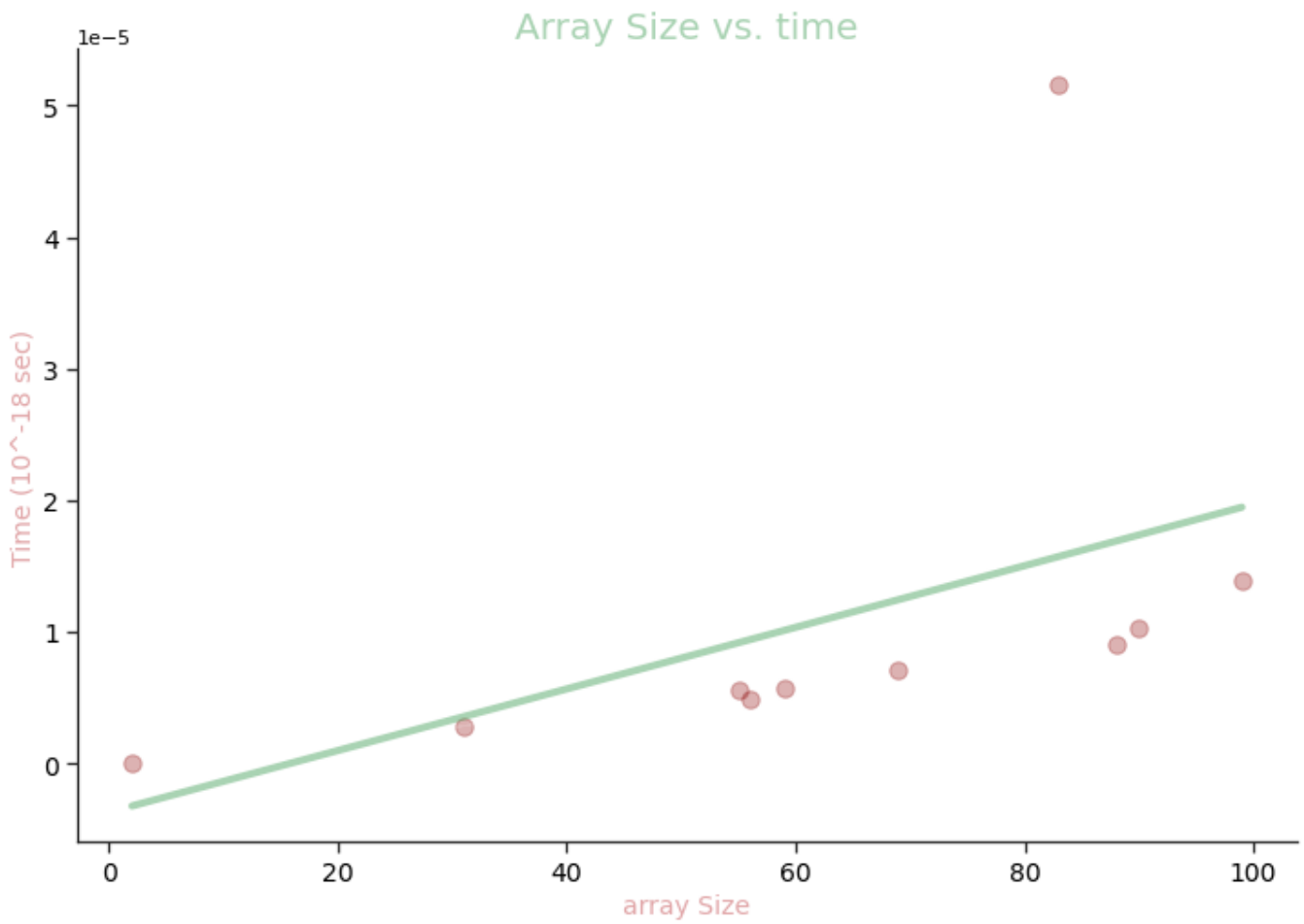
Output:



arraySize = [83, 88, 56, 69, 90, 59, 2, 99, 55, 31]

time = [5.1556e-05, 9.109e-06, 4.9e-06, 7.176e-06, 1.0281e-05, 5.658e-06, 9e-08, 1.3916e-05, 5.566e-06, 2.774e-06]




Array Size vs. time
1e−5
5
4
3
2
1
0
Time (10^-18 sec)
0
20
40
60
80
100
array Size

Shell Sort

Viva Questions

1. How can the time complexity of shell sort be improved?

Ans.

Shell Sort improves its time complexity by taking the advantage of the fact that using Insertion Sort on a partially sorted array results in less number of moves.

2. State any application of shell sort?

Ans.

- Shellsort performs more operations and has higher cache miss ratio than quicksort.

- However, since it can be implemented using little code and does not use the call stack, some implementations of the qsort function in the C standard library targeted at embedded systems use it instead of quicksort. Shellsort is, for example, used in the uClibc library. For similar reasons, an implementation of Shellsort is present in the Linux kernel.

- Shellsort can also serve as a sub-algorithm of introspective sort, to sort short subarrays and to prevent a slowdown when the recursion depth exceeds a given limit. This principle is employed, for instance, in the bzip2 compressor.

3. Given the following list of numbers: [5,16,20,12,3,8,9,17,19,7]. What is the content of the list after all swapping is complete for a gap size of 3?

Ans.

[5, 3, 8, 7, 16, 19, 9, 17, 20, 12]

4. Is this sort stable and does it takes extra additional space?

Ans.

Yes, Shell sort is a stable Algorithm.

No, No addition space is required. Shell sort is an in-place sorting algorithm as it requires **O(1)** auxiliary space.

1.5 Selection Sort:

Selection sort is a simple sorting algorithm. This sorting algorithm is an in-place comparison-based algorithm in which the list is divided into two parts, the sorted part at the left end and the unsorted part at the right end. Initially, the sorted part is empty and the unsorted part is the entire list. The smallest element is selected from the unsorted array and swapped with the leftmost element, and that element becomes a part of the sorted array. This process continues moving unsorted array boundary by one element to the right.

The selection sort algorithm sorts an array by repeatedly finding the minimum element (considering ascending order) from unsorted part and putting it at the beginning. The algorithm maintains two subarrays in a given array.

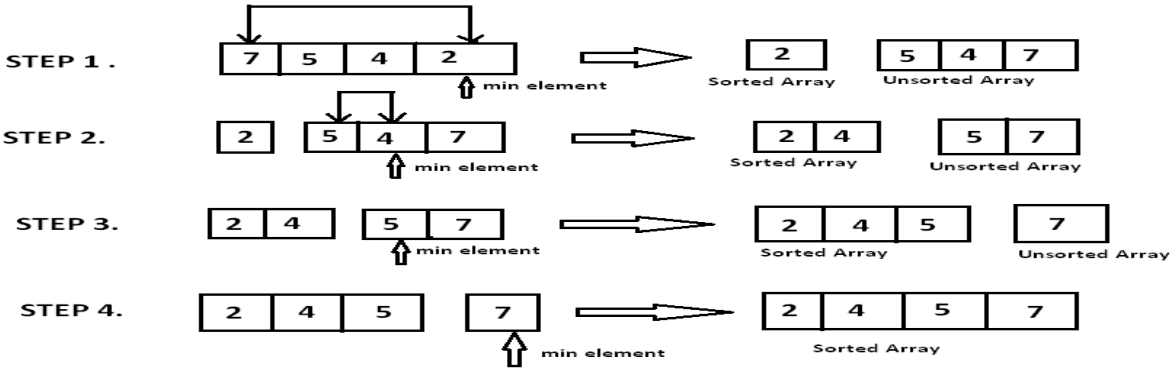
1) The subarray which is already sorted.

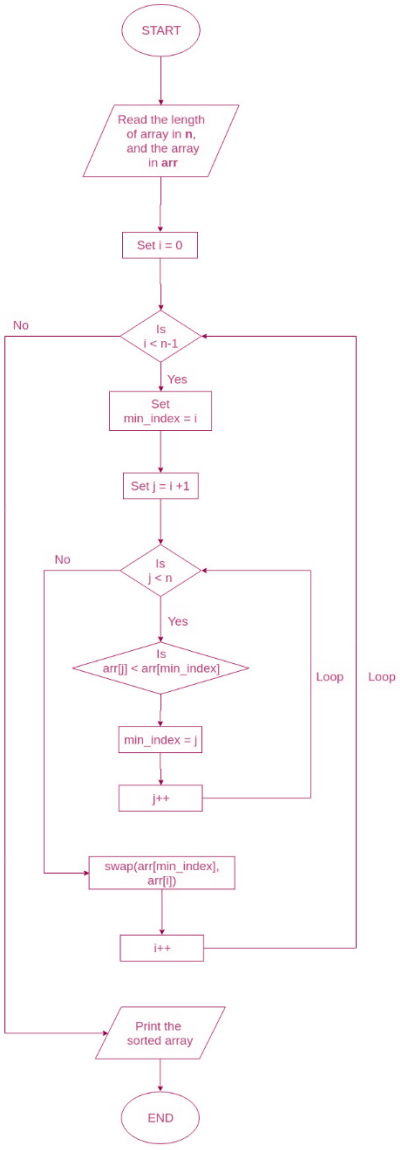
2) Remaining subarray which is unsorted.

Pseudo code

```
Procedure selection sort
list: array of items
  n: size of list
   for i = 1 to n - 1
   /* set current element as minimum*/
      min = i
      /* check the element to be minimum */
      for j = i+1 to n
    if list[j] < list [min] then
min = j;
         end if
end for
      /* swap the minimum element with the current element*/
      if indexMin != i  then
         swap list[min] and list[i]
      end if
   end for
end procedure:
```

Example:





Flowchart for Selection Sort

Result and Analysis

To find the minimum element from the array of N elements, N−1 comparisons are required. After putting the minimum element in its proper position, the size of an unsorted array reduces to N−1 and then N−2 comparisons are required to find the minimum in the unsorted array. Therefore (N−1) + (N−2) + ... + 1 = (N (N−1))/2 comparisons and N swaps result in the overall complexity of O (N2

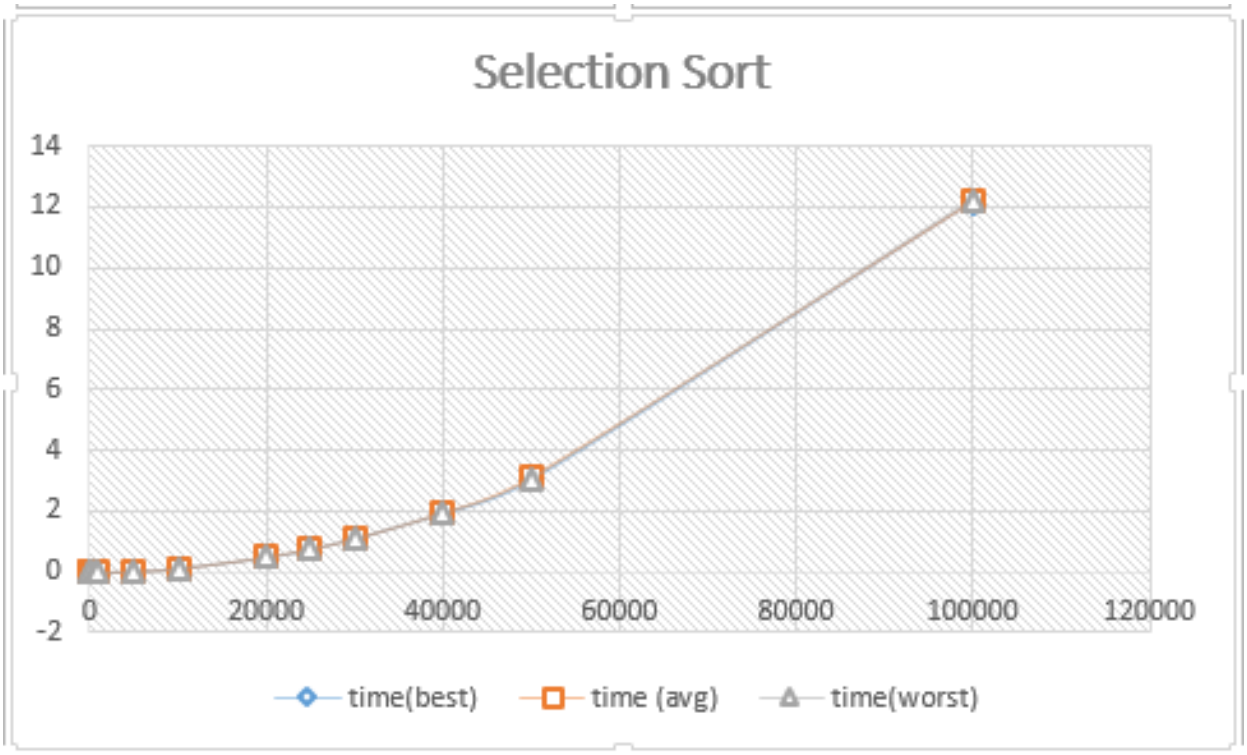
Source Code:

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

void swap(int *xp, int *yp)
{
    int temp = *xp;
    *xp = *yp;
    *yp = temp;
}

// keep finding minimum element and place it in beginning
void selectionSort(int arr[], int n)
{
    for (int i = 0; i < n - 1; i++)
    {
        int min_idx = i;
        for (int j = i + 1; j < n; j++)
            if (arr[j] < arr[min_idx])
                min_idx = j;
        swap(&arr[min_idx], &arr[i]); // swap min with first
    }
}

void printArray(int arr[], int size)
{
    int i;
    for (i = 0; i < size; i++)
        cout << arr[i] << " ";
    cout << endl;
```

```
}

int main()
{
    // int arr[] = {64, 34, 25, 12, 22, 11, 90};
    // int n = sizeof(arr)/sizeof(arr[0]);

    int n = rand() % 100;
    int arr[n];

    for (int i = 0; i < n; i++)
    {
        arr[i] = rand() % 100;
    }

    cout << "Array: ";
    printArray(arr, n);
    cout << endl;

    auto start = chrono::high_resolution_clock::now();

    // unsync the I/O of C and C++.
    ios_base::sync_with_stdio(false);

    selectionSort(arr, n);

    auto end = chrono::high_resolution_clock::now();

    cout << "Sorted array: ";
    printArray(arr, n);
    cout << endl;

    double time_taken = chrono::duration_cast<chrono::nanoseconds>(end -
start).count();
    time_taken *= 1e-9;

    cout << "Time difference is: " << time_taken << setprecision(6);

    return 0;
}
```

Output:

```
input
Array: 86 77 15 93 35 86 92 49 21 62 27 90 59 63 26 40 26 72 36 11 68 67 29
82 30 62 23 67 35 29 2 22 58 69 67 93 56 11 42 29 73 21 19 84 37 98 24 15 70
 13 26 91 80 56 73 62 70 96 81 5 25 84 27 36 5 46 29 13 57 24 95 82 45 14 67
 34 64 43 50 87 8 76 78

Sorted array: 2 5 5 8 11 11 13 13 14 15 15 19 21 21 22 23 24 24 25 26 26 26
27 27 29 29 29 29 30 34 35 35 36 36 37 40 42 43 45 46 49 50 56 56 57 58 59 6
2 62 62 63 64 67 67 67 67 68 69 70 70 72 73 73 76 77 78 80 81 82 82 84 84 86
 86 87 90 91 92 93 93 95 96 98

Time difference is: 4.2556e-05

...Program finished with exit code 0
Press ENTER to exit console.
```



Batch Analysis:

Source Code:

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

void swap(int *xp, int *yp)
{
    int temp = *xp;
    *xp = *yp;
    *yp = temp;
}

// keep finding minimum element and place it in beginning
void selectionSort(int arr[], int n)
{
    for (int i = 0; i < n - 1; i++)
    {
        int min_idx = i;
        for (int j = i + 1; j < n; j++)
            if (arr[j] < arr[min_idx])
                min_idx = j;
        swap(&arr[min_idx], &arr[i]); // swap min with first
    }
}

void printArray(double arr[], int size)
{
    for (int i = 0; i < size; i++)
```

```
        cout << arr[i] << ", ";
    cout << endl;
}

void printArray(int arr[], int size)
{
    for (int i = 0; i < size; i++)
        cout << arr[i] << ", ";
    cout << endl;
}

void printArray(float arr[], int size)
{
    for (int i = 0; i < size; i++)
        cout << arr[i] << ", ";
    cout << endl;
}

double sortApp(int n)
{
    int arr[n];
    for (int i = 0; i < n; i++)
    {
        arr[i] = rand() % 100;
    }

    auto start = chrono::high_resolution_clock::now();
    ios_base::sync_with_stdio(false);

    selectionSort(arr, n);
```

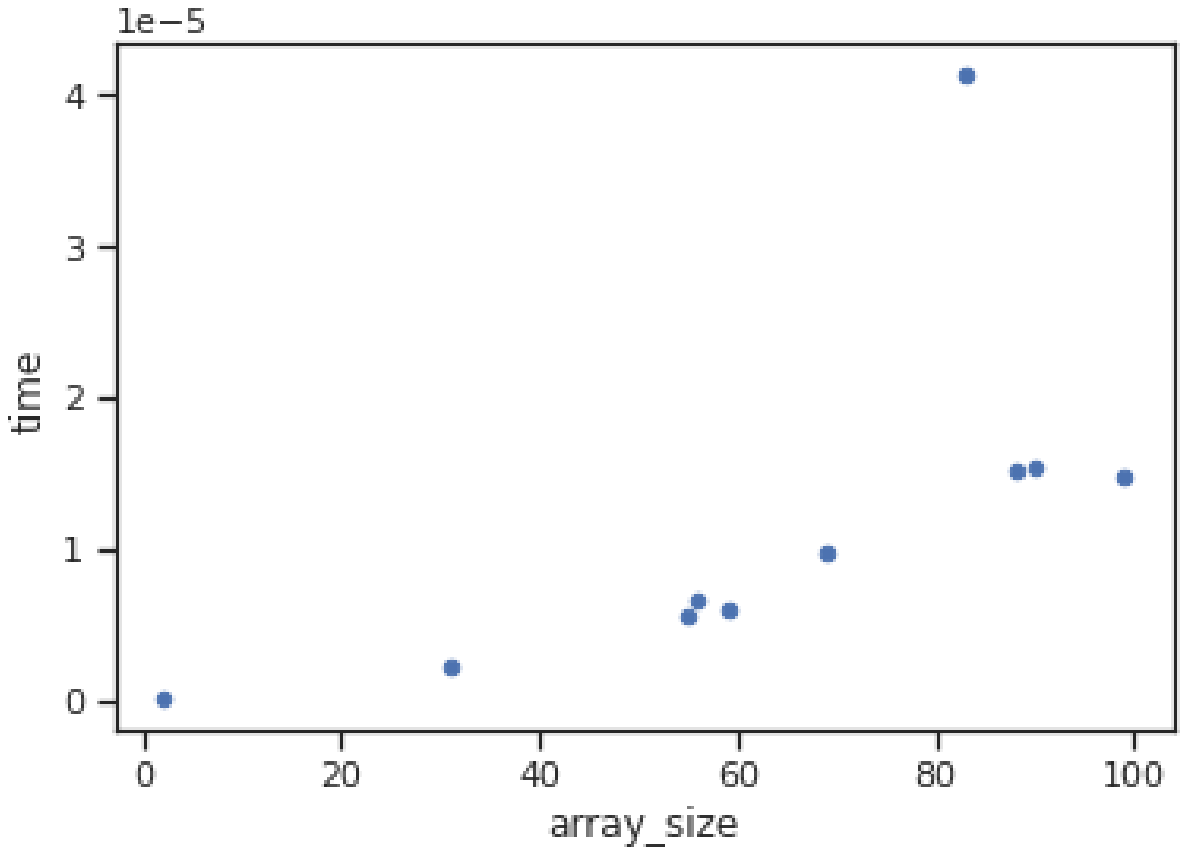
```
    auto end = chrono::high_resolution_clock::now();


    double time_taken = chrono::duration_cast<chrono::nanoseconds>(end -
start).count();
    time_taken *= 1e-9;


    return time_taken;
}


int main()
{
    double times[10];
    int ns[10];
    for (int x = 0; x < 10; x++)
    {
        int n = rand() % 100;
        ns[x] = n;
        times[x] = sortApp(n);
    }
    cout << "value of n's: " << endl;
    printArray(ns, 10);
    cout << "time for each n: " << endl;
    printArray(times, 10);
}
```

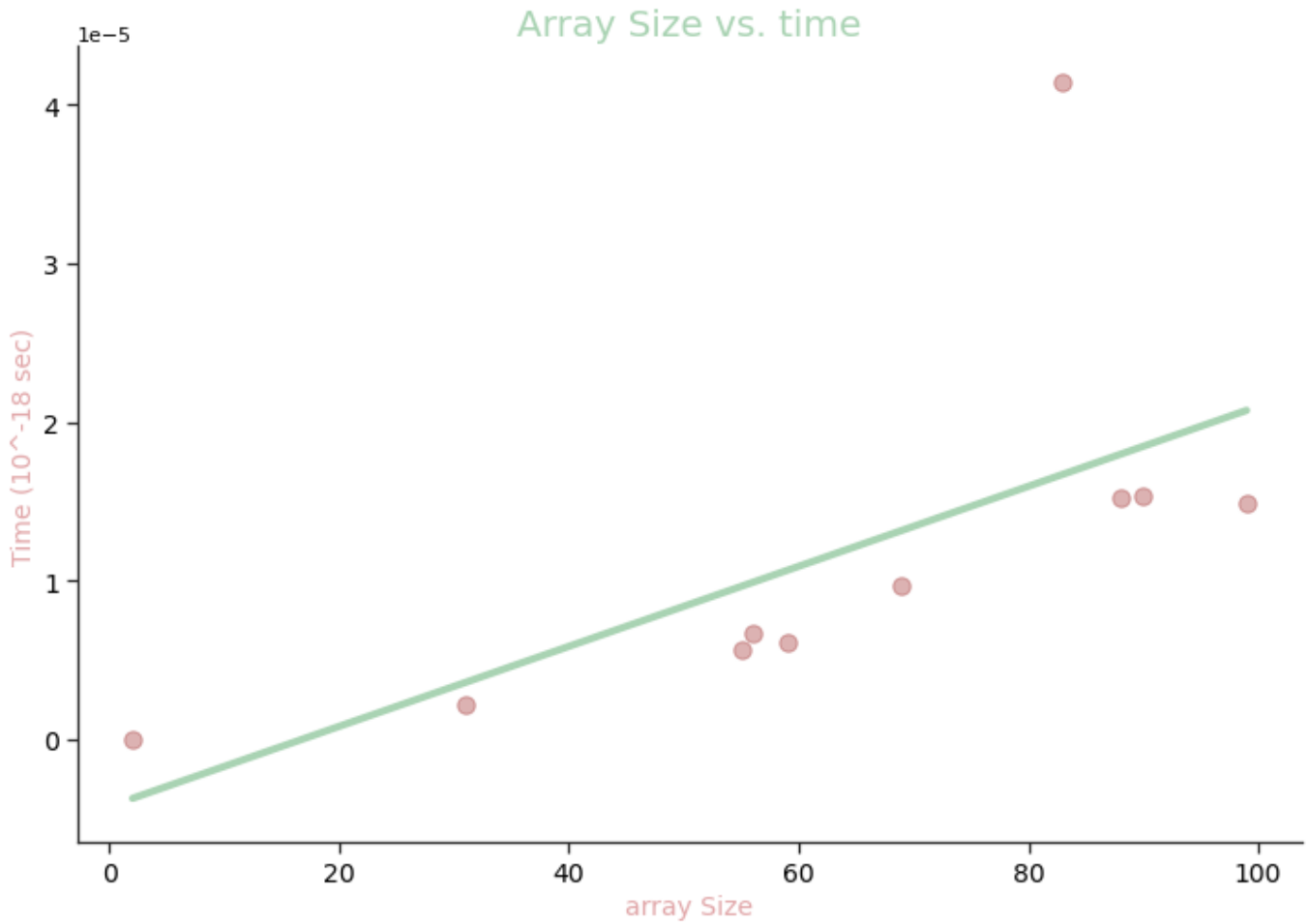
Output:

```
input
value of n's:
83, 88, 56, 69, 90, 59, 2, 99, 55, 31,
time for each n:
4.4366e-05, 1.6036e-05, 6.706e-06, 9.483e-06, 1.5256e-05, 7.488e-06, 8.3e-08
, 1.8461e-05, 6.788e-06, 2.525e-06,


...Program finished with exit code 0
Press ENTER to exit console.
```

arraySize = [83, 88, 56, 69, 90, 59, 2, 99, 55, 31]

time = [4.1399e-05, 1.5271e-05, 6.697e-06, 9.652e-06, 1.537e-05, 6.069e-06, 4.2e-08, 1.4875e-05, 5.673e-06, 2.172e-06]





Selection Sort

Viva Questions

1. What is the average case performance of Selection Sort?

Ans.

$\Theta(N^2)$

Average Case Time Complexity is: **$O(N^2)$**

2. For each i from 1 to n-1, how many comparisons are there in this sort?

Ans.

Selection sort finds the smallest element and swaps it into the first position. The, from the remaining N-1 elements after the first one (which we know is the smallest), it finds the next smallest and swaps it into the 2nd position. And so on.

To find the smallest element from a group of N elements takes N-1 comparisons. Basically, compare item 1 to 2, remember the smallest. Compare item 3 to smallest and remember which ever is smaller, and so on. You end up doing 1 comparison for every element from 2 through N.

We have to find the smallest element N-1 times*, but each time we have a smaller list to look through.

For N elements it takes 1+2+3+…+N-1 = (N-1)N/2 comparisons.

3. If all the input elements are identical then how it will affect the time taken by this sort?

Ans.

Selection sort will run in **O(n^2)** regardless of whether the array is already sorted or not.

4. What is the output of selection sort after the 2nd iteration given the following sequence of numbers: 16 3 46 9 28 14

Ans.

3 9 46 16 28 14

5. What is straight selection sort?

Ans.

Selection sorting refers to a class of algorithms for sorting a list of items using comparisons. These algorithms select successively smaller or larger items from the list and add them to the output sequence. This is an improvement of the Simple Selection Sort and it is called Straight Selection Sort. Therefore, instead of replacing the selected element by a unique value in the i-th pass (as happens in Simple Selection Sort), the selected element is exchanged with the i-th element.

1.6 Heap Sort:

This algorithm is divided into two basic parts:

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

What is a heap?

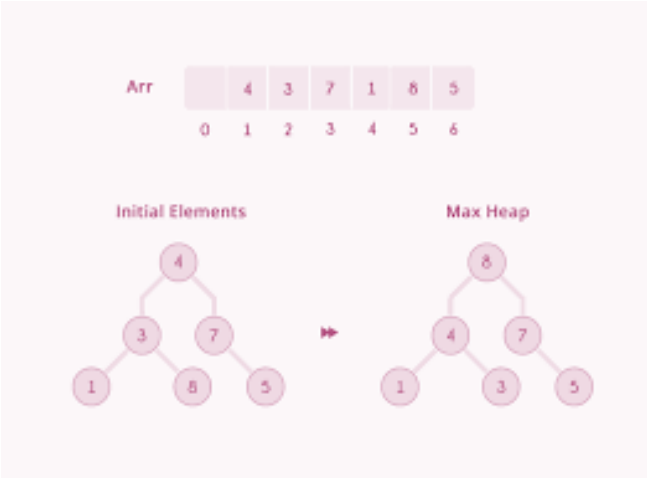
Heap is a special tree-based data structure, which satisfies the following special heap properties:

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

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

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

Heap sort is a comparison-based sorting technique based on Binary Heap data structure. It is similar to selection sort where we first find the minimum element and place the minimum element at the beginning. We repeat the same process for the remaining elements.



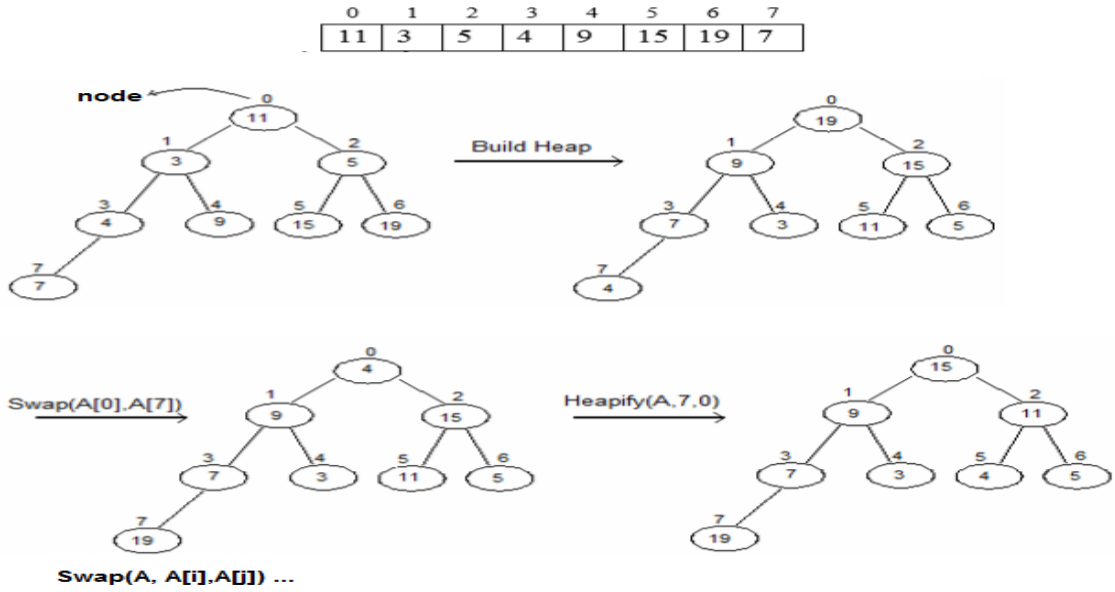
Pseudo code

```
Heapsort (A) {
   BuildHeap(A)
   for i <- length (A) downto 2 {
      exchange A [1] <-> A[i]
      heapsize <- heapsize -1
      Heapify(A, 1)
}
```

```
BuildHeap (A) {
   heapsize <- length (A)
   for i <- floor (length/2) downto 1
Heapify (A, i)
}

Heapify (A, i) {
   le <- left (i)
   ri <- right(i)
   if (le<=heapsize) and (A[le]>A[i])
```

```
      largest <- le
   else
      largest <- i
   if (ri<=heapsize) and (A[ri]>A[largest])
      largest <- ri
   if (largest != i) {
      exchange A[i] <-> A[largest]
      Heapify(A, largest)
   }
}
```

Example:



Result and Analysis

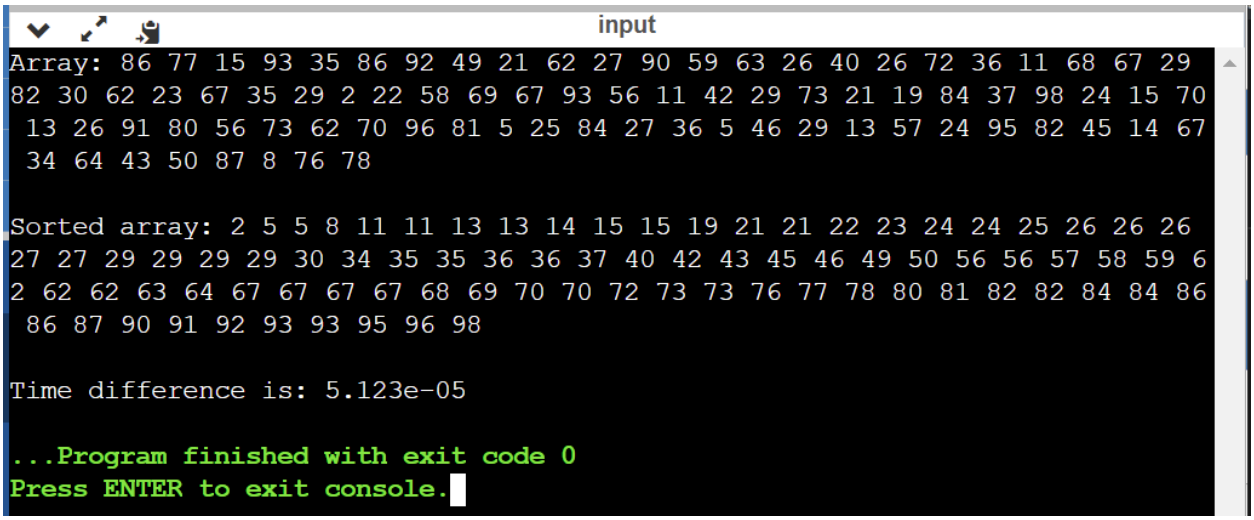
Heap sort has the best possible worst case running time complexity of O (n Log n).It doesn't need any extra storage and that makes it good for situations where array size is large. Heapify runs in time O (h) and there are at most $n=2^{h+1}-1$ nodes in an almost complete binary tree of height h, so Heapify runs in time O (ln n). Build-Heap calls Heapify n/2 times, so it takes O (n h) = O (n ln n).

Source Code:

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

void heapify(int arr[], int n, int root)
{
    int largest = root;
    int leftChild = 2 * root + 1;
    int rightChild = 2 * root + 2;

    if (leftChild < n && arr[leftChild] > arr[largest])
        largest = leftChild;

    if (rightChild < n && arr[rightChild] > arr[largest])
        largest = rightChild;

    if (largest != root)
    {
        swap(arr[root], arr[largest]);
        heapify(arr, n, largest);
    }
}

void heapSort(int arr[], int n)
{
    // Build heap
    for (int i = n / 2 - 1; i >= 0; i--)
        heapify(arr, n, i);

    for (int i = n - 1; i > 0; i--)
    {                          // ectract elements from heap
        swap(arr[0], arr[i]); // Move current root to end
        heapify(arr, i, 0);   // call max heapify on the reduced heap
    }
}

void printArray(int arr[], int size)
{
    for (int i = 0; i < size; i++)
        cout << arr[i] << " ";

    cout << endl;
```

```
}

int main()
{
    // int arr[] = {64, 34, 25, 12, 22, 11, 90};
    // int n = sizeof(arr)/sizeof(arr[0]);

    int n = rand() % 100;
    int arr[n];

    for (int i = 0; i < n; i++)
    {
        arr[i] = rand() % 100;
    }

    cout << "Array: ";
    printArray(arr, n);
    cout << endl;

    auto start = chrono::high_resolution_clock::now();

    // unsync the I/O of C and C++.
    ios_base::sync_with_stdio(false);

    heapSort(arr, n); // sort arr

    auto end = chrono::high_resolution_clock::now();

    cout << "Sorted array: ";
    printArray(arr, n);
    cout << endl;

    double time_taken = chrono::duration_cast<chrono::nanoseconds>(end -
start).count();
    time_taken *= 1e-9;

    cout << "Time difference is: " << time_taken << setprecision(6);

    return 0;
}
```

Output:


input
Array: 86 77 15 93 35 86 92 49 21 62 27 90 59 63 26 40 26 72 36 11 68 67 29
82 30 62 23 67 35 29 2 22 58 69 67 93 56 11 42 29 73 21 19 84 37 98 24 15 70
13 26 91 80 56 73 62 70 96 81 5 25 84 27 36 5 46 29 13 57 24 95 82 45 14 67
34 64 43 50 87 8 76 78
Sorted array: 2 5 5 8 11 11 13 13 14 15 15 19 21 21 22 23 24 24 25 26 26 26
27 27 29 29 29 29 30 34 35 35 36 36 37 40 42 43 45 46 49 50 56 56 57 58 59 6
2 62 62 63 64 67 67 67 67 68 69 70 70 72 73 73 76 77 78 80 81 82 82 84 84 86
86 87 90 91 92 93 93 95 96 98
Time difference is: 5.123e-05
...Program finished with exit code 0
Press ENTER to exit console.

Batch Analysis:

Source Code:

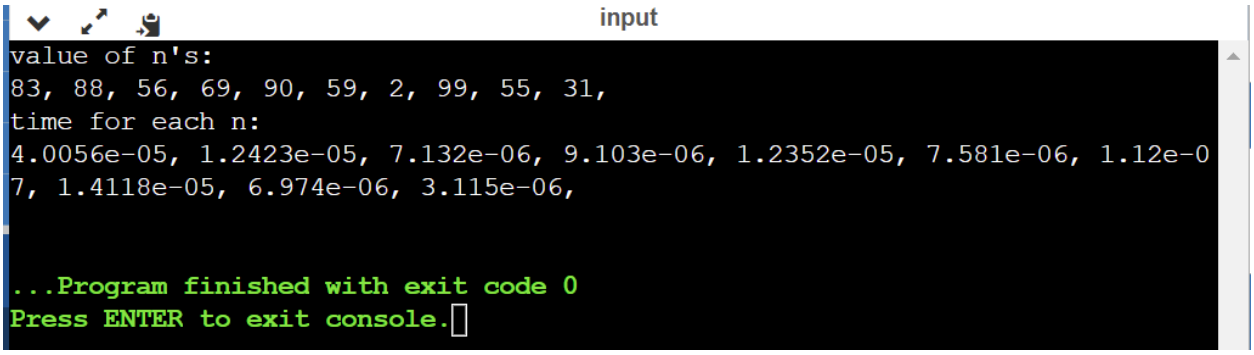
```
#include <bits/stdc++.h>
#include <chrono>
using namespace std;

void heapify(int arr[], int n, int root)
{
    int largest = root;
    int leftChild = 2 * root + 1;
    int rightChild = 2 * root + 2;

    if (leftChild < n && arr[leftChild] > arr[largest])
        largest = leftChild;

    if (rightChild < n && arr[rightChild] > arr[largest])
        largest = rightChild;

    if (largest != root)
    {
        swap(arr[root], arr[largest]);
        heapify(arr, n, largest);
    }
}

void heapSort(int arr[], int n)
{
    // Build heap
    for (int i = n / 2 - 1; i >= 0; i--)
```

```
        heapify(arr, n, i);

    for (int i = n - 1; i > 0; i--)
    {                          // ectract elements from heap
        swap(arr[0], arr[i]); // Move current root to end
        heapify(arr, i, 0);   // call max heapify on the reduced heap
    }
}

void printArray(double arr[], int size)
{
    for (int i = 0; i < size; i++)
        cout << arr[i] << ", ";
    cout << endl;
}

void printArray(int arr[], int size)
{
    for (int i = 0; i < size; i++)
        cout << arr[i] << ", ";
    cout << endl;
}

void printArray(float arr[], int size)
{
    for (int i = 0; i < size; i++)
        cout << arr[i] << ", ";
    cout << endl;
}
```

```
double sortApp(int n)
{
    int arr[n];

    for (int i = 0; i < n; i++)
    {
        arr[i] = rand() % 100;
    }

    auto start = chrono::high_resolution_clock::now();
    ios_base::sync_with_stdio(false);

    heapSort(arr, n); // sort arr

    auto end = chrono::high_resolution_clock::now();

    double time_taken = chrono::duration_cast<chrono::nanoseconds>(end -
start).count();
    time_taken *= 1e-9;

    return time_taken;
}

int main()
{
    double times[10];
    int ns[10];

    for (int x = 0; x < 10; x++)
    {
        int n = rand() % 100;
```

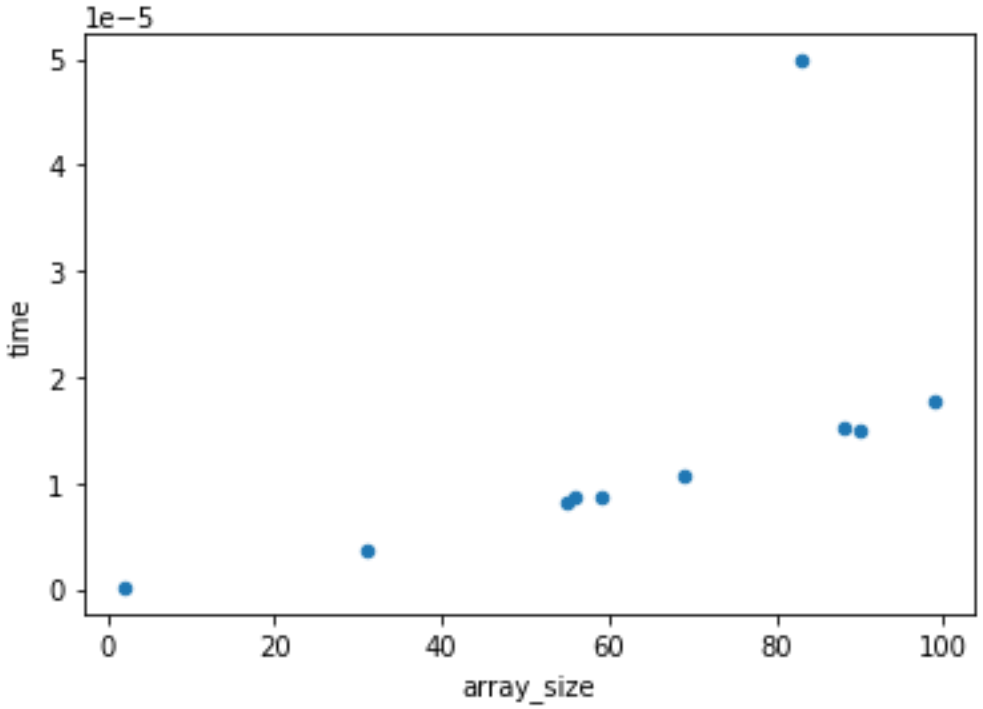
```
        ns[x] = n;
        times[x] = sortApp(n);
    }

    cout << "value of n's: " << endl;
    printArray(ns, 10);

    cout << "time for each n: " << endl;
    printArray(times, 10);
}
```

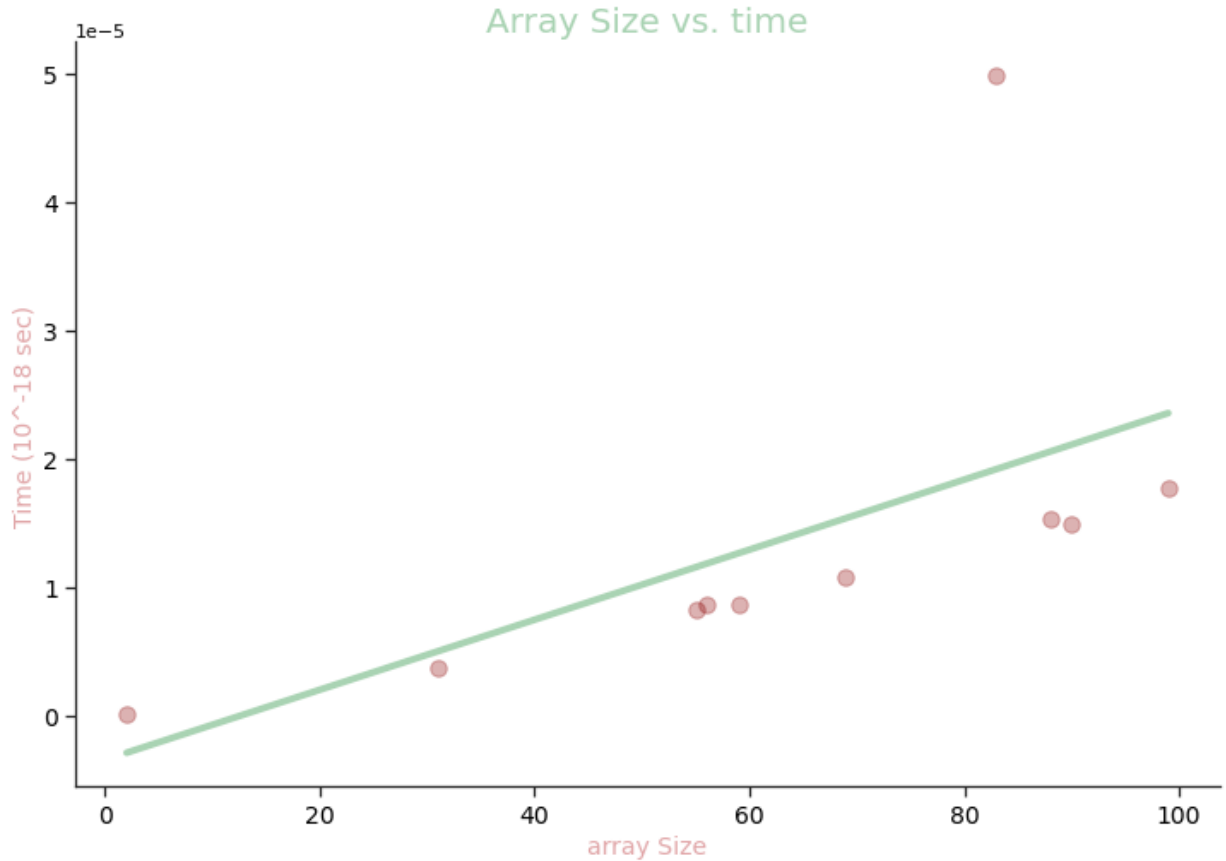
Output:



arraySize = [83, 88, 56, 69, 90, 59, 2, 99, 55, 31]

time = [4.9828e-05, 1.5332e-05, 8.655e-06, 1.0828e-05, 1.4977e-05, 8.714e-06, 1.48e-07, 1.7703e-05, 8.224e-06, 3.716e-06]


1e−5
5
4
3
2
1
0
time
0
20
40
60
80
100
array_size


Array Size vs. time
1e−5
5
4
3
2
1
0
Time (10^-18 sec)
0
20
40
60
80
100
array Size

Heap Sort

Viva Questions

1. What are the minimum and maximum numbers of elements in a heap of height *h*?

Ans.

2h

A heap of height h is complete up to the level at depth h−1and needs to have at least one node on level h.

Therefore the minimum total number of nodes must be at least $\sum h-1 i=02i+1=2h-1+1=2h.\sum i=0h-12i+1=2h-1+1=2h.$, and this tight since an heap with 2h nodes has height h.

2. Where in a heap might the smallest element reside?

Ans.

Smallest element will be at the last level of the max heap.

3. Is an array that is in reverse sorted order a heap?

Ans.

The correct version of this statement is "An array sorted in ascending order is can be treated **as min- heap**" and its complementry statement is "An array sorted in descending order can be treated as max heap"

4. Does heap sort uses extra space for storage?

Ans.

No, 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.

5. What is the effect of calling HEAPIFY *(A, i)* when the element A[*i*] is larger than its children?

Ans.

No effect. The comparisons are carried out, A[i]A[i] is found to be largest and the procedure just returns.

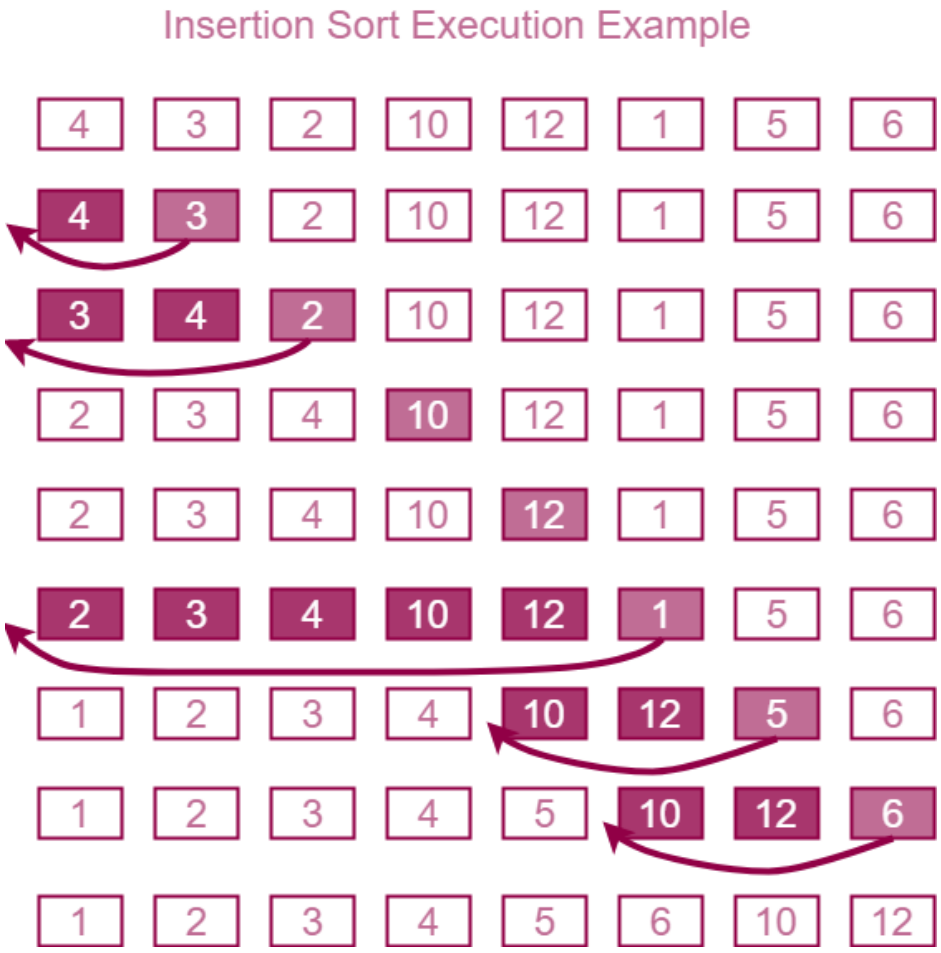
1.7 Insertion Sort:

Insertion sort is a simple sorting algorithm that works similar to the way you sort playing cards in your hands. The array is virtually split into a sorted and an unsorted part. Values from the unsorted part are picked and placed at the correct position in the sorted part.

Algorithm

To sort an array of size n in ascending order:
1: Iterate from arr[1] to arr[n] over the array.
2: Compare the current element (key) to its predecessor.
3: If the key element is smaller than its predecessor, compare it to the elements before. Move the greater elements one position up to make space for the swapped element.

Example:



Result and Analysis

Time Complexity: O(n^2)
Auxiliary Space: O(1)
Boundary Cases: Insertion sort takes maximum time to sort if elements are sorted in reverse order. And it takes minimum time (Order of n) when elements are already sorted.
Algorithmic Paradigm: Incremental Approach
Sorting In Place: Yes
Stable: Yes
Online: Yes
Uses: Insertion sort is used when number of elements is small. It can also be useful when input array is almost sorted, only few elements are misplaced in complete big array.

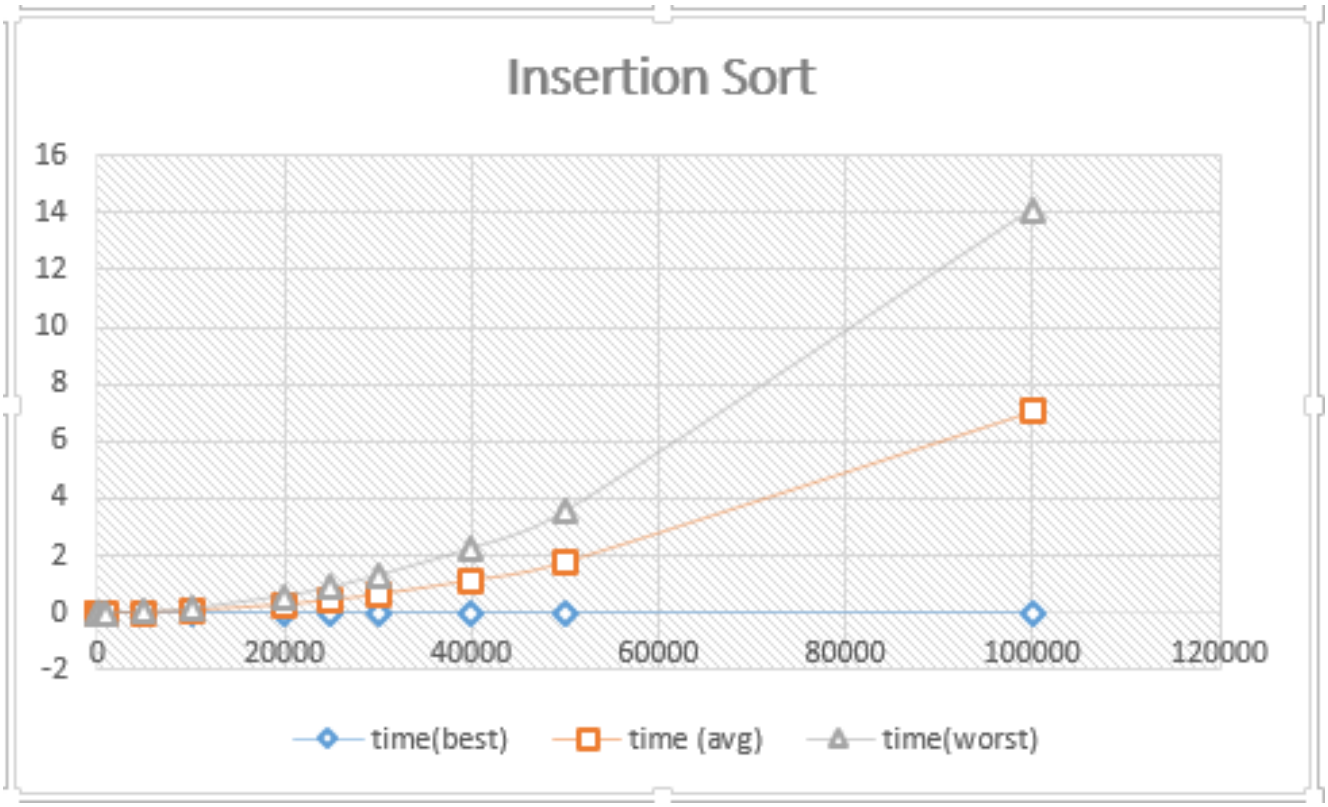
Source Code:

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

// place current element in right order as we move forward
void insertionSort(int arr[], int n)
{
    for (int i = 1; i < n; i++)
    {
        int key = arr[i];
        int j = i - 1;
        // compare to all predecessors
        while (j >= 0 && arr[j] > key)
        {
            arr[j + 1] = arr[j];
            j = j - 1;
        }
        arr[j + 1] = key;
    }
}

void printArray(int arr[], int size)
```

```
{
    for (int i = 0; i < size; i++)
        cout << arr[i] << " ";
    cout << endl;
}

int main()
{
    // int arr[] = {64, 34, 25, 12, 22, 11, 90};
    // int n = sizeof(arr)/sizeof(arr[0]);
    int n = rand() % 100;
    int arr[n];
    for (int i = 0; i < n; i++)
    {
        arr[i] = rand() % 100;
    }
    cout << "Array: ";
    printArray(arr, n);
    cout << endl;

    auto start = chrono::high_resolution_clock::now();
    // unsync the I/O of C and C++.
    ios_base::sync_with_stdio(false);

    insertionSort(arr, n);

    auto end = chrono::high_resolution_clock::now();

    cout << "Sorted array: ";
    printArray(arr, n);
    cout << endl;

    double time_taken = chrono::duration_cast<chrono::nanoseconds>(end -
start).count();
    time_taken *= 1e-9;

    cout << "Time difference is: " << time_taken << setprecision(6);
    return 0;
}
```

Output:

```
input
Array: 86 77 15 93 35 86 92 49 21 62 27 90 59 63 26 40 26 72 36 11 68 67 29
82 30 62 23 67 35 29 2 22 58 69 67 93 56 11 42 29 73 21 19 84 37 98 24 15 70
 13 26 91 80 56 73 62 70 96 81 5 25 84 27 36 5 46 29 13 57 24 95 82 45 14 67
 34 64 43 50 87 8 76 78

Sorted array: 2 5 5 8 11 11 13 13 14 15 15 19 21 21 22 23 24 24 25 26 26 26
27 27 29 29 29 29 30 34 35 35 36 36 37 40 42 43 45 46 49 50 56 56 57 58 59 6
2 62 62 63 64 67 67 67 67 68 69 70 70 72 73 73 76 77 78 80 81 82 82 84 84 86
 86 87 90 91 92 93 93 95 96 98

Time difference is: 5.155e-05

...Program finished with exit code 0
Press ENTER to exit console.
```



Batch Analysis:

Source Code:

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

// place current element in right order as we move forward
void insertionSort(int arr[], int n)
{
    for (int i = 1; i < n; i++)
    {
        int key = arr[i];
        int j = i - 1;
        // compare to all predecessors
        while (j >= 0 && arr[j] > key)
        {
            arr[j + 1] = arr[j];
            j = j - 1;
        }
        arr[j + 1] = key;
    }
}

void printArray(double arr[], int size)
{
    for (int i = 0; i < size; i++)
        cout << arr[i] << ", ";
    cout << endl;
}
```

```
void printArray(int arr[], int size)
{
    for (int i = 0; i < size; i++)
        cout << arr[i] << ", ";
    cout << endl;
}

void printArray(float arr[], int size)
{
    for (int i = 0; i < size; i++)
        cout << arr[i] << ", ";
    cout << endl;
}

double sortApp(int n)
{
    int arr[n];
    for (int i = 0; i < n; i++)
    {
        arr[i] = rand() % 100;
    }

    auto start = chrono::high_resolution_clock::now();
    ios_base::sync_with_stdio(false);

    insertionSort(arr, n);

    auto end = chrono::high_resolution_clock::now();
```

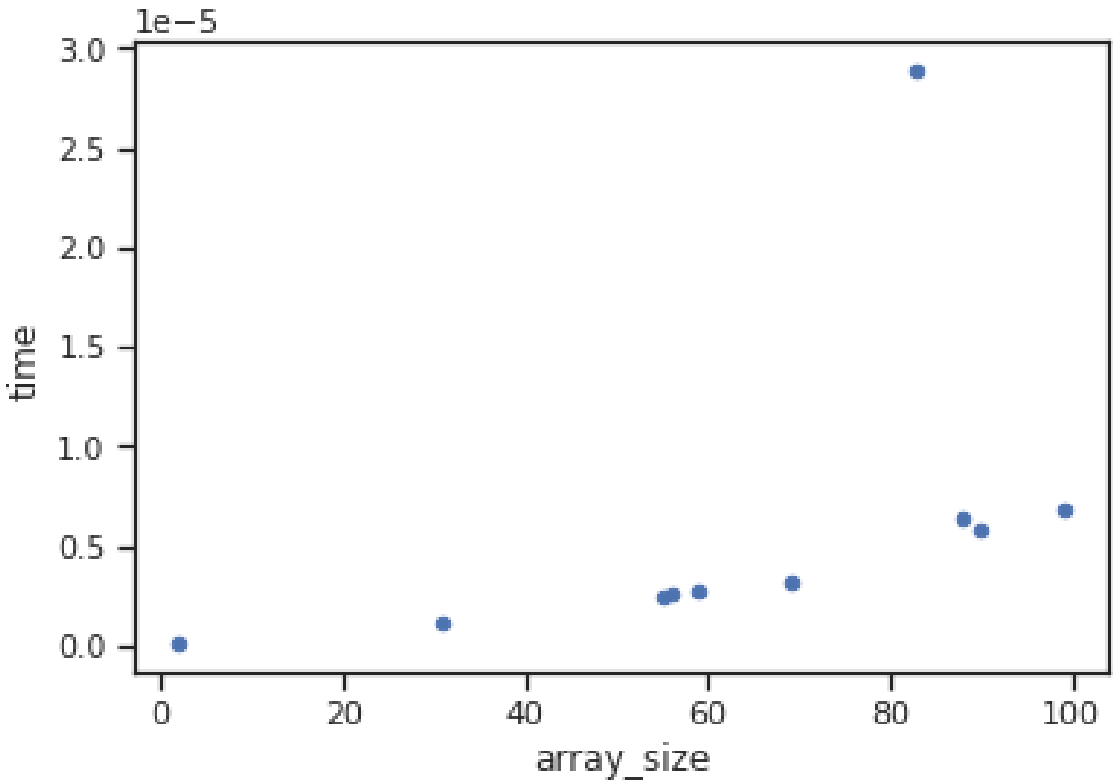
```
    double time_taken = chrono::duration_cast<chrono::nanoseconds>(end -
start).count();
    time_taken *= 1e-9;


    return time_taken;
}


int main()
{
    double times[10];
    int ns[10];


    for (int x = 0; x < 10; x++)
    {
        int n = rand() % 100;
        ns[x] = n;
        times[x] = sortApp(n);
    }


    cout << "value of n's: " << endl;
    printArray(ns, 10);


    cout << "time for each n: " << endl;
    printArray(times, 10);
}
```

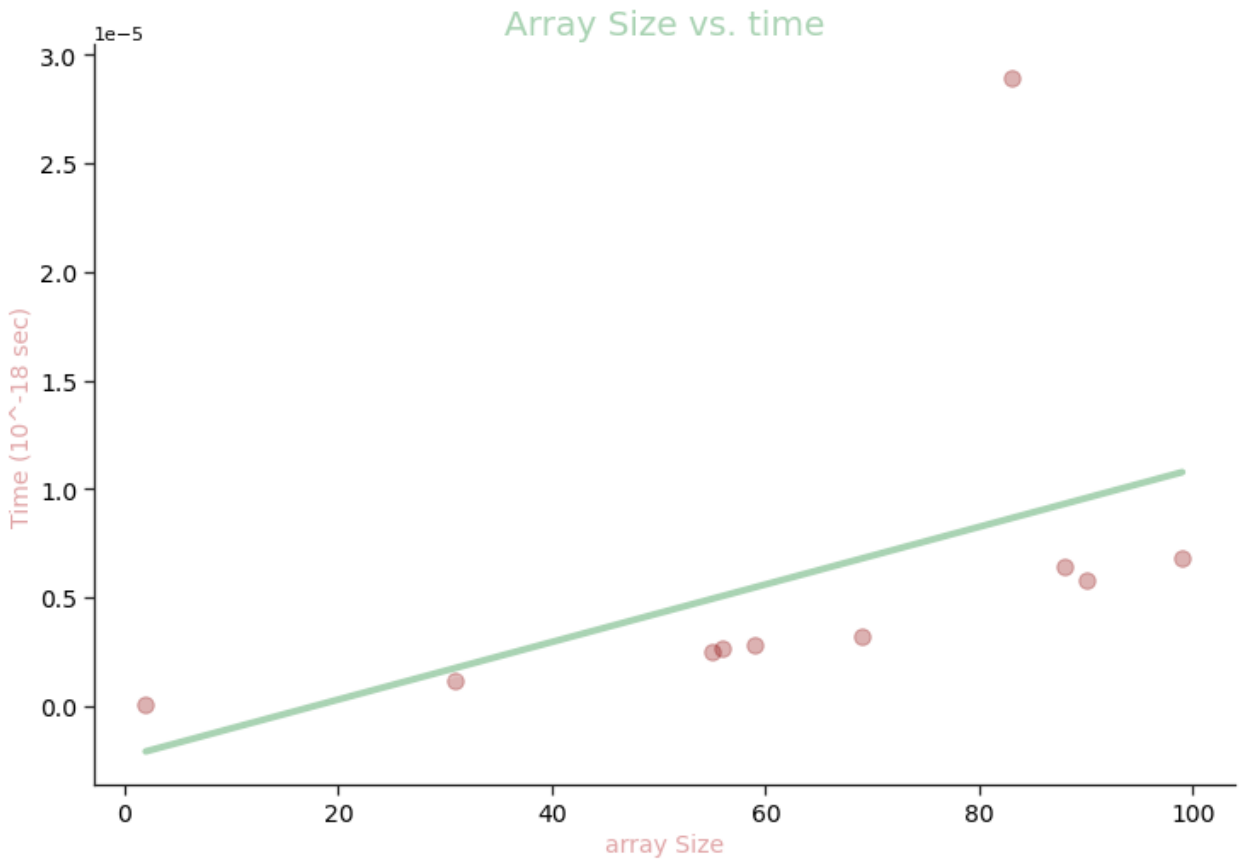
Output:

```
input
value of n's:
83, 88, 56, 69, 90, 59, 2, 99, 55, 31,
time for each n:
4.1828e-05, 7.931e-06, 3.27e-06, 3.856e-06, 7.223e-06, 3.512e-06, 9.7e-08, 8
.715e-06, 3.045e-06, 1.513e-06,


...Program finished with exit code 0
Press ENTER to exit console.
```

arraySize = [83, 88, 56, 69, 90, 59, 2, 99, 55, 31]

time = [2.8917e-05, 6.425e-06, 2.608e-06, 3.153e-06, 5.76e-06, 2.767e-06, 5.5e-08, 6.81e-06, 2.451e-06, 1.186e-06]





Insertion Sort

Viva Questions

1. Why is time complexity of insertion sort?

Ans.

Even though insertion sort is efficient, still, if we provide an already sorted array to the insertion sort algorithm, it will still execute the outer for loop, thereby requiring n steps to sort an already sorted array of n elements, which makes its best case time complexity a linear function of n.

Wherein for an unsorted array, it takes for an element to compare with all the other elements which mean every n element compared with all other n elements. Thus, making it for n x n, i.e., n2 comparisons. One can also take a look at other sorting algorithms such as *Merge sort, Quick Sort, Selection Sort*, etc. and understand their complexities.

Worst Case Time Complexity [Big-O]: $O(n^2)$

Best Case Time Complexity [Big-omega]: O(n)

Average Time Complexity [Big-theta]: $O(n^2)$

2. Is it possible to do insertion sort in place?

Ans.

Yes, It is. Insertion sort is used when number of elements is small. It can also be useful when input array is almost sorted, only few elements are misplaced in complete big array.

3. Is insertion sort the worst?

Ans.

nsertion sort has a fast best-case running time and is a good sorting algorithm to use if the input list is already mostly sorted. For larger or more unordered lists, an algorithm with a **faster worst** and average-case running time, such as mergesort, would be a better choice. Now iterate through this array just one time.The time

4. What is Binary Insertion Sort?

Ans.

We can use binary search to reduce the number of comparisons in normal insertion sort. Binary Insertion Sort uses binary search to find the proper location to insert the selected item at each iteration. In normal insertion, sorting takes O(i) (at ith iteration) in worst case. We can reduce it to O(logi) by using binary search. The algorithm, as a whole, still has a running worst case running time of O(n^2) because of the series of swaps required for each insertion.

5. Is insertion sort stable and in-place?

Ans.

Insertion sort is an online stable in-place sorting algorithm that builds the final sorted list one item at a time. It works on the principle of moving a element to its correct position in a sorted array. Advantages of Insertion Sort: Stable: it does not change the relative order of elements with equal keys.

6. Give any application of insertion sort?

Ans.

Insertion sort is used when number of elements is small. It can also be useful when input array is almost sorted, only few elements are misplaced in complete big array.

EXPERIMENT - 2

Algorithms Design and Analysis Lab

Aim

To implement Linear search and Binary search and analyse its time complexity.

Syeda Reeha Quasar
14114802719
4C7

EXPERIMENT – 2

Aim:

To implement Linear search and Binary search and analyse its time complexity.

Theory:

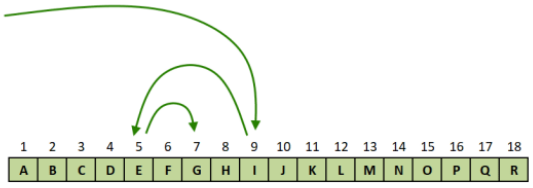
The search operation can be done in the following two ways:

Linear search:

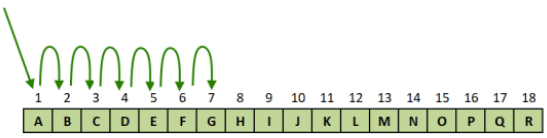
It's a sequential search is made over all items one by one. Every item is checked and if a match is found then that particular item is returned, otherwise the search continues till the end of the data collection.

Binary search:

This search algorithm works on the principle of divide and conquers. For this algorithm, the data collection should be in the sorted form. Binary search looks for a particular item by comparing the middle most item of the collection. If a match occurs, then the index of item is returned. If the middle item is greater than the item, then the item is searched in the sub-array to the left of the middle item. Otherwise, the item is searched for in the sub-array to the right of the middle item. This process continues on the sub-array as well until the size of the sub array reduces to zero.



Binary Search - Find 'G' in sorted list A-R



Linear Search - Find 'G' in sorted list A-R

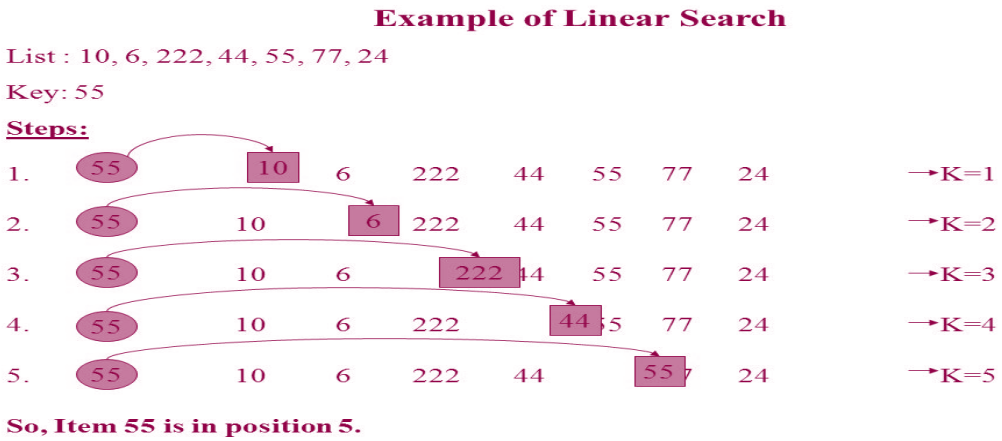
Pseudo code for linear search:

```
Procedure linear_search (list, value)
  for each item in the list
    if match item == value
      return the item's location
    end if
  end for
end procedure
```

Pseudo code for Binary search:

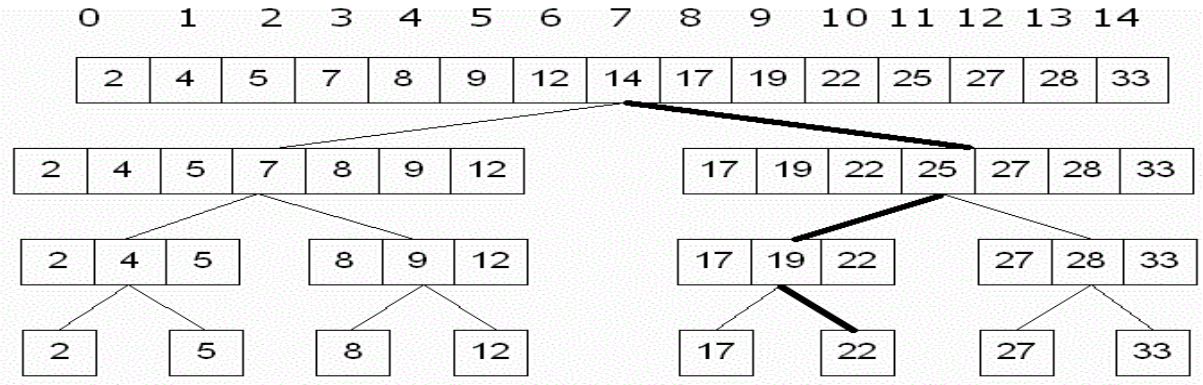
```
BinarySearch (A [0...N-1], value)
 {
    low = 0
    high = N - 1
    while (low <= high) {
       // invariants: value > A[i] for all i < low
                      value < A[i] for all i > high
       mid = (low + high) / 2
       if (A[mid] > value)
          high = mid - 1
       else if (A[mid] < value)
          low = mid + 1
       else
          return mid
    }
    return not_found // value would be inserted at index "low"
 }
```

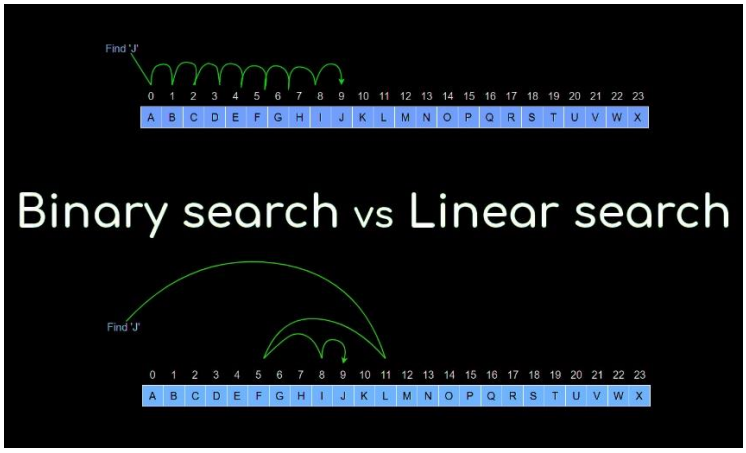
Sample Example:



Example of Binary Search:

A sorted array is taken as input. The mid value is found out recursively.





Result and Analysis

Input data needs to be sorted in Binary Search and not in Linear Search and it does the sequential access whereas Binary search access data randomly. So, time complexity of linear search is O (n) while Binary search has time complexity O (log n) as search is done to either half of the given list.

Base of comparison	Linear search	Binary search
Time complexity	O(N)	O (log2 *N*)
Best case time	O (1) first element	O (1) center element
Prerequisite of an array	No prerequisite	Array must be sorted in order
Input data	No need to be sorted	Need to be sorted
Access	Sequential	random

LINEAR SEARCH	BINARY SEARCH
An algorithm to find an element in a list by sequentially checking the elements of the list until finding the matching element	An algorithm that finds the position of a target value within a sorted array
Also called sequential search	Also called half-interval search and logarithmic search
Time complexity is O(N)	Time complexity is O(log2N)
Best case is to find the element in the first position	Best case is to find the element in the middle position
It is not required to sort the array before searching the element	It is necessary to sort the array before searching the element
Less efficient	More efficient
Less complex	More complex

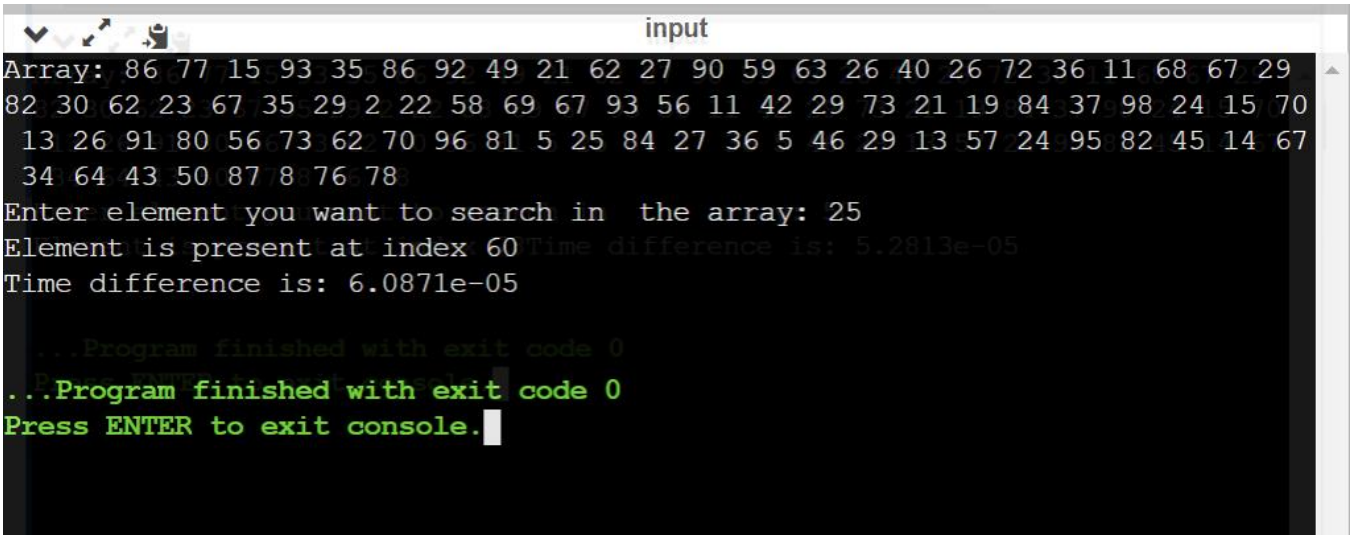
2.1 Linear search:

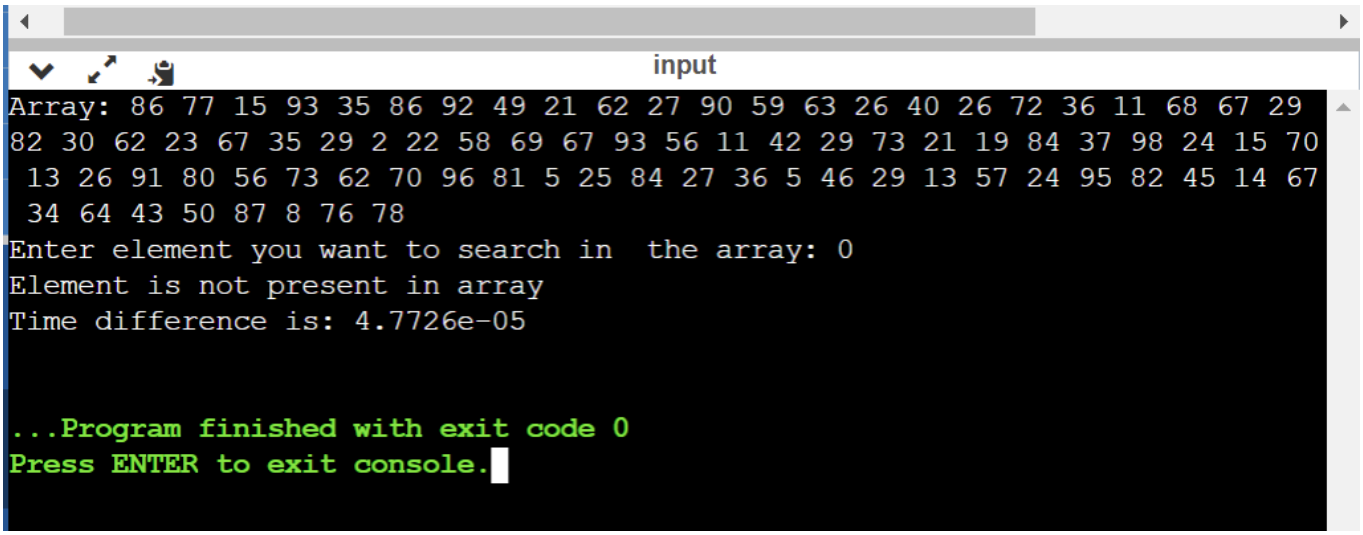
Source Code:

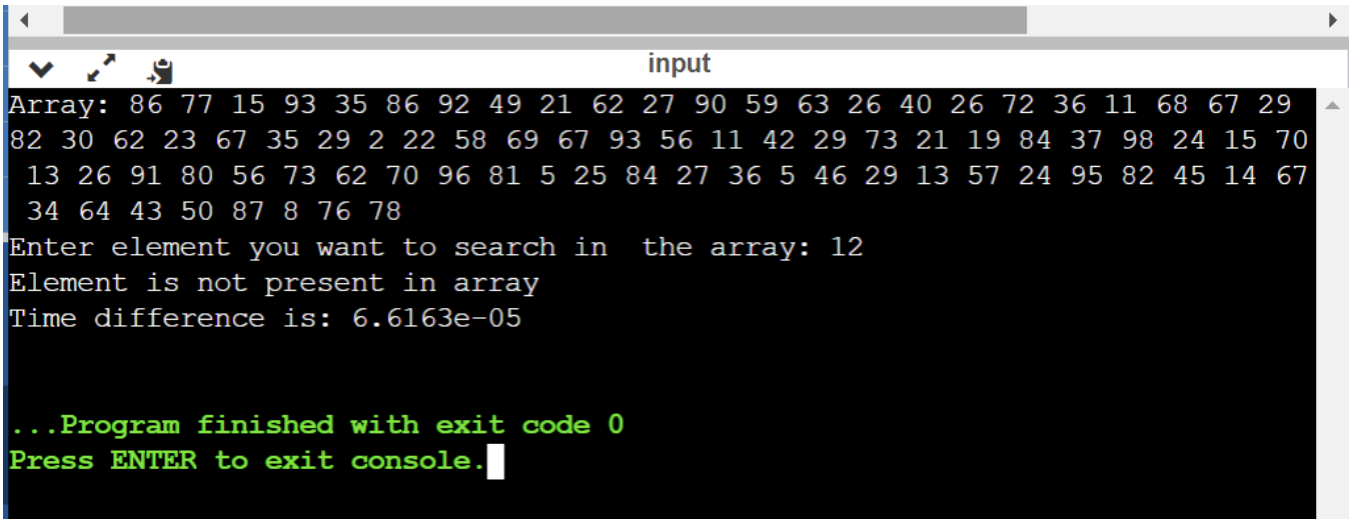
```
#include <bits/stdc++.h>
using namespace std;

int linearSearch(int arr[], int n, int x)
{
    for (int i = 0; i < n; i++)
        if (arr[i] == x)
            return i;
    return -1;
}

void printArray(int arr[], int size)
{
    int i;
    for (i = 0; i < size; i++)
        cout << arr[i] << " ";
    cout << endl;
}

int main()
{
    // int arr[] = { 2, 3, 4, 10, 40 };
    // int n = sizeof(arr) / sizeof(arr[0]);

    int n = rand() % 100;
    int arr[n];

    for (int i = 0; i < n; i++)
    {
        arr[i] = rand() % 100;
    }

    cout << "Array: ";
    printArray(arr, n);

    int x;

    cout << "Enter element you want to search in  the array: ";
```

```
    cin >> x;

    auto start = chrono::high_resolution_clock::now();
    // unsync the I/O of C and C++.
    ios_base::sync_with_stdio(false);

    int result = linearSearch(arr, n, x);

    auto end = chrono::high_resolution_clock::now();

    (result == -1)
        ? cout << "Element is not present in array"
        : cout << "Element is present at index " << result;

    double time_taken = chrono::duration_cast<chrono::nanoseconds>(end -
start).count();
    time_taken *= 1e-9;

    cout << "Time difference is: " << time_taken << setprecision(6) << endl;

    return 0;
}
```

Output:




input
Array: 86 77 15 93 35 86 92 49 21 62 27 90 59 63 26 40 26 72 36 11 68 67 29
82 30 62 23 67 35 29 2 22 58 69 67 93 56 11 42 29 73 21 19 84 37 98 24 15 70
13 26 91 80 56 73 62 70 96 81 5 25 84 27 36 5 46 29 13 57 24 95 82 45 14 67
34 64 43 50 87 8 76 78
Enter element you want to search in the array: 0
Element is not present in array
Time difference is: 4.7726e-05
...Program finished with exit code 0
Press ENTER to exit console.


input
Array: 86 77 15 93 35 86 92 49 21 62 27 90 59 63 26 40 26 72 36 11 68 67 29
82 30 62 23 67 35 29 2 22 58 69 67 93 56 11 42 29 73 21 19 84 37 98 24 15 70
13 26 91 80 56 73 62 70 96 81 5 25 84 27 36 5 46 29 13 57 24 95 82 45 14 67
34 64 43 50 87 8 76 78
Enter element you want to search in the array: 12
Element is not present in array
Time difference is: 6.6163e-05
...Program finished with exit code 0
Press ENTER to exit console.

Batch Analysis:

Source Code:

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

int linearSearch(int arr[], int n, int x)
{
    for (int i = 0; i < n; i++)
        if (arr[i] == x)
            return i;

    return -1;
}

void printArray(double arr[], int size)
{
    for (int i = 0; i < size; i++)
        cout << arr[i] << ", ";

    cout << endl;
}

void printArray(int arr[], int size)
{
    for (int i = 0; i < size; i++)
        cout << arr[i] << ", ";

    cout << endl;
}

void printArray(float arr[], int size)
{
    for (int i = 0; i < size; i++)
        cout << arr[i] << ", ";

    cout << endl;
}

double search(int n)
{
    int arr[n];
```

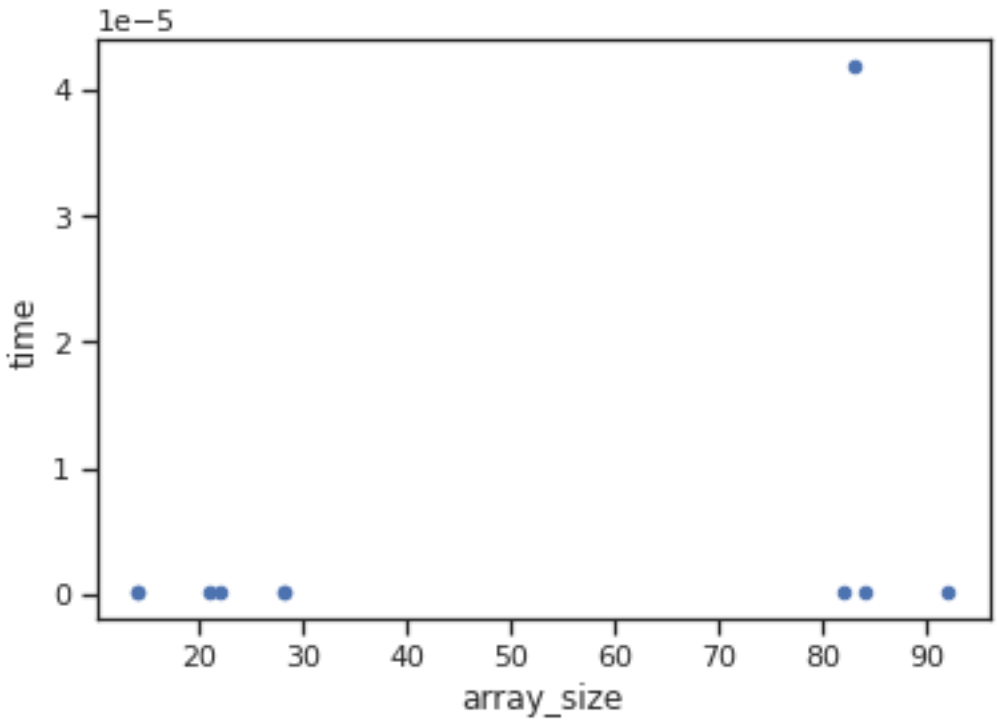
```
    for (int i = 0; i < n; i++)
    {
        arr[i] = rand() % 100;
    }
    sort(arr, arr + n);

    int x = rand() % 100;

    auto start = chrono::high_resolution_clock::now();
    ios_base::sync_with_stdio(false);

    linearSearch(arr, n, x);

    auto end = chrono::high_resolution_clock::now();

    double time_taken = chrono::duration_cast<chrono::nanoseconds>(end -
start).count();
    time_taken *= 1e-9;

    return time_taken;
}

int main()
{
    double times[10];
    int ns[10];

    for (int x = 0; x < 10; x++)
    {
        int n = rand() % 100;
        ns[x] = n;
        times[x] = search(n);
    }

    cout << "value of n's: " << endl;
    printArray(ns, 10);

    cout << "time for each n: " << endl;
    printArray(times, 10);
}
```

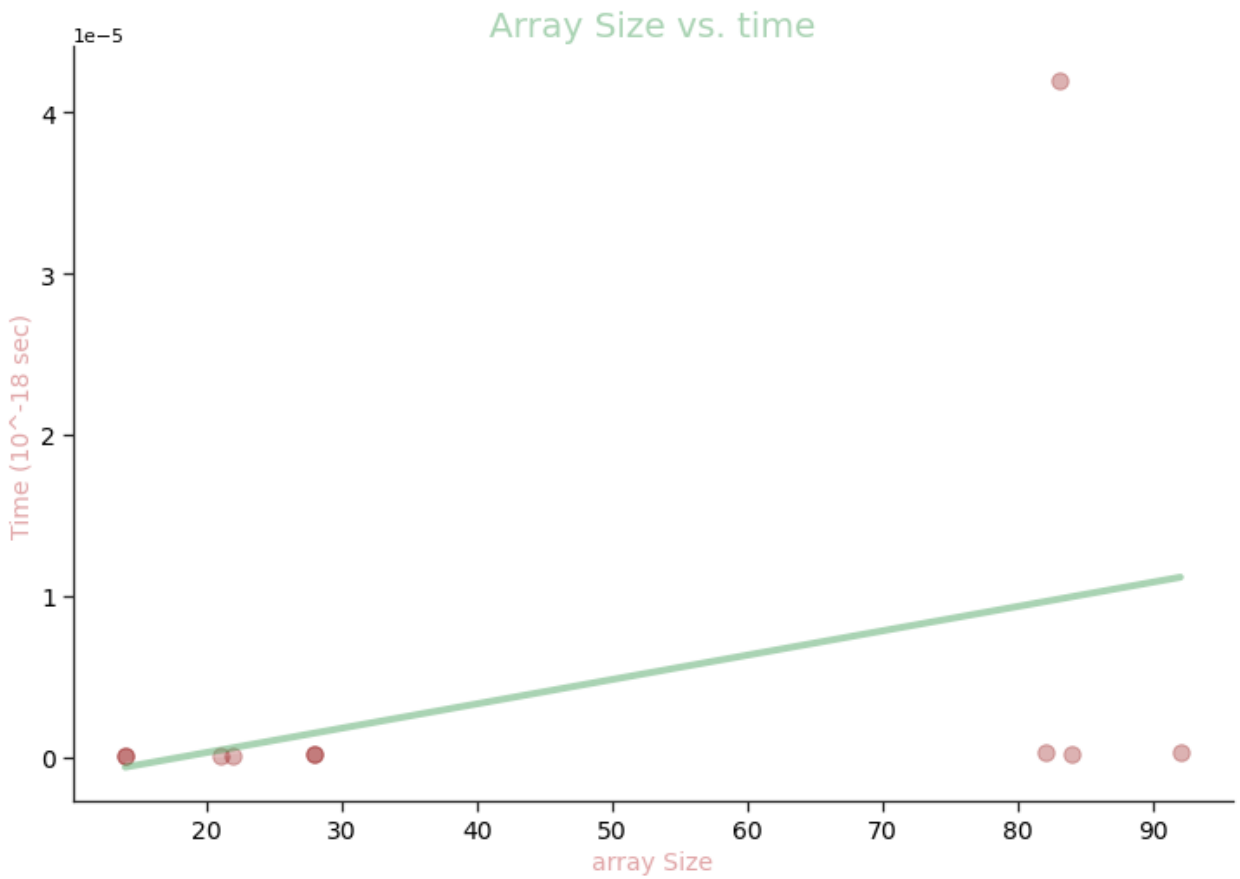
Output:

input

```
value of n's:
83, 84, 14, 21, 28, 22, 82, 92, 14, 28,
time for each n:
3.1241e-05, 1.35e-07, 1.01e-07, 9.3e-08, 1.43e-07, 1.14e-07, 2.44e-07, 2.25e
-07, 5.8e-08, 1.33e-07,


...Program finished with exit code 0
Press ENTER to exit console.
```

```
arraySize = [83, 84, 14, 21, 28, 22, 82, 92, 14, 28]

time = [4.1967e-05, 1.51e-07, 1.03e-07, 1.02e-07, 1.65e-07, 1.46e-07,
2.93e-07, 2.79e-07, 1.04e-07, 1.68e-07]
```





Linear Search

2.2 Binary search:

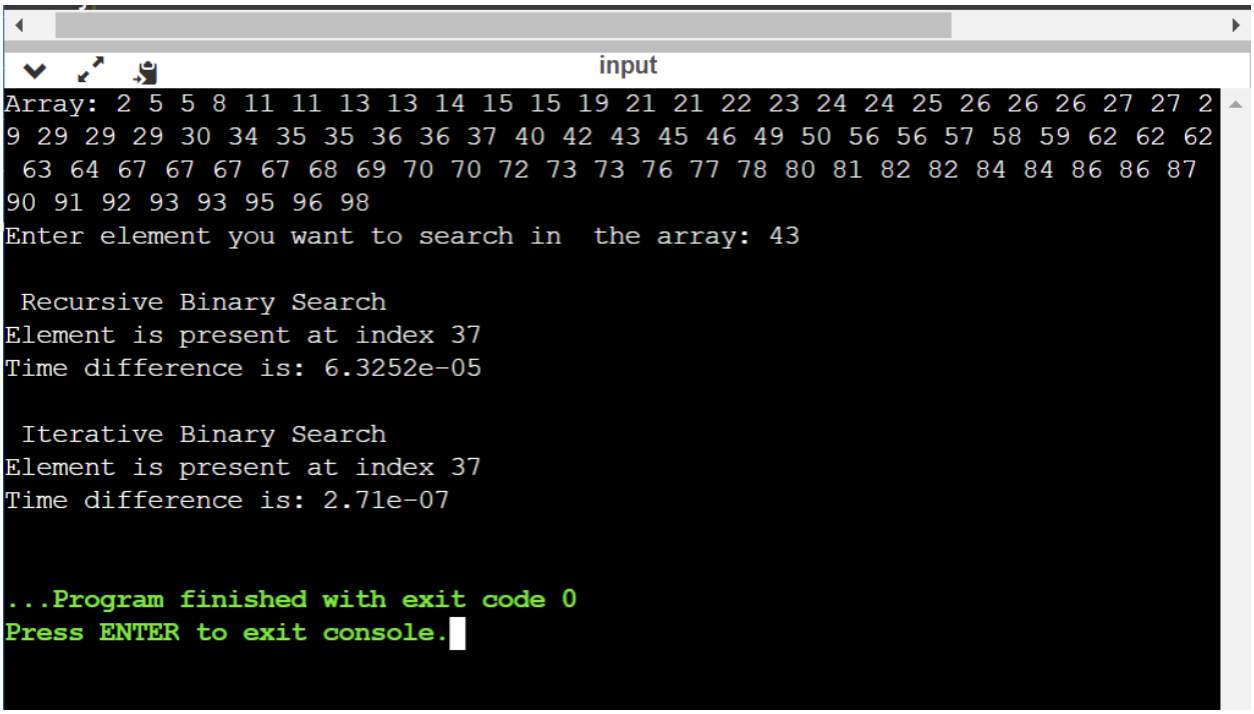
Source Code:

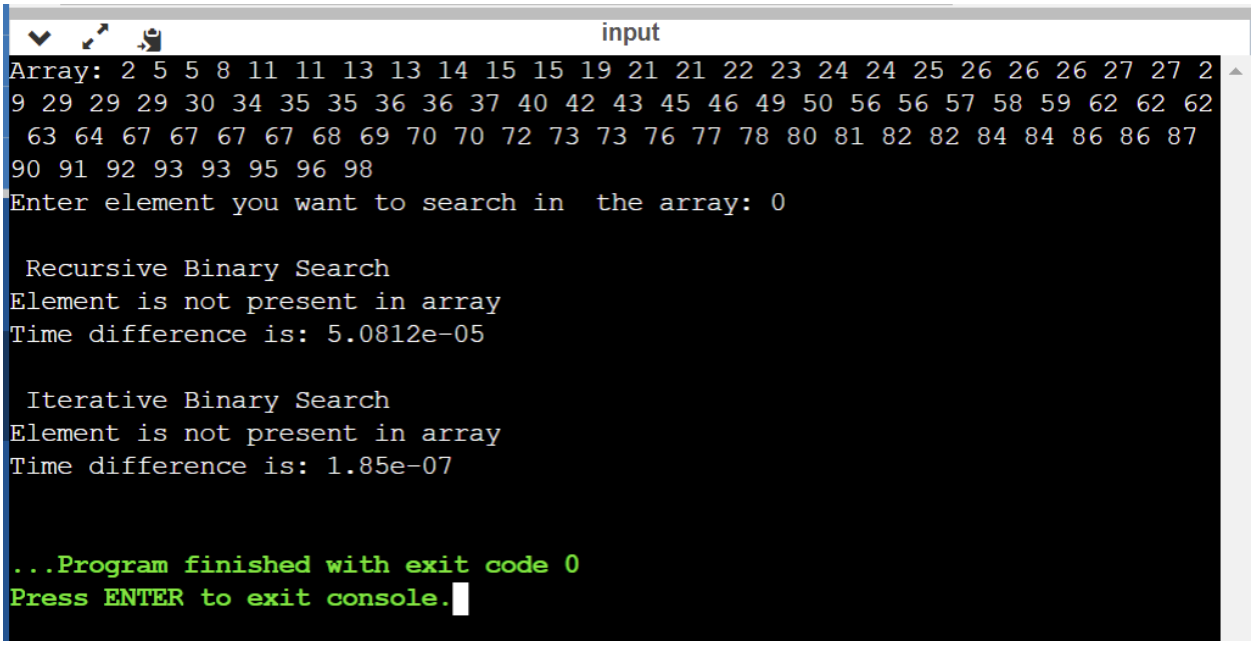
```
#include <bits/stdc++.h>
using namespace std;

int binarySearch(int arr[], int l, int r, int x)
{
    if (r >= l)
    {
        int mid = l + (r - l) / 2;

        if (arr[mid] == x)
            return mid;

        else if (arr[mid] > x)
            return binarySearch(arr, l, mid - 1, x);

        return binarySearch(arr, mid + 1, r, x);
    }
    return -1;
}

int binarySearchIter(int arr[], int l, int r, int x)
{
    while (l <= r)
    {
        int m = l + (r - l) / 2;

        if (arr[m] == x)
            return m;

        if (arr[m] < x)
            l = m + 1;

        else
            r = m - 1;
    }
    return -1;
}
```

```
void printArray(int arr[], int size)
{
    int i;

    for (i = 0; i < size; i++)
        cout << arr[i] << " ";

    cout << endl;
}

int main()
{
    // int arr[] = { 2, 3, 4, 10, 40 };
    // int n = sizeof(arr) / sizeof(arr[0]);

    int n = rand() % 100;
    int arr[n];

    for (int i = 0; i < n; i++)
    {
        arr[i] = rand() % 100;
    }

    sort(arr, arr + n);

    cout << "Array: ";
    printArray(arr, n);

    int x;

    cout << "Enter element you want to search in  the array: ";
    cin >> x;

    cout << "\n Recursive Binary Search" << endl;

    auto start = chrono::high_resolution_clock::now();
    // unsync the I/O of C and C++.
    ios_base::sync_with_stdio(false);

    int result = binarySearch(arr, 0, n, x);

    auto end = chrono::high_resolution_clock::now();

    (result == -1)
        ? cout << "Element is not present in array" << endl
```

```
        : cout << "Element is present at index " << result << endl;

    double time_taken = chrono::duration_cast<chrono::nanoseconds>(end -
start).count();
    time_taken *= 1e-9;

    cout << "Time difference is: " << time_taken << setprecision(6) << endl;

    cout << "\n Iterative Binary Search" << endl;

    start = chrono::high_resolution_clock::now();

    int resultIter = binarySearchIter(arr, 0, n, x);

    end = chrono::high_resolution_clock::now();

    (resultIter == -1)
        ? cout << "Element is not present in array" << endl
        : cout << "Element is present at index " << resultIter << endl;

    time_taken = chrono::duration_cast<chrono::nanoseconds>(end - start).count();
    time_taken *= 1e-9;

    cout << "Time difference is: " << time_taken << setprecision(6) << endl;

    return 0;
}
```

Output:


input
Array: 2 5 5 8 11 11 13 13 14 15 15 19 21 21 22 23 24 24 25 26 26 26 27 27 2
9 29 29 29 30 34 35 35 36 36 37 40 42 43 45 46 49 50 56 56 57 58 59 62 62 62
63 64 67 67 67 67 68 69 70 70 72 73 73 76 77 78 80 81 82 82 84 84 86 86 87
90 91 92 93 93 95 96 98
Enter element you want to search in the array: 43
Recursive Binary Search
Element is present at index 37
Time difference is: 6.3252e-05
Iterative Binary Search
Element is present at index 37
Time difference is: 2.71e-07
...Program finished with exit code 0
Press ENTER to exit console.


input
Array: 2 5 5 8 11 11 13 13 14 15 15 19 21 21 22 23 24 24 25 26 26 26 27 27 2
9 29 29 29 30 34 35 35 36 36 37 40 42 43 45 46 49 50 56 56 57 58 59 62 62 62
63 64 67 67 67 67 68 69 70 70 72 73 73 76 77 78 80 81 82 82 84 84 86 86 87
90 91 92 93 93 95 96 98
Enter element you want to search in the array: 0
Recursive Binary Search
Element is not present in array
Time difference is: 5.0812e-05
Iterative Binary Search
Element is not present in array
Time difference is: 1.85e-07
...Program finished with exit code 0
Press ENTER to exit console.

Batch Analysis:

Source Code:

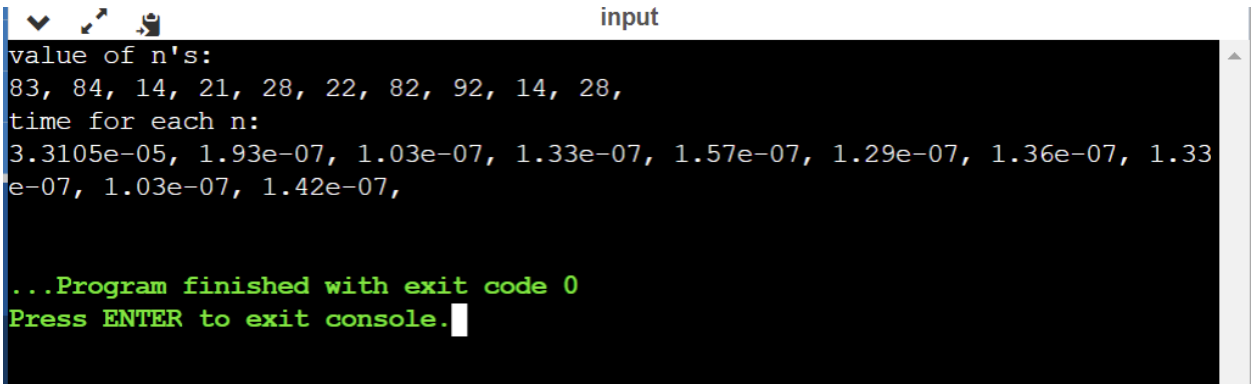
```
#include <bits/stdc++.h>
using namespace std;

int binarySearch(int arr[], int l, int r, int x)
{
    if (r >= l)
    {
        int mid = l + (r - l) / 2;

        if (arr[mid] == x)
            return mid;

        else if (arr[mid] > x)
            return binarySearch(arr, l, mid - 1, x);

        return binarySearch(arr, mid + 1, r, x);
    }
    return -1;
}

int binarySearchIter(int arr[], int l, int r, int x)
{
    while (l <= r)
    {
        int m = l + (r - l) / 2;
        if (arr[m] == x)
```

```
            return m;

        if (arr[m] < x)
            l = m + 1;

        else
            r = m - 1;
    }
    return -1;
}

void printArray(double arr[], int size)
{
    for (int i = 0; i < size; i++)
        cout << arr[i] << ", ";

    cout << endl;
}

void printArray(int arr[], int size)
{
    for (int i = 0; i < size; i++)
        cout << arr[i] << ", ";

    cout << endl;
}

void printArray(float arr[], int size)
{
    for (int i = 0; i < size; i++)
```

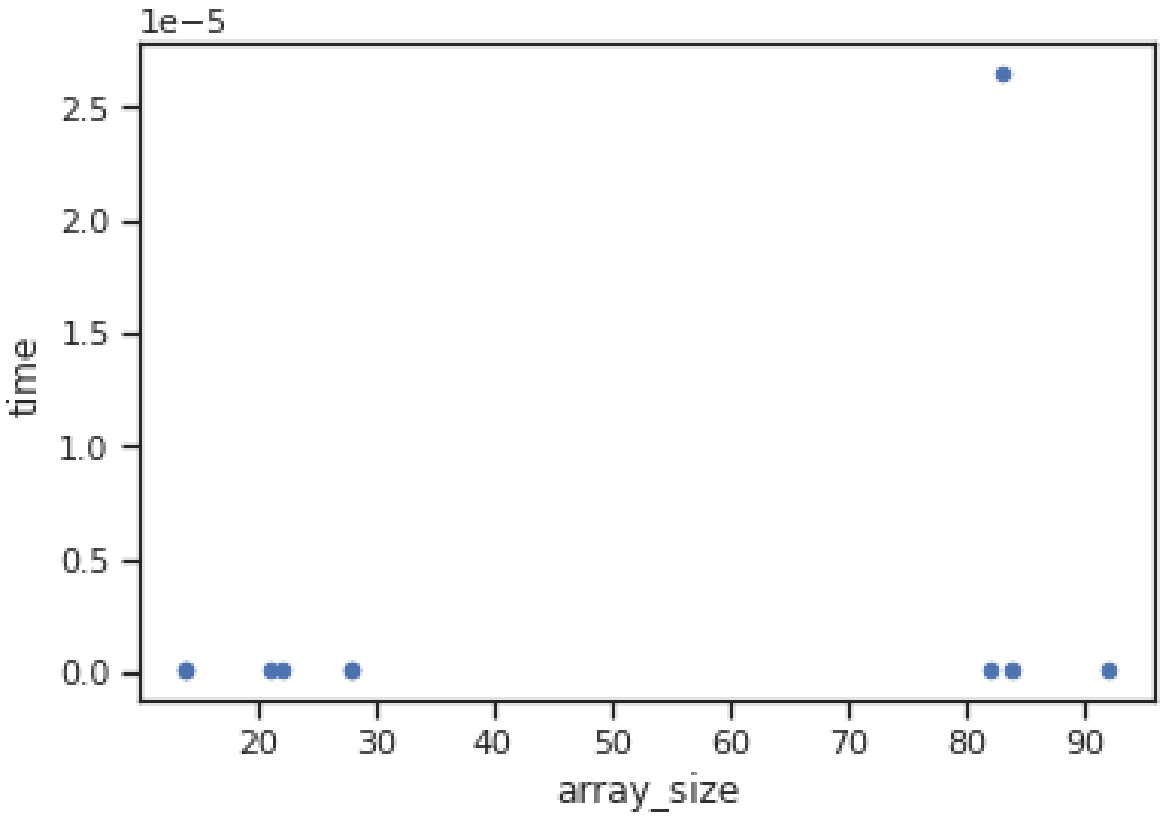
```
        cout << arr[i] << ", ";

    cout << endl;
}

double search(int n)
{
    int arr[n];
    for (int i = 0; i < n; i++)
    {
        arr[i] = rand() % 100;
    }
    sort(arr, arr + n);

    int x = rand() % 100;

    auto start = chrono::high_resolution_clock::now();
    ios_base::sync_with_stdio(false);

    binarySearch(arr, 0, n, x);

    auto end = chrono::high_resolution_clock::now();

    double time_taken = chrono::duration_cast<chrono::nanoseconds>(end -
start).count();
    time_taken *= 1e-9;

    return time_taken;
}

int main()
```

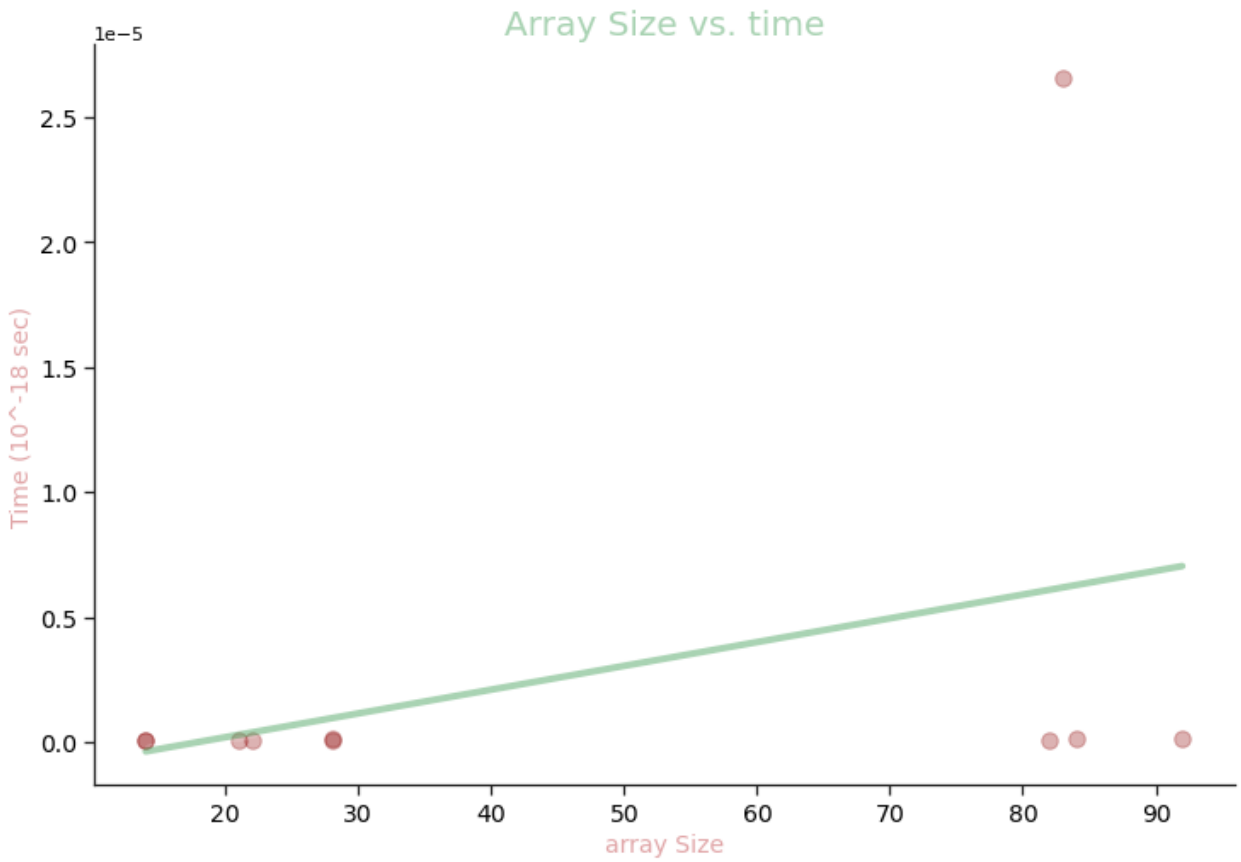
```
{
    double times[10];
    int ns[10];

    for (int x = 0; x < 10; x++)
    {
        int n = rand() % 100;
        ns[x] = n;
        times[x] = search(n);
    }

    cout << "value of n's: " << endl;
    printArray(ns, 10);

    cout << "time for each n: " << endl;
    printArray(times, 10);
}
```

Output:



```
arraySize = [83, 84, 14, 21, 28, 22, 82, 92, 14, 28]

time = [2.6536e-05, 1.6e-07, 7.9e-08, 9.3e-08, 1.31e-07, 8.7e-08, 1e-07,
1.15e-07, 8.9e-08, 1.06e-07]
```





Binary Search

Viva Questions

1. The sequential search, also known as _____?

Ans.

Linear Search

One of the most straightforward and elementary searches is the sequential search, also known as **a linear search**.

2. What is the primary requirement for implementation of binary search in an array?

Ans.

The one pre-requisite of binary search is that an array should be in sorted order, whereas the linear search works on both sorted and unsorted array. The binary search algorithm is based on the divide and conquer technique, which means that it will divide the array recursively.

3. Is binary search applicable on array and linked list both?

Ans.

Yes, Binary search is possible on the linked list if the list is ordered and you know the count of elements in list. But While sorting the list, you can access a single element at a time through a pointer to that node i.e. either a previous node or next node.

4. What is the principle of working of Binary search?

Ans.

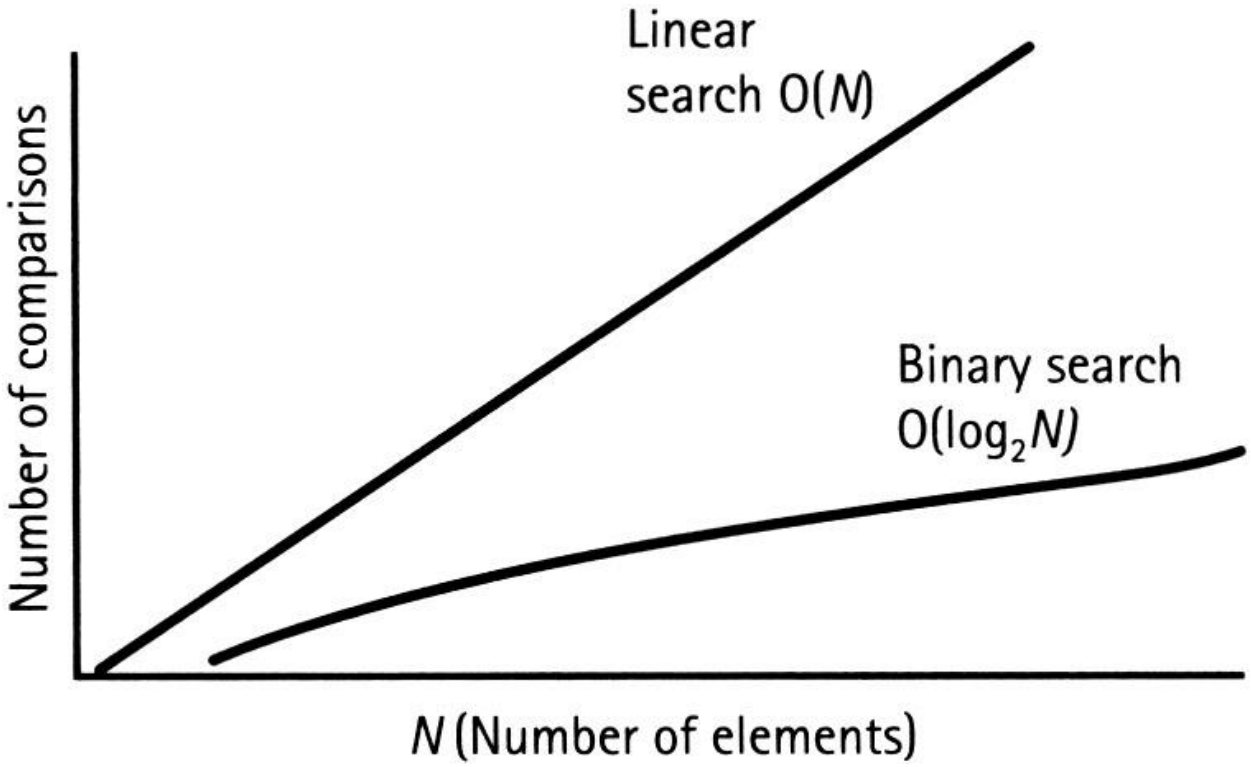
Divide and Conquer

Binary search is a fast search algorithm with run-time complexity of O(log n). This search algorithm works on the principle of **divide and conquer**. For this algorithm to work properly, the data collection should be in the sorted form.

5. Which searching algorithm is efficient one?

Ans.

Binary search is a more efficient search algorithm which relies on the elements in the list being sorted. We apply the same search process to progressively smaller sub-lists of the original list, starting with the whole list and approximately halving the search area every time.



6. Under what circumstances, binary search cannot be applied to a list of elements?

Ans.

In case the list of elements is not sorted, there's no way to use binary search because the median value of the list can be anywhere and when the list is split into two parts, the element that you were searching for could be cut off.

The main problem that binary search takes **O(n) time in Linked List** due to fact that in linked list we are not able to do indexing which led traversing of each element in Linked list take O(n) time. In this paper a method is implemented through which binary search can be done with time complexity of O(log2n).

EXPERIMENT - 3

Algorithms Design and Analysis Lab

Aim

To implement divide and conquer techniques and analyse its time complexity.

Syeda Reeha Quasar
14114802719
4C7

Experiment – 3

Aim:

To implement divide and conquer techniques and analyse its time complexity.

Theory:

A **divide and conquer algorithm** is a strategy of solving a large problem by

1. breaking the problem into smaller sub-problems
2. solving the sub-problems, and
3. combining them to get the desired output.

Here are the steps involved to perform divide and conquer techniques:

1. **Divide**: Divide the given problem into sub-problems using recursion.
2. **Conquer**: Solve the smaller sub-problems recursively. If the subproblem is small enough, then solve it directly.
3. **Combine:** Combine the solutions of the sub-problems that are part of the recursive process to solve the actual problem.

Time Complexity

The complexity of the divide and conquer algorithm is calculated using the master theorem.

```
T(n) = aT(n/b) + f(n),

where,

n = size of input

a = number of subproblems in the recursion

n/b = size of each subproblem. All subproblems are assumed to have the same size.

f(n) = cost of the work done outside the recursive call, which includes the cost of dividing the problem and cost of merging the solutions
```

Let us take an example to find the time complexity of a recursive problem.

For a merge sort, the equation can be written as:

```
T(n) = aT(n/b) + f(n)

     = 2T(n/2) + O(n)

Where,

a = 2 (each time, a problem is divided into 2 subproblems)

n/b = n/2 (size of each sub problem is half of the input)

f(n) = time taken to divide the problem and merging the subproblems

T(n/2) = O(n log n) (To understand this, please refer to the master theorem.)


Now, T(n) = 2T(n log n) + O(n)

          ≈ O(n log n)
```

Advantages of Divide and Conquer Algorithm

- The complexity for the multiplication of two matrices using the naive method is $O(n^3)$, whereas using the divide and conquer approach (i.e. Strassen's matrix multiplication) is $O(n^{2.8074})$. This approach also simplifies other problems, such as the Tower of Hanoi.
- This approach is suitable for multiprocessing systems.
- It makes efficient use of memory caches.

3.1 Merge Sort:

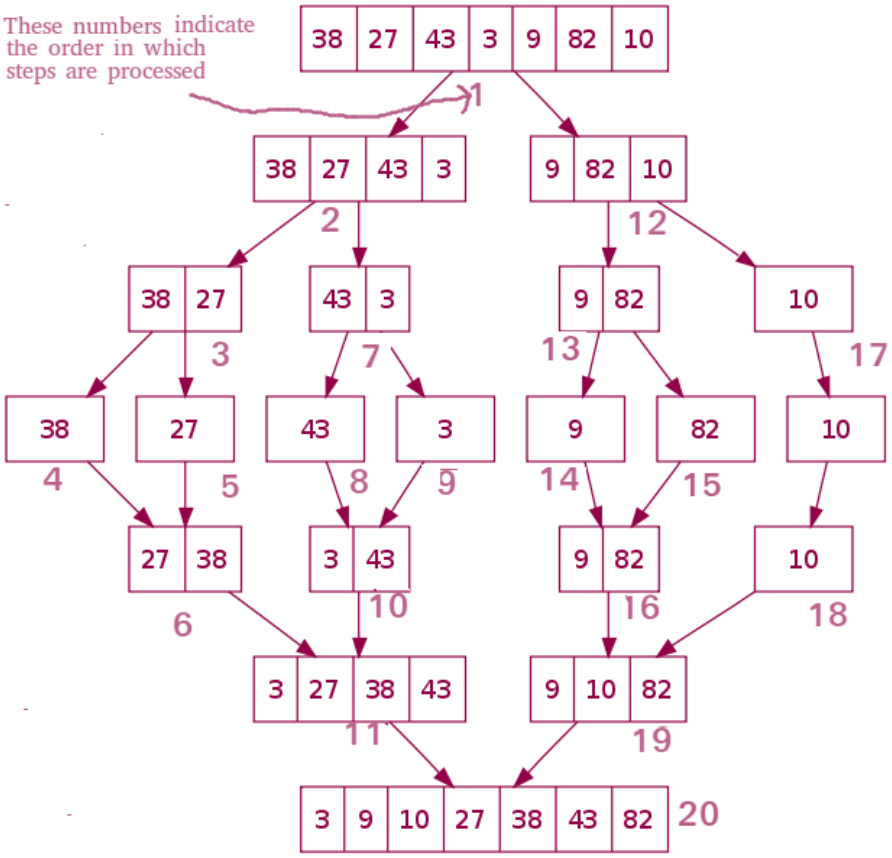
Conceptually, a merge sort works as follows:

1. Divide the unsorted list into *n* subsists, each containing 1 element (a list of 1 element is considered sorted).
2. Repeatedly merge subsists to produce new sorted subsists until there is only 1 subsist remaining. This will be the sorted list.

Pseudo code

```
MergeSort(arr[], l,  r)
If r > l
     1. Find the middle point to divide the array into two halves:
             middle m = l+ (r-l)/2
     2. Call mergeSort for first half:
             Call mergeSort(arr, l, m)
     3. Call mergeSort for second half:
             Call mergeSort(arr, m+1, r)
     4. Merge the two halves sorted in step 2 and 3:
             Call merge(arr, l, m, r)
```

Example:



Result and Analysis

The unordered list of elements gets sorted but additional space is used for merging.
The list of size N is divided into a max of logN parts, and the merging of all sub lists into a single list takes O(N) time. Time complexity of Merge Sort is O (nlogn) in all 3 cases (worst, average and best) as merge sort always divides the array in two halves and take linear time to merge two halves. The recurrence equation used is:T(n) = 2T(n/2) + O(n)

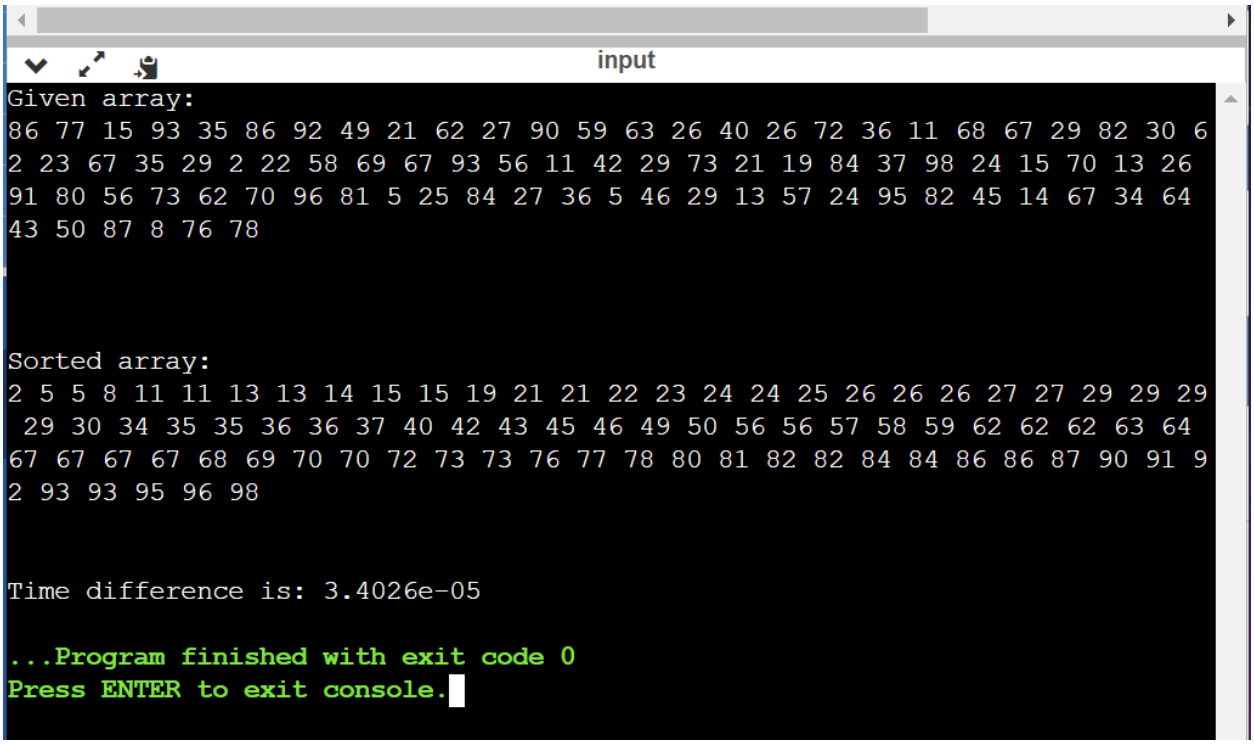
Source Code:

```
#include <bits/stdc++.h>
#include <chrono>

using namespace std;

// merging 2 arrays
void merge(int arr[], int const left, int const midIdx, int const right) {
    int temp[right - left + 1]; // sorted Arr

    int start = left, mid =  midIdx + 1, tempIdx = 0;

    // sorted merging of left and right arrays
    while (start <= midIdx && mid <= right) {
        if (arr[start] <= arr[mid]) {
            temp[tempIdx] = arr[start];
            tempIdx++;
            start++;
        }
        else {
            temp[tempIdx] = arr[mid];
            tempIdx++;
            mid++;
        }
    }

    // check and add for remaining element in left arr
    while (start <= midIdx) {
        temp[tempIdx] = arr[start];
        tempIdx++;
```

```
        start++;
    }

    // check and add for remaining element in right arr
    while (mid <= right) {
        temp[tempIdx] = arr[mid];
        tempIdx++;
        mid++;
    }

    for (int i = left; i <= right; i++) {
        arr[i] = temp[i - left];
    }
}

void mergeSort(int array[], int const begin, int const end) {
    if (begin >= end) {
        return; // Returns recursively
    }
    auto mid = begin + (end - begin) / 2;
    mergeSort(array, begin, mid);
    mergeSort(array, mid + 1, end);
    merge(array, begin, mid, end);
}

void printArr(int arr[], int size)  {
    for (auto i = 0; i < size; i++) {
        cout << arr[i] << " ";
    }
    cout << "\n\n" << endl;
}


int main()  {
    int n = rand() % 100;
    int arr[n];
    for (int i = 0; i < n; i++) {
        arr[i] = rand() % 100;
    }
    cout << "Given array: \n";
    printArr(arr, n);

    auto start = chrono::high_resolution_clock::now();
    // unsync the I/O of C and C++.
    ios_base::sync_with_stdio(false);
```

```
    mergeSort(arr, 0, n - 1);

    auto end = chrono::high_resolution_clock::now();

    cout << "\nSorted array: \n";
    printArr(arr, n);

    double time_taken = chrono::duration_cast<chrono::nanoseconds>(end -
start).count();
    time_taken *= 1e-9;

    cout << "Time difference is: " << time_taken << setprecision(6);

    return 0;
}
```

Output:



Source Code:

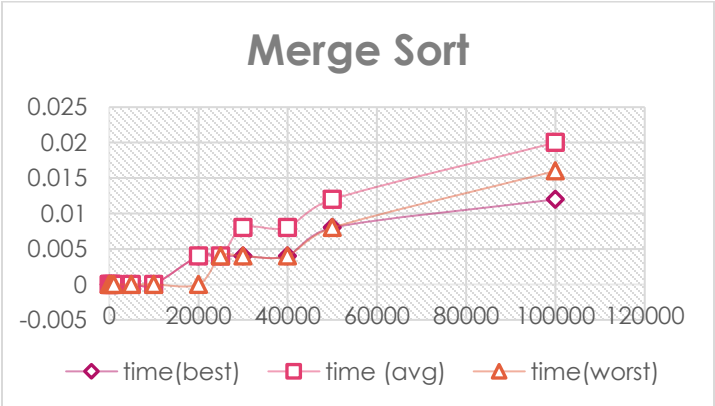
```
#include <bits/stdc++.h>
#include <chrono>

using namespace std;

// merging 2 arrays
void merge(int arr[], int const left, int const mid, int const right) {
	auto const subArr1 = mid - left + 1;
	auto const subArr2 = right - mid;

	auto *leftArr = new int[subArr1],
		*rightArr = new int[subArr2];

	for (auto i = 0; i < subArr1; i++)
		leftArr[i] = arr[left + i];

	for (auto j = 0; j < subArr2; j++)
		rightArr[j] = arr[mid + 1 + j];

	auto indexOfSubArr1 = 0,
		indexOfSubArr2 = 0;
	int indexOfMergedArr = left;

    // sorted merging of left and right arrays
	while (indexOfSubArr1 < subArr1 && indexOfSubArr2 < subArr2) {
		if (leftArr[indexOfSubArr1] <= rightArr[indexOfSubArr2]) {
			arr[indexOfMergedArr] = leftArr[indexOfSubArr1];
			indexOfSubArr1++;
		}
		else {
			arr[indexOfMergedArr] = rightArr[indexOfSubArr2];
			indexOfSubArr2++;
		}
		indexOfMergedArr++;
	}

	// check and add for remaining element in left arr
	while (indexOfSubArr1 < subArr1) {
		arr[indexOfMergedArr] = leftArr[indexOfSubArr1];
		indexOfSubArr1++;
		indexOfMergedArr++;
	}
```

```
      // check and add for remaining element in left arr
      while (indexOfSubArr2 < subArr2) {
            arr[indexOfMergedArr] = rightArr[indexOfSubArr2];
            indexOfSubArr2++;
            indexOfMergedArr++;
      }
}

void mergeSort(int array[], int const begin, int const end) {
      if (begin >= end) {
            return; // Returns recursively
    }
      auto mid = begin + (end - begin) / 2;
      mergeSort(array, begin, mid);
      mergeSort(array, mid + 1, end);
      merge(array, begin, mid, end);
}

void printArr(int arr[], int size)  {
      for (auto i = 0; i < size; i++) {
            cout << arr[i] << " ";
    }
    cout << "\n\n" << endl;
}


int main()  {
    int n = rand() % 100;
    int arr[n];
    for (int i = 0; i < n; i++) {
        arr[i] = rand() % 100;
    }
      cout << "Given array: \n";
      printArr(arr, n);

    auto start = chrono::high_resolution_clock::now();
    // unsync the I/O of C and C++.
    ios_base::sync_with_stdio(false);

      mergeSort(arr, 0, n - 1);

    auto end = chrono::high_resolution_clock::now();

      cout << "\nSorted array: \n";
```

```
      printArr(arr, n);

    double time_taken = chrono::duration_cast<chrono::nanoseconds>(end -
start).count();
    time_taken *= 1e-9;

    cout << "Time difference is: " << time_taken << setprecision(6);

      return 0;
}
```

Output:

```
input
Given array:
86 77 15 93 35 86 92 49 21 62 27 90 59 63 26 40 26 72 36 11 68 67 29 82 30 6
2 23 67 35 29 2 22 58 69 67 93 56 11 42 29 73 21 19 84 37 98 24 15 70 13 26
91 80 56 73 62 70 96 81 5 25 84 27 36 5 46 29 13 57 24 95 82 45 14 67 34 64
43 50 87 8 76 78



Sorted array:
2 5 5 8 11 11 13 13 14 15 15 19 21 21 22 23 24 24 25 26 26 26 27 27 29 29 29
 29 30 34 35 35 36 36 37 40 42 43 45 46 49 50 56 56 57 58 59 62 62 62 63 64
67 67 67 67 68 69 70 70 72 73 73 76 77 78 80 81 82 82 84 84 86 86 87 90 91 9
2 93 93 95 96 98


Time difference is: 5.0381e-05

...Program finished with exit code 0
Press ENTER to exit console.
```



Batch Analysis:

Source Code:

```
#include <bits/stdc++.h>
#include <chrono>

using namespace std;

// merging 2 arrays
void merge(int arr[], int const left, int const mid, int const right) {
    auto const subArr1 = mid - left + 1;
    auto const subArr2 = right - mid;

    auto *leftArr = new int[subArr1],
        *rightArr = new int[subArr2];

    for (auto i = 0; i < subArr1; i++)
        leftArr[i] = arr[left + i];

    for (auto j = 0; j < subArr2; j++)
        rightArr[j] = arr[mid + 1 + j];

    auto indexOfSubArr1 = 0,
        indexOfSubArr2 = 0;
    int indexOfMergedArr = left;

    // sorted merging of left and right arrays
    while (indexOfSubArr1 < subArr1 && indexOfSubArr2 < subArr2) {
        if (leftArr[indexOfSubArr1] <= rightArr[indexOfSubArr2]) {
            arr[indexOfMergedArr] = leftArr[indexOfSubArr1];
            indexOfSubArr1++;
        }
        else {
            arr[indexOfMergedArr] = rightArr[indexOfSubArr2];
            indexOfSubArr2++;
        }
        indexOfMergedArr++;
    }

    // check and add for remaining element in left arr
    while (indexOfSubArr1 < subArr1) {
        arr[indexOfMergedArr] = leftArr[indexOfSubArr1];
        indexOfSubArr1++;
```

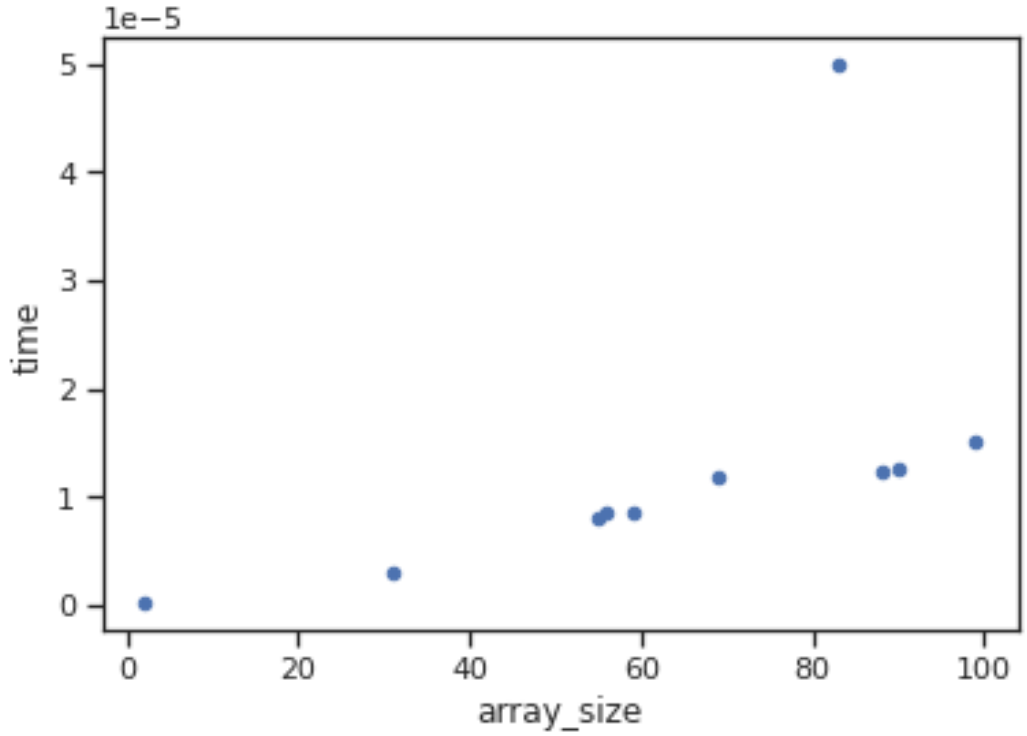
```
        indexOfMergedArr++;
    }

    // check and add for remaining element in left arr
    while (indexOfSubArr2 < subArr2) {
        arr[indexOfMergedArr] = rightArr[indexOfSubArr2];
        indexOfSubArr2++;
        indexOfMergedArr++;
    }
}

void mergeSort(int array[], int const begin, int const end) {
    if (begin >= end) {
        return; // Returns recursively
    }
    auto mid = begin + (end - begin) / 2;
    mergeSort(array, begin, mid);
    mergeSort(array, mid + 1, end);
    merge(array, begin, mid, end);
}

void printArray(double arr[], int size)
{
    for (int i = 0; i < size; i++)
        cout << arr[i] << ", ";
    cout << endl;
}

void printArray(int arr[], int size)
{
    for (int i = 0; i < size; i++)
        cout << arr[i] << ", ";
    cout << endl;
}

void printArray(float arr[], int size)
{
    for (int i = 0; i < size; i++)
        cout << arr[i] << ", ";
    cout << endl;
}

double sortApp(int n)  {
    // int n = rand() % 100;
    int arr[n];
```

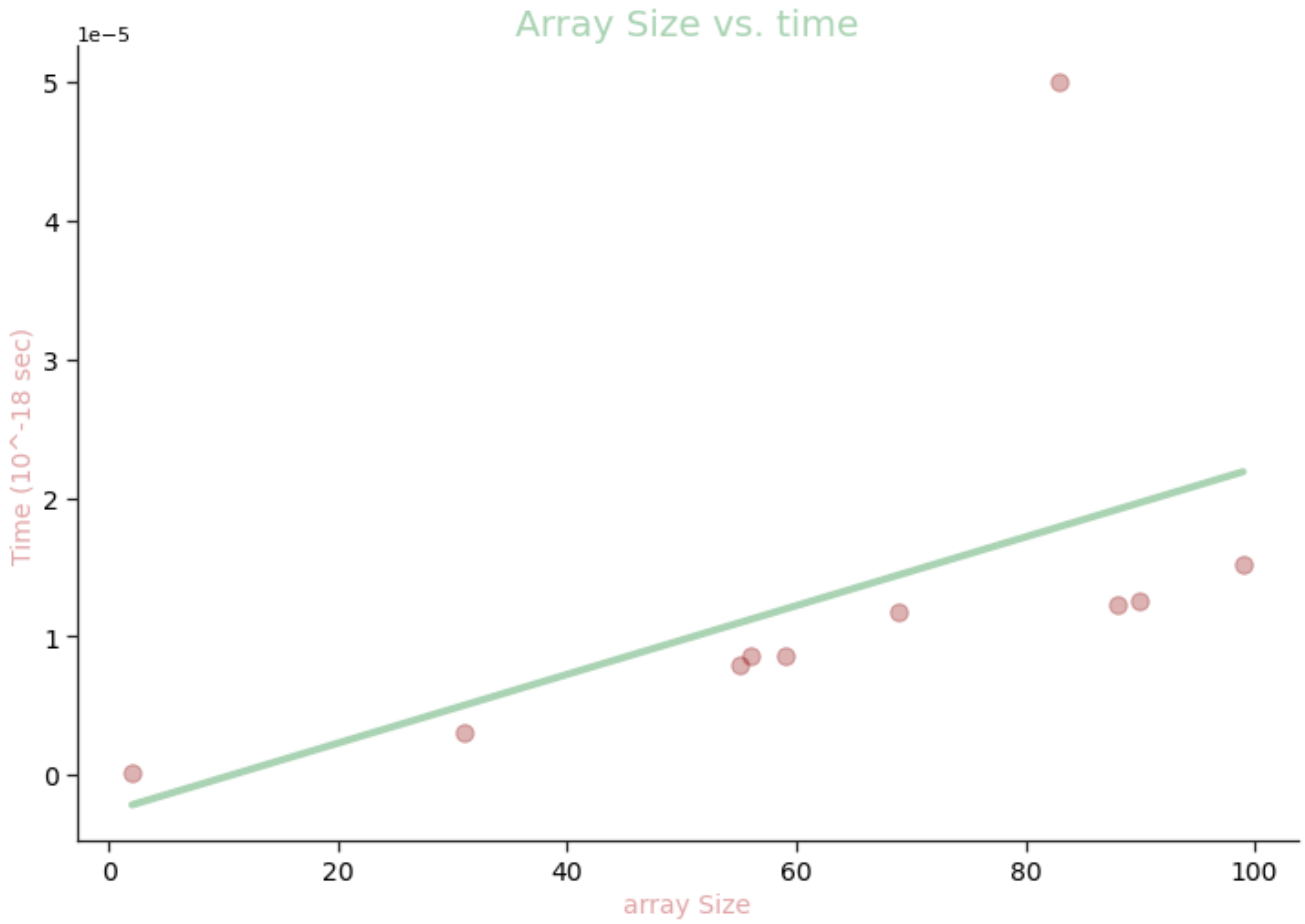
```
    for (int i = 0; i < n; i++) {
        arr[i] = rand() % 100;
    }
//  cout << "Given array: \n";
//  printArr(arr, n);

    auto start = chrono::high_resolution_clock::now();
    // unsync the I/O of C and C++.
    ios_base::sync_with_stdio(false);

    mergeSort(arr, 0, n - 1);

    auto end = chrono::high_resolution_clock::now();

//  cout << "\nSorted array: \n";
//  printArr(arr, n);

    double time_taken = chrono::duration_cast<chrono::nanoseconds>(end -
start).count();
    time_taken *= 1e-9;

    // cout << "Time difference is: " << time_taken << setprecision(6);

    return time_taken;
}

int main() {
    double times[10];
    int ns[10];
    for (int x = 0; x < 10; x++)
    {
        int n = rand() % 100;
        ns[x] = n;
        times[x] = sortApp(n);
    }
    cout << "value of n's: " << endl;
    printArray(ns, 10);
    cout << "time for each n: " << endl;
    printArray(times, 10);
}
```

Output:

```
input
value of n's:
83, 88, 56, 69, 90, 59, 2, 99, 55, 31,
time for each n:
7.1799e-05, 2.4215e-05, 1.657e-05, 2.1988e-05, 2.472e-05, 1.6775e-05, 2.34e-
07, 2.9243e-05, 1.5824e-05, 6.567e-06,


...Program finished with exit code 0
Press ENTER to exit console.
```

```
arraySize = [83, 88, 56, 69, 90, 59, 2, 99, 55, 31]
time = [5.0004e-05, 1.2248e-05, 8.572e-06, 1.1836e-05, 1.2539e-05, 8.589e-
06, 1.34e-07, 1.5209e-05, 7.996e-06, 3.056e-06]
```




Array Size vs. time
1e−5
5
4
3
2
1
0
Time (10^-18 sec)
0
20
40
60
80
100
array Size

Merge Sort

Viva Questions

1. Why is time complexity of merge sort?

Ans.

$T(n) = 2T(n/2) + \theta(n)$

Time complexity of Merge Sort is $\theta(nLogn)$ in all 3 cases (worst, average and best) as merge sort always divides the array into two halves and takes linear time to merge two halves.

O(nlogn)

Auxiliary Space: O(n)
Algorithmic Paradigm: Divide and Conquer
Sorting In Place: No in a typical implementation
Stable: Yes

2. Is it possible to do merge sort in place?

Ans.

There do exist in-place merge sorts. They must be implemented carefully. First, naive in-place merge such as described here isn't the right solution. It downgrades the performance to $O(N^2)$.

3. What are the total number of passes in merge sort of n numbers?

Ans.

log(n)

Suppose an array of n elements.

Now iterate through this array just one time.The time complexity of this operation is O(n). This time complexity tells us that we are iterating one time through an array of length n.

Now iterate through this array n times.The time complexity of this operation is O(n*n). This time complexity tells us that we are iterating n times through an array of length n.

Now the time complexity of merge sort is O(nlog(n)). This says that we are iterating through an n element array, log(n) times.

4. Given two sorted lists of size m, n then what are the number of comparisons needed in the worst case by merge sort?

Ans.

To merge two lists of size m and n, we need to do **m+n-1 comparisons** in worst case.

5. What is the output of merge sort after the 2nd pass given the following sequence of numbers: 3,41,52,26,38,57,9,49

Ans.

3,26,41,52,9,38,49,57

6. Give any application of merge sort?

Ans.

- 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

3.2 Quick Sort:

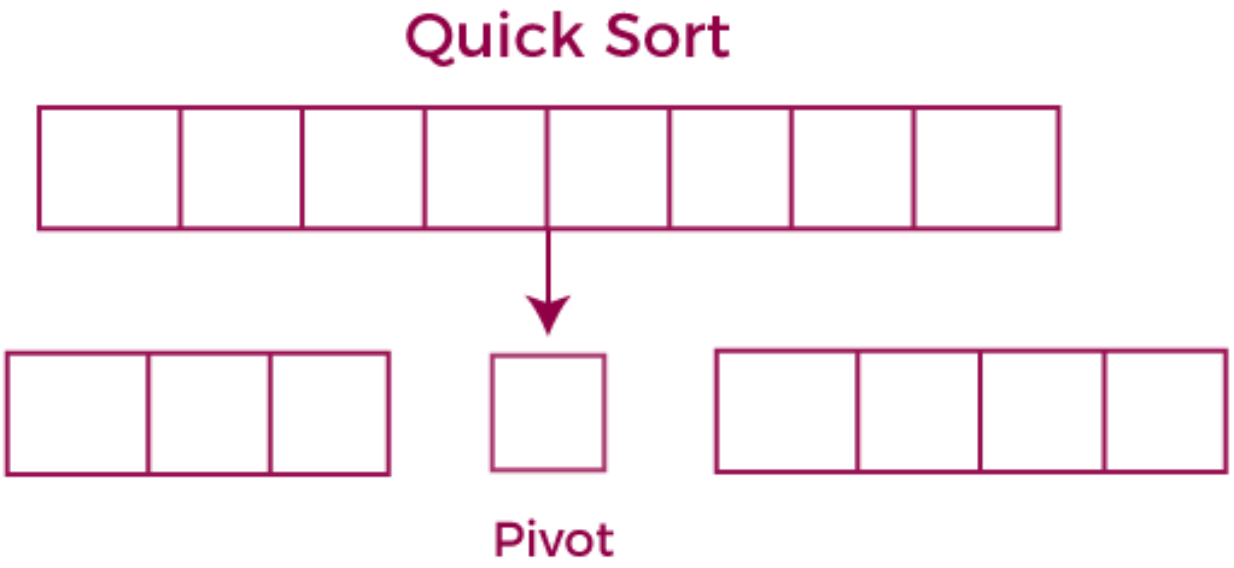
Sorting is a way of arranging items in a systematic manner. Quicksort is the widely used sorting algorithm that makes **n log n** comparisons in average case for sorting an array of n elements. It is a faster and highly efficient sorting algorithm. This algorithm follows the divide and conquer approach. Divide and conquer is a technique of breaking down the algorithms into subproblems, then solving the subproblems, and combining the results back together to solve the original problem.

Divide: In Divide, first pick a pivot element. After that, partition or rearrange the array into two sub-arrays such that each element in the left sub-array is less than or equal to the pivot element and each element in the right sub-array is larger than the pivot element.

Conquer: Recursively, sort two subarrays with Quicksort.

Combine: Combine the already sorted array.

Quicksort picks an element as pivot, and then it partitions the given array around the picked pivot element. In quick sort, a large array is divided into two arrays in which one holds values that are smaller than the specified value (Pivot), and another array holds the values that are greater than the pivot. After that, left and right sub-arrays are also partitioned using the same approach. It will continue until the single element remains in the sub-array.



Choosing the pivot

Picking a good pivot is necessary for the fast implementation of quicksort. However, it is typical to determine a good pivot. Some of the ways of choosing a pivot are as follows -

- Pivot can be random, i.e. select the random pivot from the given array.
- Pivot can either be the rightmost element of the leftmost element of the given array.
- Select median as the pivot element.

Pseudo code

Algorithm:

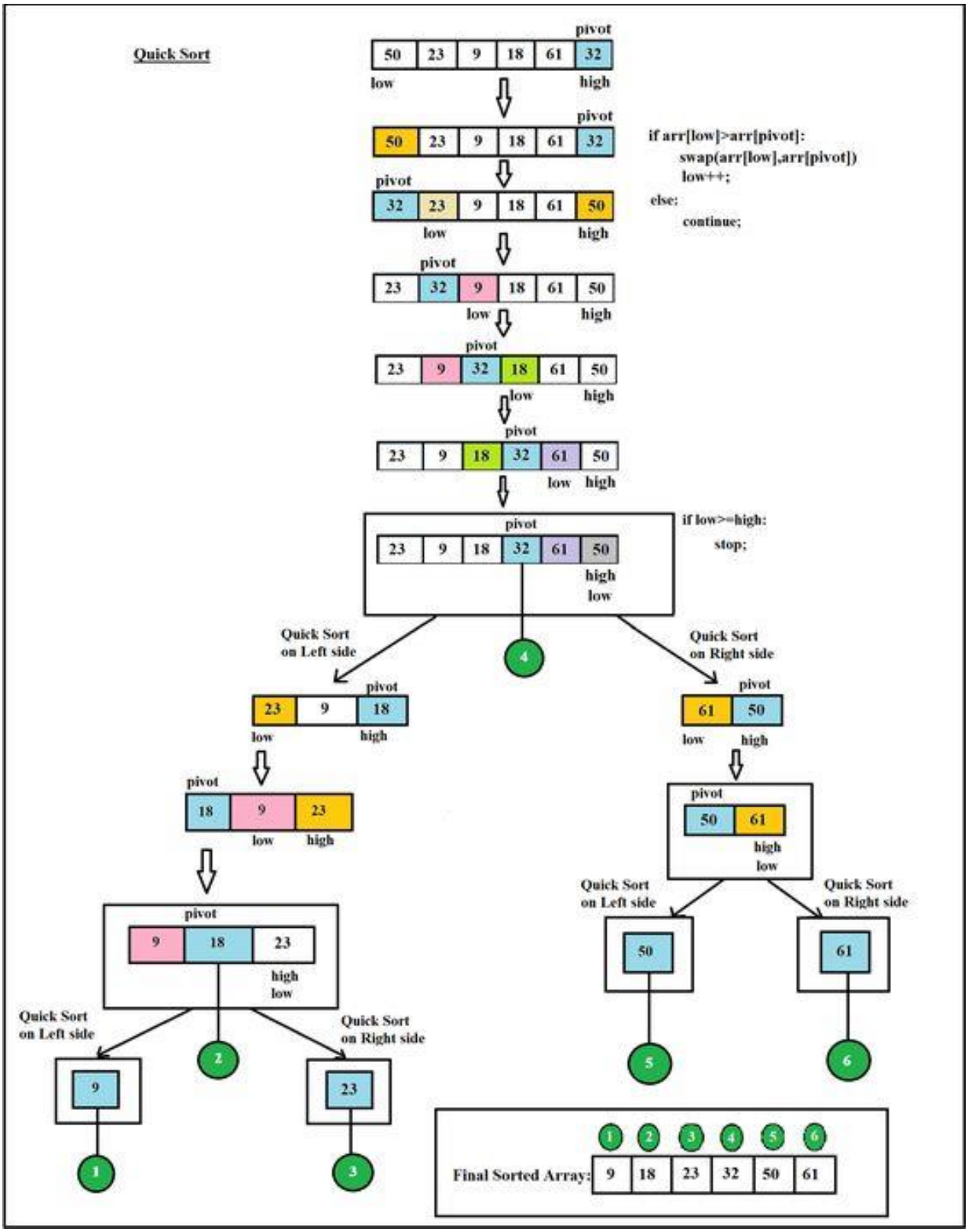
```
1. QUICKSORT (array A, start, end)
2. {
3.  1 if (start < end)
4.  2 {
5. 3 p = partition(A, start, end)
6. 4 QUICKSORT (A, start, p - 1)
7. 5 QUICKSORT (A, p + 1, end)
8. 6 }
9. }
```

Partition Algorithm:

The partition algorithm rearranges the sub-arrays in a place.

```
1. PARTITION (array A, start, end)
2. {
3.  1 pivot ? A[end]
4.  2 i ? start-1
5.  3 for j ? start to end -1 {
6.  4 do if (A[j] < pivot) {
7.  5 then i ? i + 1
8.  6 swap A[i] with A[j]
9.  7  }}
10.      8 swap A[i+1] with A[end]
11.      9 return i+1
12.     }
```

Example:


Quick Sort
pivot
50 23 9 18 61 32
low
high
pivot
50 23 9 18 61 32
pivot
32 23 9 18 61 50
low
high
pivot
23 32 9 18 61 50
low
high
pivot
23 9 32 18 61 50
low
high
pivot
23 9 18 32 61 50
low high
pivot
23 9 18 32 61 50
high
low
if arr[low]>arr[pivot]:
swap(arr[low],arr[pivot])
low++;
else:
continue;
if low>=high:
stop;
Quick Sort on Left side
Quick Sort on Right side
4
pivot
23 9 18
low
high
pivot
18 9 23
low
high
pivot
9 18 23
high
low
Quick Sort on Left side
Quick Sort on Right side
2
9
23
1
3
pivot
61 50
low
high
pivot
50 61
high
low
Quick Sort on Left side
Quick Sort on Right side
50
61
5
6
Final Sorted Array:
1 2 3 4 5 6
9 18 23 32 50 61

Result and Analysis

Time complexity

Case	Time Complexity
Best Case	O(n*logn)
Average Case	O(n*logn)
Worst Case	$O(n^2)$

- **Best Case Complexity -** In Quicksort, the best-case occurs when the pivot element is the middle element or near to the middle element. The best-case time complexity of quicksort is **O(n*logn)**.

- **Average Case Complexity -** It occurs when the array elements are in jumbled order that is not properly ascending and not properly descending. The average case time complexity of quicksort is **O(n*logn)**.

- **Worst Case Complexity -** In quick sort, worst case occurs when the pivot element is either greatest or smallest element. Suppose, if the pivot element is always the last element of the array, the worst case would occur when the given array is sorted already in ascending or descending order. The worst-case time complexity of quicksort is **$O(n^2)$**.

Space Complexity

Space Complexity	O(n*logn)
Stable	NO

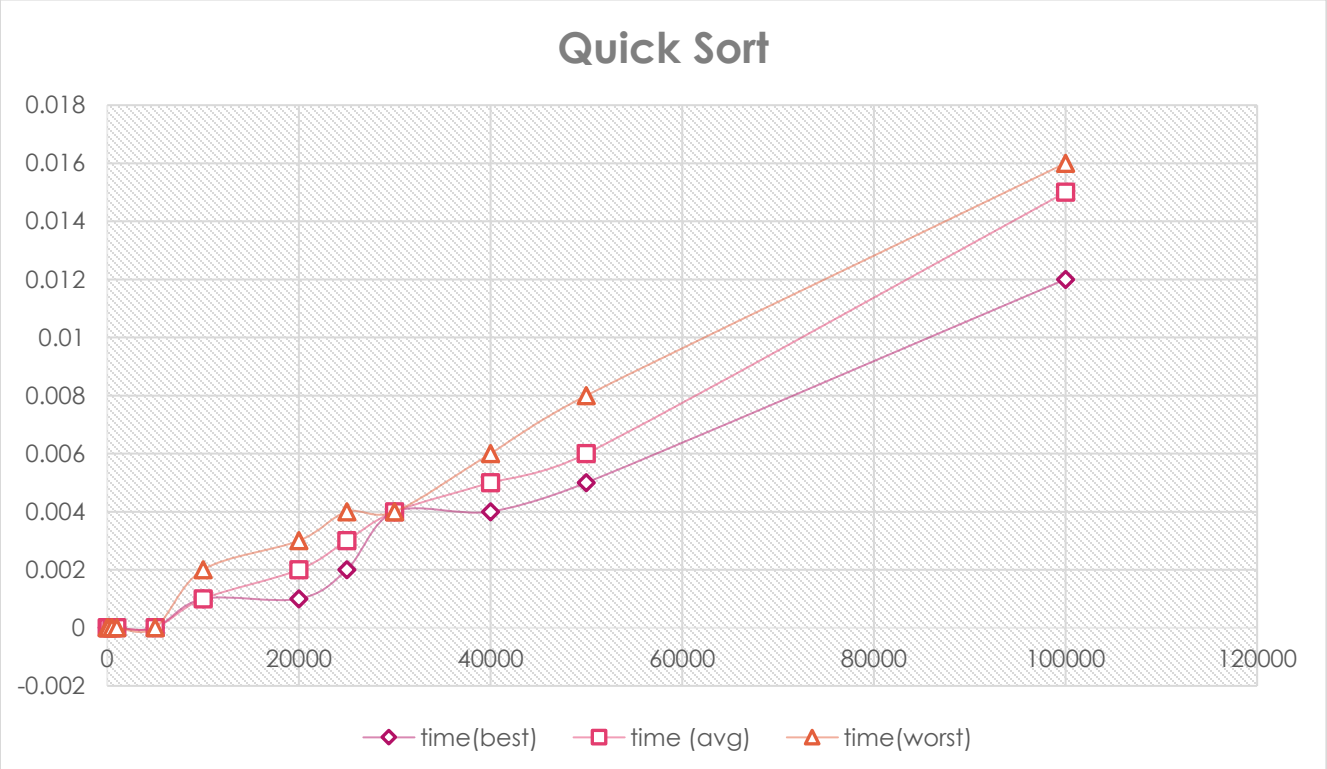
Source Code:

```
#include <bits/stdc++.h>
#include <chrono>

using namespace std;

// A utility function to swap two elements
void swap(int *a, int *b)
{
    int temp = *a;
    *a = *b;
    *b = temp;
}

// last element pivot and lower in 1 half and greater in 1 half divide
int partition(int arr[], int low, int high)
{
    int pivot = arr[high];
    int i = (low - 1);     // Index of smaller element and indicates the right
position of pivot found so far

    for (int j = low; j <= high - 1; j++)
    {
        // j == current element
        if (arr[j] < pivot)
        {
            i++; // increment index of smaller element
            swap(&arr[i], &arr[j]);
        }
    }
    swap(&arr[i + 1], &arr[high]);
    return (i + 1);
}

void quickSort(int arr[], int low, int high)
{
    if (low < high)
    {
        int p = partition(arr, low, high);

        // Separately sort elements before partition and after partition
```

```
        quickSort(arr, low, p - 1);
        quickSort(arr, p + 1, high);
    }
}

void printArray(int arr[], int size)
{
    int i;
    for (i = 0; i < size; i++)
        cout << arr[i] << " ";
    cout << "\n\n" << endl;
}

int main()
{
    int n = rand() % 100;
    int arr[n];
    for (int i = 0; i < n; i++)
    {
        arr[i] = rand() % 100;
    }
    cout << "Given array: \n";
    printArray(arr, n);

    auto start = chrono::high_resolution_clock::now();

    quickSort(arr, 0, n - 1);

    auto end = chrono::high_resolution_clock::now();

    cout << "\nSorted array: \n";
    printArray(arr, n);

    double time_taken = chrono::duration_cast<chrono::nanoseconds>(end -
start).count();
    time_taken *= 1e-9;

    cout << "Time difference is: " << time_taken << setprecision(6);

    return 0;
}
```

Output:

```
input
Given array:
86 77 15 93 35 86 92 49 21 62 27 90 59 63 26 40 26 72 36 11 68 67 29 82 30 6
2 23 67 35 29 2 22 58 69 67 93 56 11 42 29 73 21 19 84 37 98 24 15 70 13 26
91 80 56 73 62 70 96 81 5 25 84 27 36 5 46 29 13 57 24 95 82 45 14 67 34 64
43 50 87 8 76 78



Sorted array:
2 5 5 8 11 11 13 13 14 15 15 19 21 21 22 23 24 24 25 26 26 26 27 27 29 29 29
 29 30 34 35 35 36 36 37 40 42 43 45 46 49 50 56 56 57 58 59 62 62 62 63 64
67 67 67 67 68 69 70 70 72 73 73 76 77 78 80 81 82 82 84 84 86 86 87 90 91 9
2 93 93 95 96 98


Time difference is: 3.4026e-05

...Program finished with exit code 0
Press ENTER to exit console.
```



Batch Analysis:

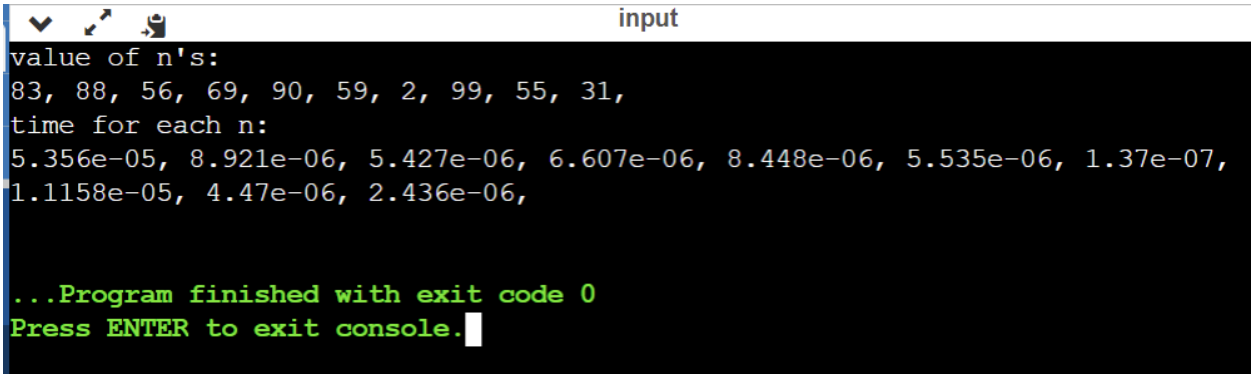
Source Code:

```
#include <bits/stdc++.h>
#include <chrono>

using namespace std;

// A utility function to swap two elements
void swap(int *a, int *b)
{
    int temp = *a;
    *a = *b;
    *b = temp;
}

// last element pivot and lower in 1 half and greater in 1 half divide
int partition(int arr[], int low, int high)
{
    int pivot = arr[high];
    int i = (low - 1);     // Index of smaller element and indicates the right
position of pivot found so far

    for (int j = low; j <= high - 1; j++)
    {
        // j == current element
        if (arr[j] < pivot)
        {
            i++; // increment index of smaller element
            swap(&arr[i], &arr[j]);
        }
    }
    swap(&arr[i + 1], &arr[high]);
    return (i + 1);
}

void quickSort(int arr[], int low, int high)
{
    if (low < high)
    {
        int p = partition(arr, low, high);

        // Separately sort elements before partition and after partition
```

```
        quickSort(arr, low, p - 1);
        quickSort(arr, p + 1, high);
    }
}

void printArray(double arr[], int size)
{
    for (int i = 0; i < size; i++)
        cout << arr[i] << ", ";
    cout << endl;
}

void printArray(int arr[], int size)
{
    for (int i = 0; i < size; i++)
        cout << arr[i] << ", ";
    cout << endl;
}

void printArray(float arr[], int size)
{
    for (int i = 0; i < size; i++)
        cout << arr[i] << ", ";
    cout << endl;
}

double sortApp(int n)  {
    int arr[n];
    for (int i = 0; i < n; i++) {
        arr[i] = rand() % 100;
    }

    auto start = chrono::high_resolution_clock::now();
    ios_base::sync_with_stdio(false);

      quickSort(arr, 0, n - 1);

    auto end = chrono::high_resolution_clock::now();

    double time_taken = chrono::duration_cast<chrono::nanoseconds>(end -
start).count();
    time_taken *= 1e-9;

      return time_taken;
}
```

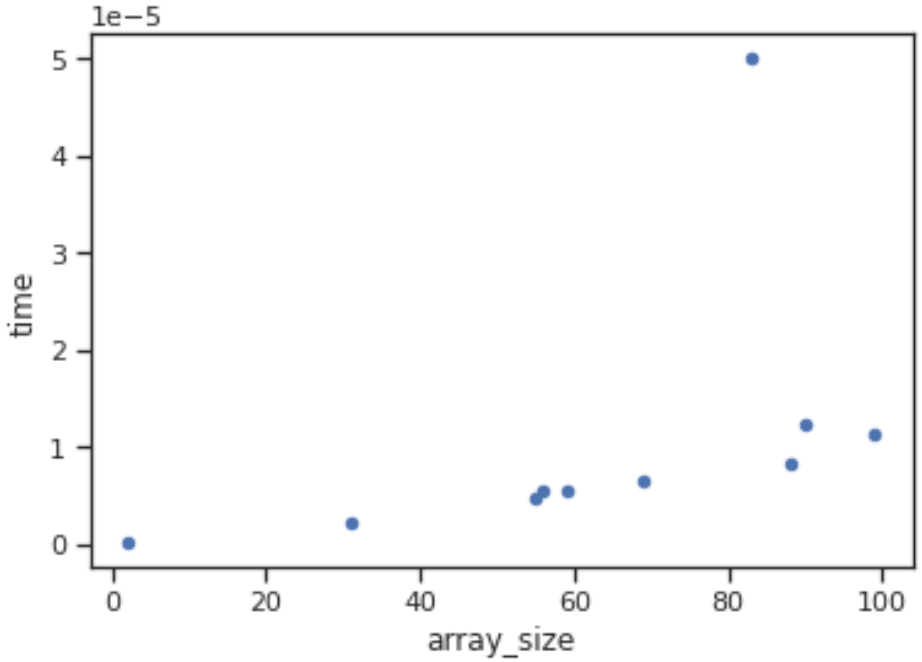
```
int main() {
    double times[10];
    int ns[10];
    for (int x = 0; x < 10; x++)
    {
        int n = rand() % 100;
        ns[x] = n;
        times[x] = sortApp(n);
    }
    cout << "value of n's: " << endl;
    printArray(ns, 10);
    cout << "time for each n: " << endl;
    printArray(times, 10);
}
```

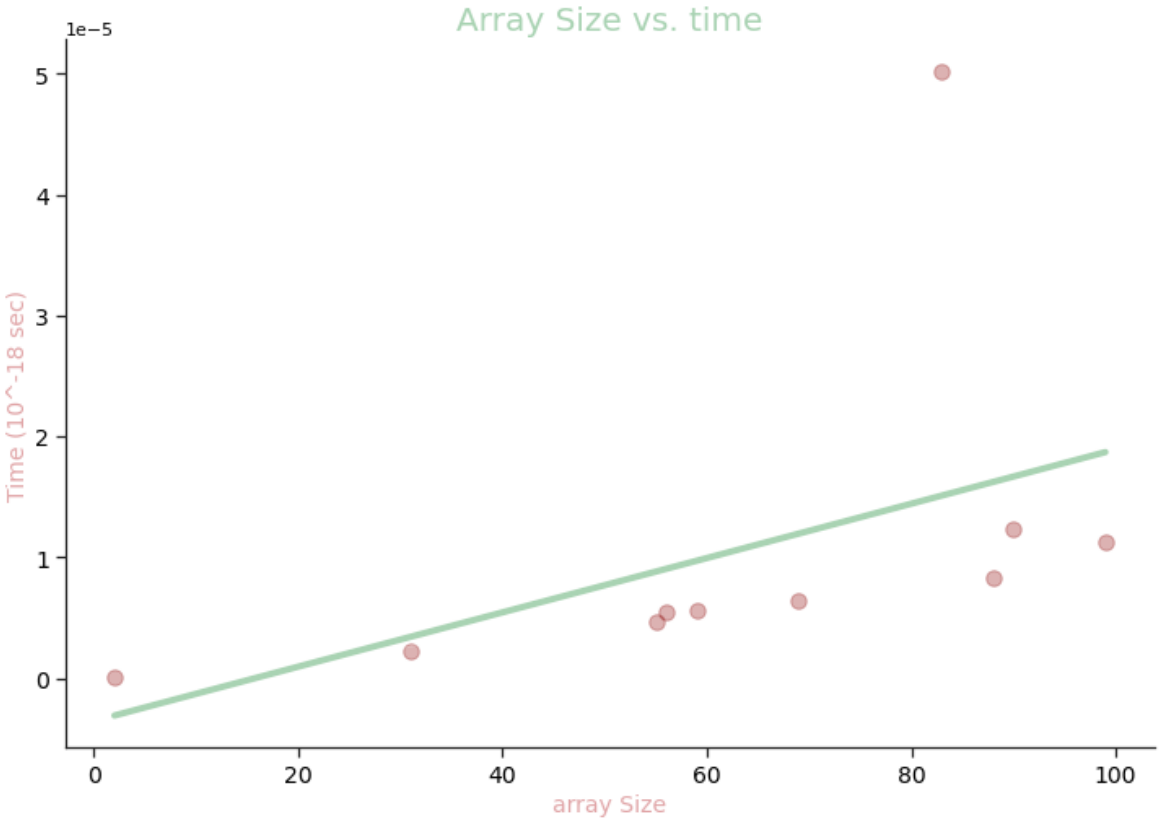
Output:



arraySize = [83, 88, 56, 69, 90, 59, 2, 99, 55, 31]

time = [5.017e-05, 8.247e-06, 5.499e-06, 6.48e-06, 1.2334e-05, 5.57e-06, 1.35e-07, 1.1314e-05, 4.678e-06, 2.284e-06]


1e−5
5
4
3
2
1
0
time
0
20
40
60
80
100
array_size


Array Size vs. time
1e−5
5
4
3
2
1
0
Time (10^-18 sec)
0
20
40
60
80
100
array Size

Quick Sort

Viva Questions

1. What value of *q* does PARTITION return when all elements in the array A [*p . . . r*] have the same value?

Ans.

If all elements are equal, then when PARTITION returns, q = r and all elements in A[p…q-1] are equal. We get the recurrence $T(n) = T(n - 1) + \Theta(n)$ for the running time, and so $T(n) = \Theta(n\ 2\ )$.

PARTITION(A, p, r) 1 x = A[r] 2 i = p - 1 3 for j = p to r - 1 4 if A[j] <= x 5 i = i + 1 6 exchange A[i] with A[j] 7 exchange A[i + 1] with A[r] 8 return i + 1 Since the if test on the analogue of line 4 is thus successful when all entries are equal, in that case the value of k when the for loop terminates is k = r-1, so the value of q returned is q = k+1 = r.

2. Give a brief argument that the running time of PARTITION on a sub array of size n is (n).

Ans.

There is a for statement whose body executes r - 1 - p = \Theta(n)r−1−p=Θ(n) times. In the worst case every time the body of the if is executed, but it takes constant time and so does the code outside of the loop. Thus the running time is \Theta(n)Θ(n).

3. How would you modify QUICKSORT to sort in non-increasing order?

Ans.

It is one of the fastest sorting algorithms known and is the method of choice in most sorting libraries. QUICKSORT is based on the divide and conquer strategy.

4. Why Quick sort is considered as best sorting?

Ans.

The cache efficiency argument has already been explained in detail. In addition, there is an intrinsic argument, why Quicksort is fast. If implemented like with two "crossing pointers", e.g. here, the inner loops have a very small body. As this is the code executed most often, this pays off.

5. What is randomized version of Quick sort?

Ans.

Randomized quick sort chooses a random element as a pivot. It is done so as to avoid the worst case of quick sort in which the input array is already sorted.

3.3 Matrix Multiplication and Strassen's Algorithm

Given a sequence of matrices *A*1, *A*2… *An*, insert parentheses so that the product of the matrices, in order, is unambiguous and needs the minimal number of multiplications

Matrix multiplication is associative: *A*1 (*A*2*A*3) = (*A*1*A*2) *A*3

A product is unambiguous if no factor is multiplied on both the left and the right and all factors are either a single matrix or an unambiguous product.

Multiplying an *i*×*j* and a *j*×*k* matrix requires *ijk* multiplications. Each element of the product requires *j* multiplications, and there are *ik* elements

$$\begin{array}{cc} & \begin{array}{cc} \vec{b_1} & \vec{b_2} \\ \downarrow & \downarrow \end{array} \\ \begin{array}{l} \vec{a_1} \rightarrow \\ \vec{a_2} \rightarrow \end{array} & \begin{bmatrix} 1 & 7 \\ 2 & 4 \end{bmatrix} \cdot \begin{bmatrix} 3 & 3 \\ 5 & 2 \end{bmatrix} = \begin{bmatrix} \vec{a_1} \cdot \vec{b_1} & \vec{a_1} \cdot \vec{b_2} \\ \vec{a_2} \cdot \vec{b_1} & \vec{a_2} \cdot \vec{b_2} \end{bmatrix} \\ & \begin{array}{cc} A & \quad B \end{array} \qquad\qquad C \end{array}$$

Number of Parenthesizations

Given the matrices *A*1,*A*2,*A*3, *A*4. Assume the dimensions of *A*1=*d*0×*d*1, etc. Below are the five possible parenthesizations of these arrays, along with the number of multiplications:

1. (*A*1*A*2)(*A*3*A*4):*d*0*d*1*d*2+*d*2*d*3*d*4+*d*0*d*2*d*4
2. ((*A*1*A*2)*A*3)*A*4:*d*0*d*1*d*2+*d*0*d*2*d*3+*d*0*d*3*d*4
3. (*A*1(*A*2*A*3))*A*4:*d*1*d*2*d*3+*d*0*d*1*d*3+*d*0*d*3*d*4
4. *A*1((*A*2*A*3)*A*4):*d*1*d*2*d*3+*d*1*d*3*d*4+*d*0*d*1*d*4

5. $A1(A2(A3A4)):d2d3d4+d1d2d4+d0d1d4$

The number of parenthesizations is at least $T(n)\geq T(n-1)+T(n-1)$.Since the number with the first element removed is $T(n-1)$, which is also the number with the last removed. Thus the number of parenthesizations is $\Omega(2n)$. The number is actually $T(n)=n-1\sum k=1T(k)T(n-k)$ which is related to the *Catalan numbers*.This is because the original product can be split into 2 sub products in *k* places. Each split is to be parenthesized optimally.This recurrence is related to the *Catalan numbers*.

Characterizing the Optimal Parenthesization

An optimal parenthesization of *A*1…*An* must break the product into two expressions, each of which is parenthesized or is a single array. Assume the break occurs at position *k*. In the optimal solution, the solution to the product *A*1…*Ak* must be optimal otherwise, we could improve *A*1…*An* by improving *A*1…*Ak* but the solution to *A*1…*An* is known to be optimal.This is a contradiction.Thus the solution to *A*1…*An* is known to be optimal.

Principle of Optimality

This problem exhibits the Principle of Optimality. The optimal solution to product *A*1…*An* contains the optimal solution to two sub products. Thus we can use Dynamic Programming. Consider a recursive solution

Then improve its performance with memoization or by rewriting bottom up

Matrix Dimensions

Consider matrix product $A1\times\cdots\times An$. Let the dimensions of matrix *Ai* be $di-1\times di$.

Thus the dimensions of matrix product $Ai\times\cdots\times Aj$ are $di-1\times dj$.

Recursive Solution

Let *M* [*i*,*j*] represent the number of multiplications required for matrix product $Ai\times\cdots\times Aj$. For $1\leq i\leq j<n$: *M* [*i*,*i*]=0 since no product is required

The optimal solution of $Ai \times Aj$ must break at some point, k, with $i \leq k < j$

Thus, $M[i,j] = M[i,k] + M[k+1,j] + di-1dkdj$

Thus, $M[i,j] = \{0 \text{ if } i=j \text{ and } \min i \leq k < j \{M[i,k] + M[k+1,j] + di-1dkdj\} \text{ if } i<j$

Pseudo code:

```
// Matrix Ai has dimension p[i-1] x p[i] for i = 1..n

MATRIX-CHAIN-ORDER (p)
n←length [p]-1
for i←1 to n
    do m[i,i]←0
for l←2to n
do for i←1  to  n-l+1
do  j←i+l-1
m[i,j]←∞
for k←ito  j-1
do q←m[i,k] + m[k+1,j]+p_{i-1}p_k p_j
                                    if   q< m[i,j]
                                     then m[i,j] ←q
                                    s[i,j] ←k
return m and s

PRINT-OPTIMAL-PARENS (s, i, j)
   if i=j
   then print "A_i"
        else print " ( "
       PRINT-OPTIMAL-PARENS (s, i, s[i,j])
        PRINT-OPTIMAL-PARENS (s, s[i,j]+1, j)
```

print ") "

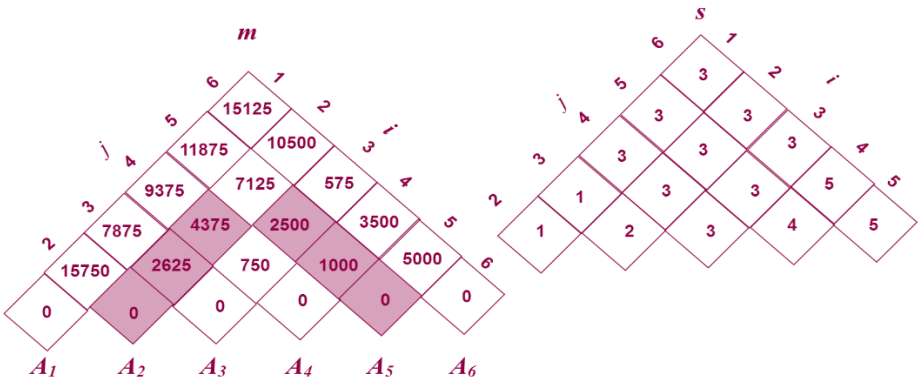
Conditions:

1. The main condition of matrix multiplication is that the number of columns of the 1st matrix must equal to the number of rows of the 2nd one.
2. As a result of multiplication you will get a new matrix that has the same quantity of rows as the 1st one has and the same quantity of columns as the 2nd one.
3. For example if you multiply a matrix of 'n' x 'k' by 'k' x 'm' size you'll get a new one of 'n' x 'm' dimension.

Sample Example:

matrix	dimension
A_1	30 x 35
A_2	35 x 15
A_3	15 x 5
A_4	5 x 10
A_5	10 x 20
A_6	20 x 25

$$m[2,5] = \min \begin{cases} m[2,2] + m[3,5] + p_1 p_2 p_5 = 0 + 2500 + 35 \cdot 15 \cdot 20 = 13000 \\ m[2,3] + m[4,5] + p_1 p_3 p_5 = 2625 + 100 + 35 \cdot 5 \cdot 20 = 7125 \\ m[2,4] + m[5,5] + p_1 p4 p_5 = 4375 + 0 + 35 \cdot 10 \cdot 20 = 11375 \end{cases}$$
$$= (7125)$$



An optimal solution can be constructed from the computed information stored in the table *s*[1...*n*, 1...*n*].The earlier matrix multiplication can be computed recursively.

p1 = a(f - h)
p2 = (a + b)h
p3 = (c + d)e
p4 = d(g - e)
p5 = (a + d)(e + h)
p6 = (b - d)(g + h)
p7 = (a - c)(e + f)

The A x B can be calculated using above seven multiplications.
Following are values of four sub-matrices of result C

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} \times \begin{bmatrix} e & f \\ g & h \end{bmatrix} = \begin{bmatrix} p5 + p4 - p2 + p6 & p1 + p2 \\ p3 + p4 & p1 + p5 - p3 - p7 \end{bmatrix}$$

A B C

A, B and C are square metrices of size N x N
a, b, c and d are submatrices of A, of size N/2 x N/2
e, f, g and h are submatrices of B, of size N/2 x N/2
p1, p2, p3, p4, p5, p6 and p7 are submatrices of size N/2 x N/2

Result and Analysis

Clearly, the space complexity of this procedure is O (n^2). Since the tables *m* and *s* require O (n^2) space. As far as the time complexity is concern, a simple inspection of the for-loop(s) structures gives us a running time of the procedure. Since, the three for-loops are nested three deep, and each one of them iterates at most *n* times (that is to say indices L, *i*, and *j* takes on at most *n* − 1 values). Therefore, the running time of this procedure is O (n^3).

Addition and Subtraction of two matrices takes $O(N^2)$ time. So time complexity can be written as

$$T(N) = 7T(N/2) + O(N^2)$$

From Master's Theorem, time complexity of this method is

$O(N^{Log7})$ which is approximately $O(N^{2.8074})$

Matrix Multiplication:

Source Code:

```
#include <bits/stdc++.h>
#include <chrono>

using namespace std;

void printMatrix(int arr[10][10], int r, int c)
{
    for (int i = 0; i < r; ++i)
        for (int j = 0; j < c; ++j)
        {
            cout << " " << arr[i][j];
            if (j == c - 1)
                cout << endl;
        }
}

int main()
{
    int a[10][10], b[10][10], multipliedMatrix[10][10], r1, c1, r2, c2, i, j, k;

    cout << "Enter rows and columns for first matrix: ";
    cin >> r1 >> c1;
    cout << "Enter rows and columns for second matrix: ";
    cin >> r2 >> c2;

    while (c1 != r2) // matrix multiplication validation check
    {
        cout << "Error! column of first matrix not equal to row of second.";

        cout << "Enter rows and columns for first matrix: ";
        cin >> r1 >> c1;

        cout << "Enter rows and columns for second matrix: ";
        cin >> r2 >> c2;
    }

    cout << endl
         << "\n Enter elements of matrix 1:" << endl;
```

```
    for (i = 0; i < r1; ++i)
        for (j = 0; j < c1; ++j)
        {
            cin >> a[i][j];
        }

    cout << endl
         << "\n Enter elements of matrix 2:" << endl;
    for (i = 0; i < r2; ++i)
        for (j = 0; j < c2; ++j)
        {
            cin >> b[i][j];
        }

    // Initializing elements of matrix multipliedMatrix to 0.
    for (i = 0; i < r1; ++i)
        for (j = 0; j < c2; ++j)
        {
            multipliedMatrix[i][j] = 0;
        }

    auto start = chrono::high_resolution_clock::now();
    // unsync the I/O of C and C++.
    ios_base::sync_with_stdio(false);

    for (i = 0; i < r1; ++i) // multiplication
        for (j = 0; j < c2; ++j)
            for (k = 0; k < c1; ++k)
            {
                multipliedMatrix[i][j] += a[i][k] * b[k][j];
            }

    auto end = chrono::high_resolution_clock::now();

    cout << "multiply of the matrix=\n";
    printMatrix(multipliedMatrix, r1, c2);

    double time_taken = chrono::duration_cast<chrono::nanoseconds>(end -
start).count();
    time_taken *= 1e-9;
    cout << "Time difference is: " << time_taken << setprecision(6) << endl;

    return 0;
}
```

Output:

```
Enter rows and columns for first matrix: 2 2
Enter rows and columns for second matrix: 3 3
Error! column of first matrix not equal to row of second.
Enter rows and columns for first matrix: 2 2
Enter rows and columns for second matrix: 2 2


 Enter elements of matrix 1:
3 4
2 1


 Enter elements of matrix 2:
1 5
3 7
multiply of the matrix=
 15 43
 5 17
Time difference is: 4.7856e-05


...Program finished with exit code 0
Press ENTER to exit console.
```

```
multiply of the matrix=
 15 43
 5 17
Time difference is: 4.7856e-05


...Program finished with exit code 0
Press ENTER to exit console.
```

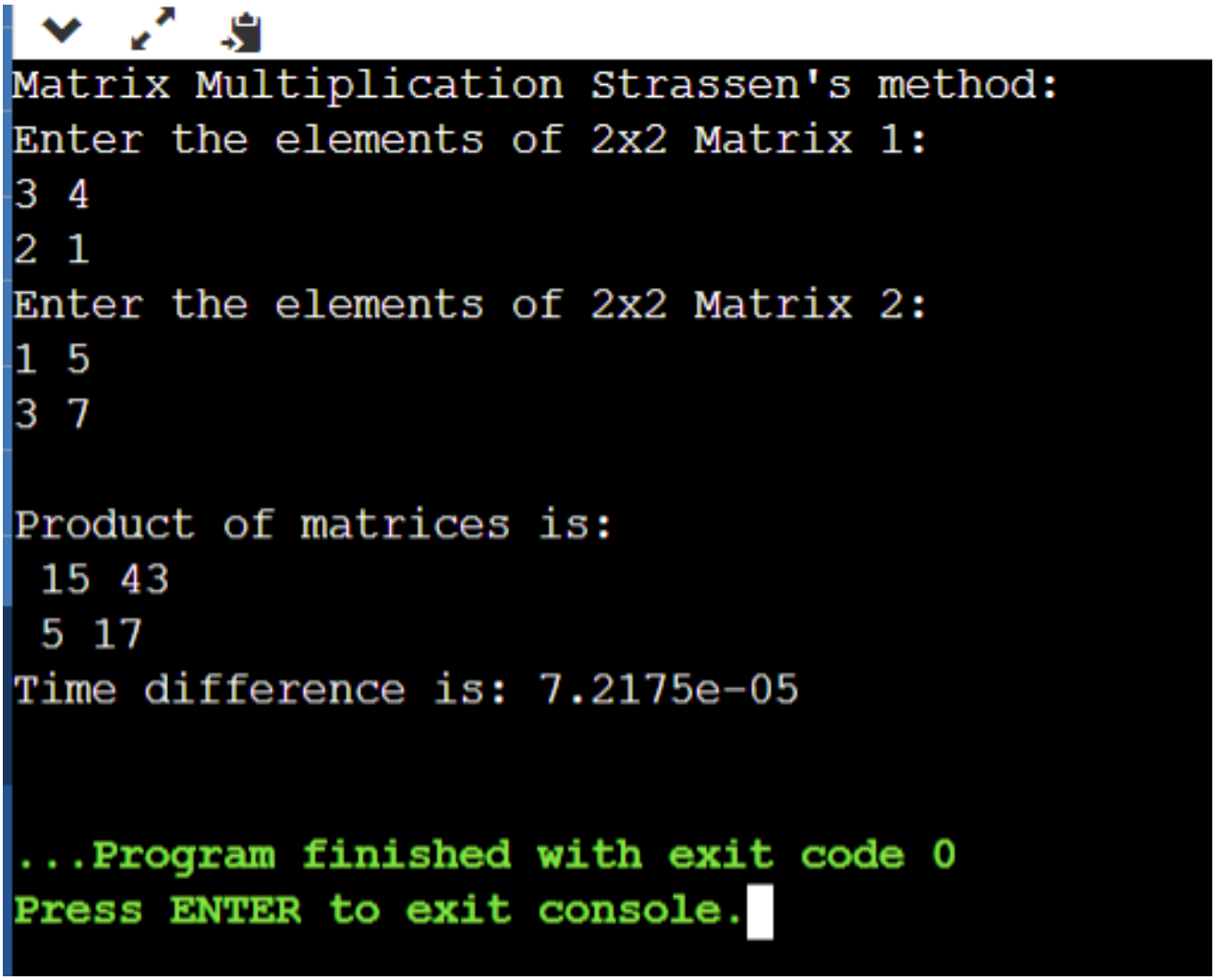
Strassen algorithm:

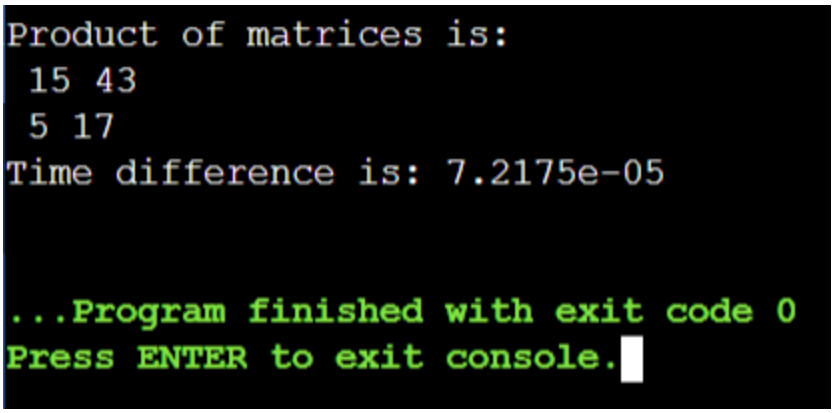
Source Code:

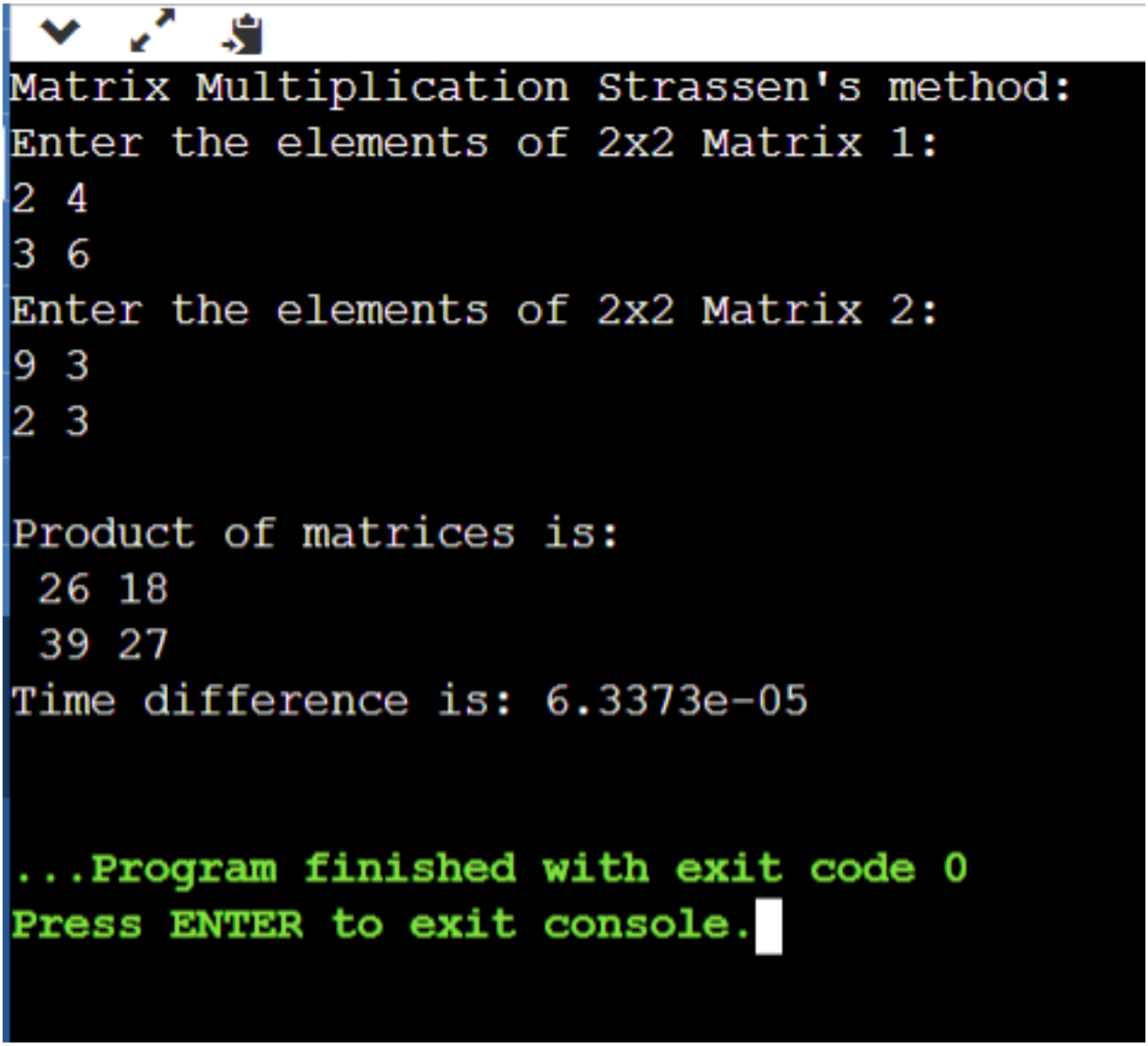
```
#include <bits/stdc++.h>
#include <chrono>

using namespace std;

void printMatrix(int arr[2][2], int r, int c)
{
    for (int i = 0; i < r; ++i)
        for (int j = 0; j < c; ++j)
        {
            cout << " " << arr[i][j];
            if (j == c - 1)
                cout << endl;
        }
}

int main()
{
    int a[2][2], b[2][2], mult[2][2];
    int m1, m2, m3, m4, m5, m6, m7, i, j;

    cout << "Matrix Multiplication Strassen's method: \n";
    cout << "Enter the elements of 2x2 Matrix 1:\n";
    for (i = 0; i < 2; i++)
    {
        for (j = 0; j < 2; j++)
        {
            cin >> a[i][j];
        }
    }

    cout << "Enter the elements of 2x2 Matrix 2:\n";
    for (i = 0; i < 2; i++)
    {
        for (j = 0; j < 2; j++)
        {
            cin >> b[i][j];
        }
```

```
    }

    auto start = chrono::high_resolution_clock::now();
    // unsync the I/O of C and C++.
    ios_base::sync_with_stdio(false);

    m1 = (a[0][0] + a[1][1]) * (b[0][0] + b[1][1]);
    m2 = (a[1][0] + a[1][1]) * b[0][0];
    m3 = a[0][0] * (b[0][1] - b[1][1]);
    m4 = a[1][1] * (b[1][0] - b[0][0]);
    m5 = (a[0][0] + a[0][1]) * b[1][1];
    m6 = (a[1][0] - a[0][0]) * (b[0][0] + b[0][1]);
    m7 = (a[0][1] - a[1][1]) * (b[1][0] + b[1][1]);

    mult[0][0] = m1 + m4 - m5 + m7;
    mult[0][1] = m3 + m5;
    mult[1][0] = m2 + m4;
    mult[1][1] = m1 - m2 + m3 + m6;

    auto end = chrono::high_resolution_clock::now();

    cout << "\nProduct of matrices is: \n";
    printMatrix(mult, 2, 2);

    double time_taken = chrono::duration_cast<chrono::nanoseconds>(end -
start).count();
    time_taken *= 1e-9;

    cout << "Time difference is: " << time_taken << setprecision(6) << endl;

    return 0;
}
```

Output:


Matrix Multiplication Strassen's method:
Enter the elements of 2x2 Matrix 1:
3 4
2 1
Enter the elements of 2x2 Matrix 2:
1 5
3 7
Product of matrices is:
15 43
5 17
Time difference is: 7.2175e-05
...Program finished with exit code 0
Press ENTER to exit console.


Product of matrices is:
15 43
5 17
Time difference is: 7.2175e-05
...Program finished with exit code 0
Press ENTER to exit console.


Matrix Multiplication Strassen's method:
Enter the elements of 2x2 Matrix 1:
2 4
3 6
Enter the elements of 2x2 Matrix 2:
9 3
2 3
Product of matrices is:
26 18
39 27
Time difference is: 6.3373e-05
...Program finished with exit code 0
Press ENTER to exit console.

Viva Questions

1. What is the recurrence relation for MCM problem?

Ans.

Matrix Chain Multiplication + Dynamic Programming + Recurrance Relation

Given a sequence of matrices, find the most efficient way to multiply these matrices together. The problem is not actually to perform the multiplications, but merely to decide in which order to perform the multiplications.
We have many options to multiply a chain of matrices because matrix multiplication is associative. In other words, no matter how we parenthesize the product, the result will be the same. For example, if we had four matrices A, B, C, and D, we would have:

(ABC)D = (AB)(CD) = A(BCD) = ...

However, the order in which we parenthesize the product affects the number of simple arithmetic operations needed to compute the product, or the efficiency. For example, suppose A is a 10 × 30 matrix, B is a 30 × 5 matrix, and C is a 5 × 60 matrix. Then,

(AB)C = (10×30×5) + (10×5×60) = 1500 + 3000 = 4500 operations

A(BC) = (30×5×60) + (10×30×60) = 9000 + 18000 = 27000 operations.

Clearly the first parenthesization requires less number of operations.
Given an array p[] which represents the chain of matrices such that the ith matrix Ai is of dimension p[i-1] x p[i]. We need to write a function MatrixChainOrder() that should return the minimum number of multiplications needed to multiply the chain.

2. Let A1, A2, A3,A4, A5 be five matrices of dimensions 2x3, 3x5,5x2,2x4,4x3 respectively. Find the minimum number of scalar multiplications required to find the product A1 A2 A3 A4 A5 using the basic matrix multiplication method?

Ans.

We have many ways to do matrix chain multiplication because matrix multiplication is associative. In other words, no matter how we parenthesize the product, the result of the matrix chain multiplication obtained will remain the same. Here we have four matrices A1, A2, A3, and A4, we would have:

((A1A2)A3)A4 = ((A1(A2A3))A4) = (A1A2)(A3A4) = A1((A2A3)A4) = A1(A2(A3A4)).

However, the order in which we parenthesize the product affects the number of simple arithmetic operations needed to compute the product, or the efficiency. Here, A1 is a 10 × 5 matrix, A2 is a 5 x 20 matrix, and A3 is a 20 x 10 matrix, and A4 is 10 x 5.

If we multiply two matrices A and B of order l x m and m x n respectively,then the number of scalar multiplications in the multiplication of A and B will be lxmxn.

Then,

The number of scalar multiplications required in the following sequence of matrices will be :

A1((A2A3)A4) = (5 x 20 x 10) + (5 x 10 x 5) + (10 x 5 x 5) = 1000 + 250 + 250 = 1500.

All other parenthesized options will require number of multiplications more than 1500.

3. Consider the two matrices P and Q which are 10 x 20 and 20 x 30 matrices respectively. What is the number of multiplications required to multiply the two matrices?

Ans.

The number of multiplications required is 10*20*30.

4. Consider the matrices P, Q and R which are 10 x 20, 20 x 30 and 30 x 40 matrices respectively. What is the minimum number of multiplications required to multiply the three matrices?

Ans.

The minimum number of multiplications are 18000. This is the case when the matrices are parenthesized as (P*Q)*R.

5. Can this problem be solved by greedy approach. Justify?

Ans.

Unfortunately, **there is no good "greedy choice"** for Matrix Chain Multiplication, meaning that for any choice that's easy to compute, there is always some input sequence of matrices for which your greedy algorithm will not find the optimum parenthesization.

At each step only least value is selected among all elements of in the array p[x, y], so that the multiplication cost is kept minimum at each step. This greedy approach ensures that the solution is optimal with least cost involved and the output is a fully parenthesized product of matrices.

EXPERIMENT - 4

Algorithms Design and Analysis Lab

Aim

To implement dynamic programming techniques and analyse its time complexity.

Syeda Reeha Quasar
14114802719
4C7

EXPERIMENT – 4

Aim:

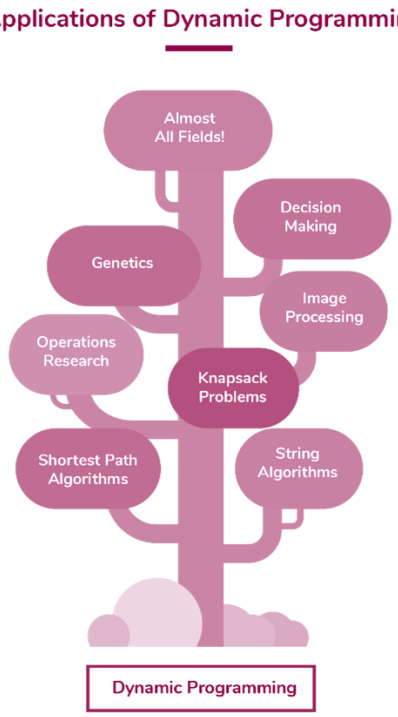
To implement dynamic programming techniques and analyse its time complexity.

Theory:

Dynamic programming solves problems by combining the solution of sub problems. It is only applicable when sub problems are not independent, that is, they share sub sub-Problems. Each time a new sub problem is solved, its solution is stored such that other sub problems sharing the stored sub problem can use the stored value instead of doing a recalculation, thereby saving work compared to applying the divide- and-conquer principle on the same problem which would have recalculated everything.

A dynamic programming solution has three components:

- Formulate the answer as a recurrence relation or recursive algorithm.
- Show that the number of different instances of your recurrence is bounded by a polynomial.
- Specify an order of evaluation for the recurrence so you always have what you need.



4.1 LCS (Longest Common Subsequence)

The longest common subsequence problem is finding the longest sequence which exists in both the given strings.

Subsequence:

Let us consider a sequence S = <s_1, s_2, s_3, s_4, …,s_n>. A sequence Z = <z_1, z_2, z_3, z_4, …,z_m> over S is called a subsequence of S, if and only if it can be derived from S deletion of some elements.

Common Subsequence:

Suppose, ***X*** and ***Y*** are two sequences over a finite set of elements. We can say that ***Z*** is a common subsequence of ***X*** and ***Y***, if ***Z*** is a subsequence of both ***X***and ***Y***.

Longest Common Subsequence:

If a set of sequences are given, the longest common subsequence problem is to find a common subsequence of all the sequences that is of maximal length. It is a classic computer science problem, the basis of data comparison programs such as the diff-utility, and has applications in bioinformatics.

Naïve Method:

Let ***X*** be a sequence of length ***m*** and ***Y*** a sequence of length ***n***. Check for every subsequence of ***X*** whether it is a subsequence of ***Y***, and return the longest common subsequence found. There are 2^m subsequences of ***X***. Testing sequences whether or not it is a subsequence of ***Y*** takes ***O(n)*** time. Thus, the naïve algorithm would take $O(n2^m)$ time.

Dynamic Programming:

Let $X = < x_1, x_2, x_3, \ldots, x_m >$ and $Y = < y_1, y_2, y_3, \ldots, y_n >$ be the sequences. To compute the length of an element the following algorithm is used. In this

procedure, table ***C[m, n]*** is computed in row major order and another table ***B[m,n]*** is computed to construct optimal solution.

Recursive Equation:

$$c[i,j] = \begin{cases} 0 & \textbf{if } i = 0 \textbf{ or } j = 0 \ , \\ c[i-1, j-1] + 1 & \textbf{if } i,j > 0 \textbf{ and } x_i = y_j \ , \\ \max(c[i, j-1], c[i-1, j]) & \textbf{if } i,j > 0 \textbf{ and } x_i \neq y_j \ . \end{cases}$$

Pseudo code:

Let X and Y are Sequences whose LCS has to be found.

LCS-Length-Table-Formulation (X, Y)

m := length(X)

n := length(Y)

for i = 1 to m do

C [i, 0] := 0

for j = 1 to n do

C [0, j] := 0

for i = 1 to m do

for j = 1 to n do

if $x_i = y_j$

C [i, j] := C [i - 1, j - 1] + 1

B [i, j] := 'D'

else

if C [i -1, j] ≥ C [i, j -1]

C [i, j] := C [i - 1, j] + 1

B [i, j] := 'U'

else

C [i, j] := C [i, j - 1] + 1

B [i, j] := 'L'

return C and B

Algorithm: Print-LCS (B, X, i, j)

if i = 0 and j = 0

return

if B[i, j] = 'D'

Print-LCS(B, X, i-1, j-1)

Print(x_i)

else if B[i, j] = 'U'

Print-LCS(B, X, i-1, j)

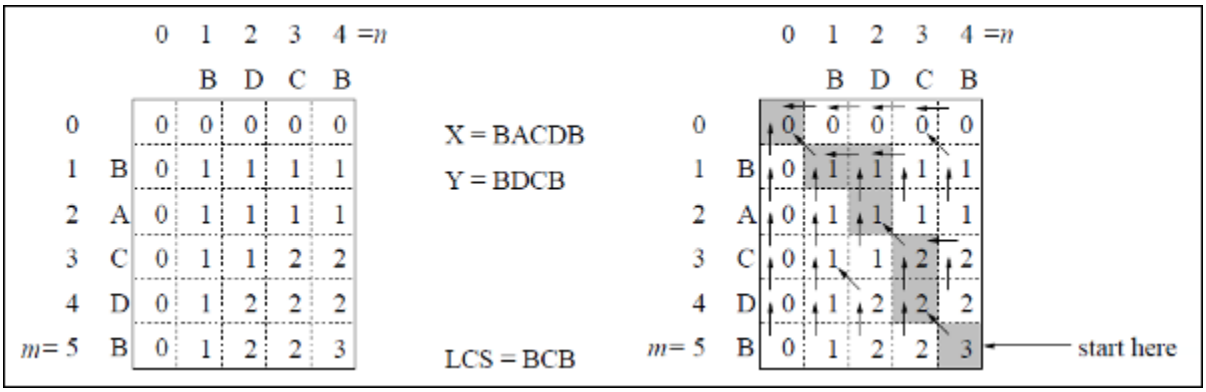
else

Print-LCS(B, X, i, j-1)

Sample Example:

We have two strings *X = BACDB* and *Y = BDCB* to find the longest common subsequence.

Following the algorithm LCS-Length-Table-Formulation (as stated above), we have calculated table C (shown on the left hand side) and table B (shown on the right hand side). In table B, instead of 'D', 'L' and 'U', we are using the diagonal arrow, left arrow and up arrow, respectively. After generating table B, the LCS is determined by function LCS-Print. The result is BCB.



Result and Analysis

To populate the table, the outer **for** loop iterates ***m*** times and the inner **for** loop iterates ***n*** times. Hence, the complexity of the algorithm is *O (m, n)*, where ***m*** and ***n*** are the length of two strings.

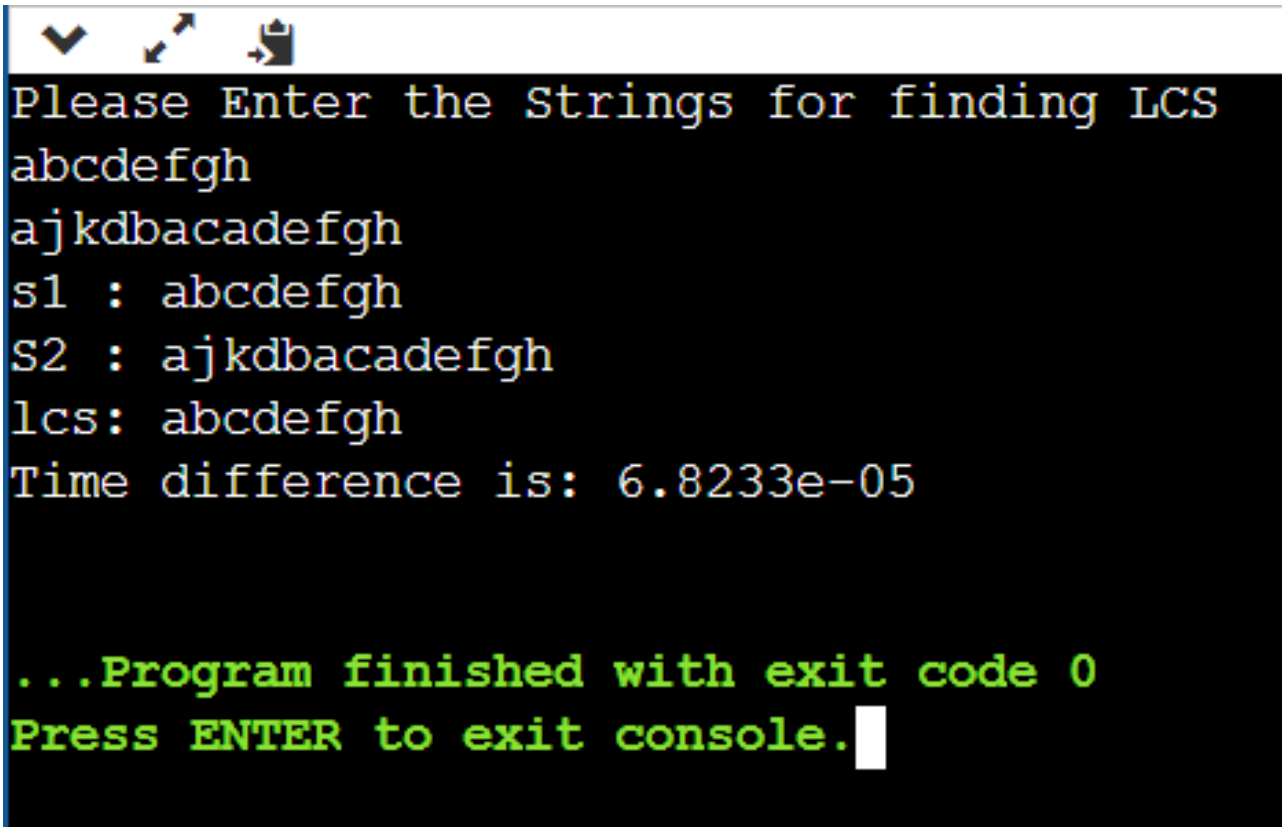
LCS:

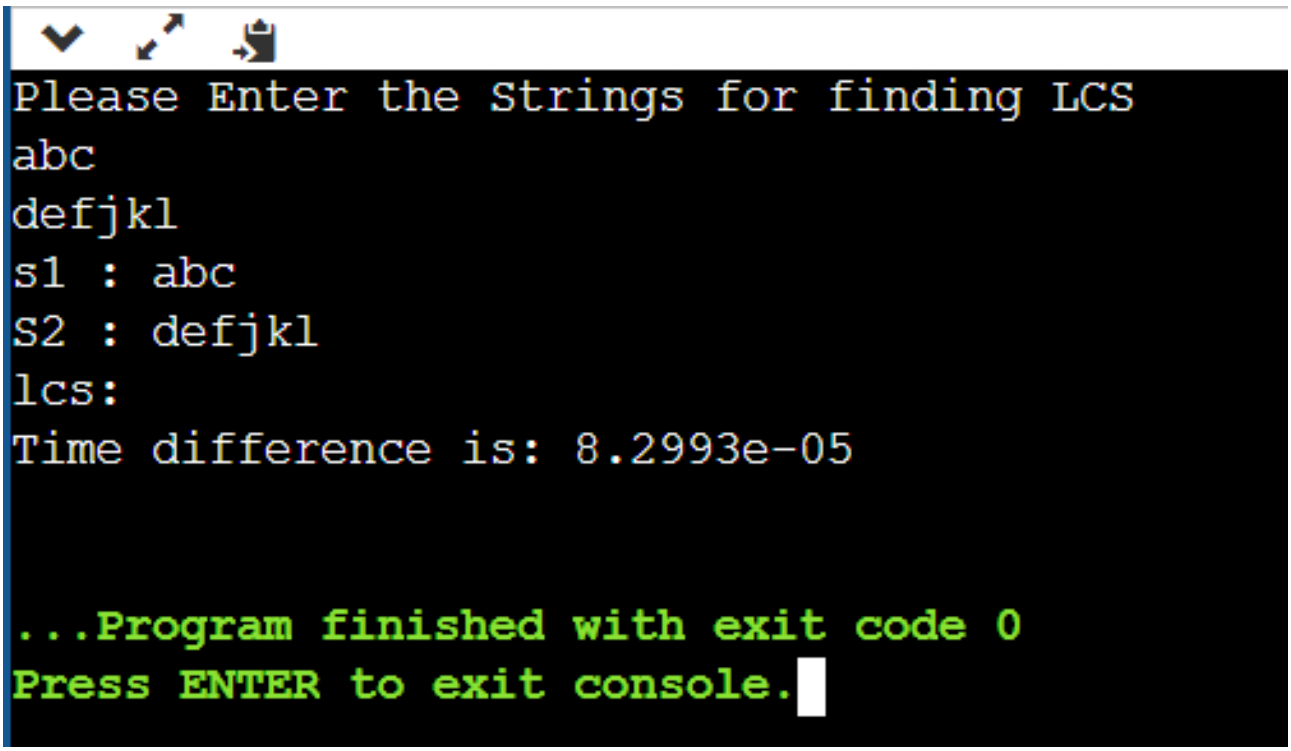
Source Code:

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

void lcs(string s1, string s2, int m, int n)
{
    int dpTable[m + 1][n + 1];

    for (int i = 0; i <= m; i++) // creating dp tale bottom up
    {
        for (int j = 0; j <= n; j++)
        {
            if (i == 0 || j == 0)
                dpTable[i][j] = 0;
            else if (s1[i - 1] == s2[j - 1])
                dpTable[i][j] = dpTable[i - 1][j - 1] + 1;
            else
                dpTable[i][j] = max(dpTable[i - 1][j], dpTable[i][j - 1]);
        }
    }

    int index = dpTable[m][n];
    char lcsAlgo[index + 1];
    lcsAlgo[index] = '\0';

    int i = m, j = n;
    while (i > 0 && j > 0)
    {
        if (s1[i - 1] == s2[j - 1])
```

```
        {
            lcsAlgo[index - 1] = s1[i - 1];
            i--;
            j--;
            index--;
        }

        else if (dpTable[i - 1][j] > dpTable[i][j - 1])
            i--;
        else
            j--;
    }
    cout << "s1 : " << s1 << "\nS2 : " << s2 << "\nlcs: " << lcsAlgo << "\n";
}

int main()
{
    // string s1 = "ACADB";
    // string s2 = "CBDA";
    string s1, s2;
    cout << "Please Enter the Strings for finding LCS" << endl;
    cin >> s1 >> s2;
    int m = s1.size();
    int n = s2.size();

    auto start = chrono::high_resolution_clock::now();
    // unsync the I/O of C and C++.
    ios_base::sync_with_stdio(false);

    lcs(s1, s2, m, n);

    auto end = chrono::high_resolution_clock::now();

    double time_taken = chrono::duration_cast<chrono::nanoseconds>(end -
start).count();
    time_taken *= 1e-9;
    cout << "Time difference is: " << time_taken << setprecision(6) << endl;
    return 0;
}
```

Output:


Please Enter the Strings for finding LCS
abcdefgh
ajkdbacadefgh
s1 : abcdefgh
S2 : ajkdbacadefgh
lcs: abcdefgh
Time difference is: 6.8233e-05
...Program finished with exit code 0
Press ENTER to exit console.


Please Enter the Strings for finding LCS
abc
defjkl
s1 : abc
S2 : defjkl
lcs:
Time difference is: 8.2993e-05
...Program finished with exit code 0
Press ENTER to exit console.

LCS Batch Analysis:

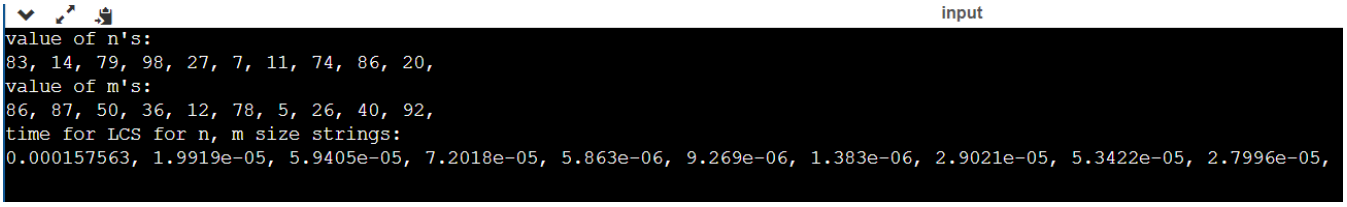
Source Code:

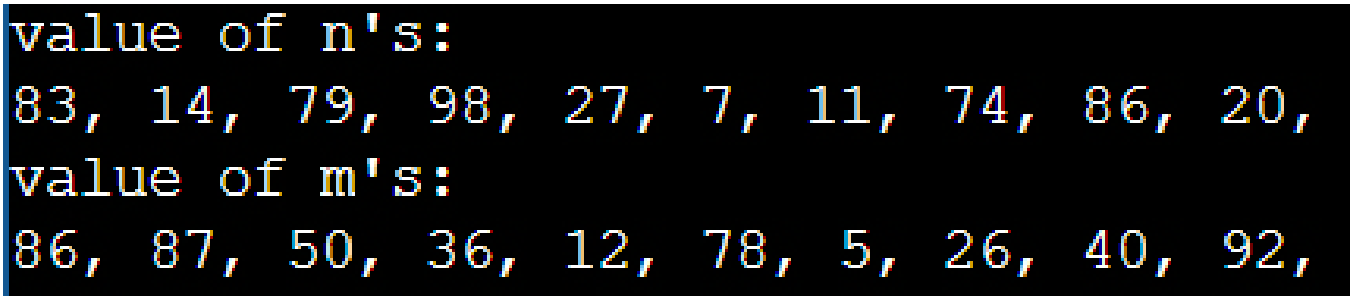
```
#include <bits/stdc++.h>
#include <chrono>
using namespace std;

double lcs(string s1, string s2, int m, int n)
{
    auto start = chrono::high_resolution_clock::now();
    // unsync the I/O of C and C++.
    ios_base::sync_with_stdio(false);

    int dpTable[m + 1][n + 1];

    for (int i = 0; i <= m; i++) // creating dp tale bottom up
    {
        for (int j = 0; j <= n; j++)
        {
            if (i == 0 || j == 0)
                dpTable[i][j] = 0;
            else if (s1[i - 1] == s2[j - 1])
                dpTable[i][j] = dpTable[i - 1][j - 1] + 1;
            else
                dpTable[i][j] = max(dpTable[i - 1][j], dpTable[i][j - 1]);
        }
    }

    int index = dpTable[m][n];
    char lcsAlgo[index + 1];
    lcsAlgo[index] = '\0';

    int i = m, j = n;
    while (i > 0 && j > 0)
    {
        if (s1[i - 1] == s2[j - 1])
        {
            lcsAlgo[index - 1] = s1[i - 1];
            i--;
            j--;
            index--;
```

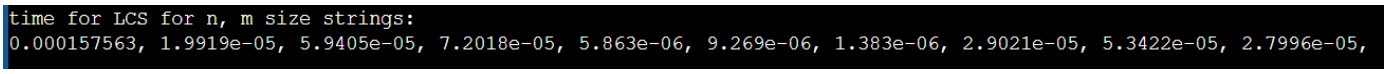
```
        }

        else if (dpTable[i - 1][j] > dpTable[i][j - 1])
            i--;
        else
            j--;
    }
    auto end = chrono::high_resolution_clock::now();

    double time_taken = chrono::duration_cast<chrono::nanoseconds>(end -
start).count();
    time_taken *= 1e-9;
    return time_taken;
}

string randomStr(const int len) {
    static const char letters[] = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
    string s;
    s.reserve(len);
    for (int i = 0; i < len; ++i) {
        s += letters[rand() % (sizeof(letters) - 1)];
    }
    return s;
}

void printArray(double arr[], int size)
{
    for (int i = 0; i < size; i++)
        cout << arr[i] << ", ";
    cout << endl;
}

void printArray(int arr[], int size)
{
    for (int i = 0; i < size; i++)
        cout << arr[i] << ", ";
    cout << endl;
}

int main()
{
    double times[10];
    int ns[10];
    int ms[10];
    for (int x = 0; x < 10; x++)
```

```
    {
        int n = rand() % 100;
        int m = rand() % 100;
        ns[x] = n;
        ms[x] = m;
        string s1 = randomStr(n);
        string s2 = randomStr(m);
        times[x] = lcs(s1, s2, n, m);
    }
    cout << "value of n's: " << endl;
    printArray(ns, 10);
    cout << "value of m's: " << endl;
    printArray(ms, 10);
    cout << "time for LCS for n, m size strings: " << endl;
    printArray(times, 10);
    return 0;
}
```

Output:







value of n's:

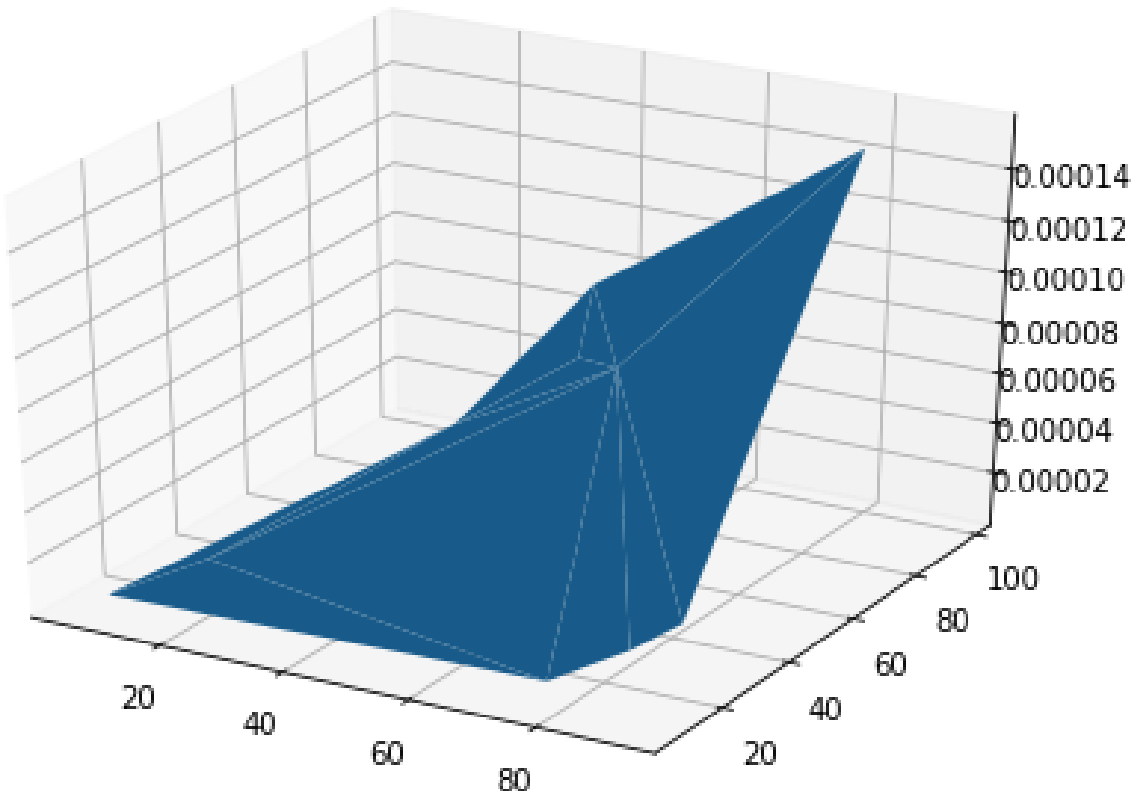
83, 14, 79, 98, 27, 7, 11, 74, 86, 20

value of m's:

86, 87, 50, 36, 12, 78, 5, 26, 40, 92

time for LCS for n, m size strings:

0.000157563, 1.9919e-05, 5.9405e-05, 7.2018e-05, 5.863e-06, 9.269e-06, 1.383e-06, 2.9021e-05, 5.3422e-05, 2.7996e-05



Viva Questions

1. Give any application of LCS problem.

Ans.

It is a classic computer science problem, the basis of diff (a file comparison program that outputs the differences between two files), and has applications in bioinformatics.

The LCS has been used to study various areas (see [2, 3]), such as text analysis, pattern recognition, file comparison, efficient tree matching [4], etc. Biological applications of the LCS and similarity measurement are varied, from sequence alignment [5] in comparative genomics [6], to phylogenetic construction and analysis, to rapid search in huge biological sequences [7], to compression and efficient storage of the rapidly expanding genomic data sets [8, 9], to re-sequencing a set of strings given a target string [10], an important step in efficient genome assembly.

2. Can LCS problem be solved by Greedy strategy?

Ans.

Yes, LCS can be solved using Greedy Strategy.

3. Find the LCS for input Sequences “ABCDGH” and “AEDFHR”

Ans.

LCS for input Sequences “ABCDGH” and “AEDFHR” is “ADH” of length 3.

4. Find X= BDCABA and Y= ABCBDAB. Find the length of LCS

Ans.

The length of LCS is 4

BCAB,BCBA,BDAB

ALGO:

LCS(m,n)

{ 1 + L(m-1,n-1) if A[m] = B[n] // if first character match then take it

max(LCS(m-1,n), LCS(m,n-1)) else

}

BDCABA & **ABCBDAB** : First character mismatch

then **two cases** :

1) BDCABA & **BCBDAB** : we hv removed Ist char from second string..

Here **Ist character match**,

LCS = **1**+ LCS(DCABA & CBDAB)

Now **D & C mismatch** again take two(can apply algo)..

Here to do quickly we observe manually that longest possible now is 3 characters(we can use algo also..) - So, i get CAB ,CBA,DAB

which when combined with B gives BCAB,BCBA,BDAB

2) DCABA & ABCBDAB : Here we hv removed first B from Ist sequence

Now, D & A first characters again mismatch.

two parts again:

2a) CABA & **ABCBDAB** : no common subsequence of length 4 possible here

2b) DCABA & **BCBDAB** : similarly here too - no possibility for a subsequence of length 4

5. Find the LCS for input Sequences "abcdghijklem" and "bcdehjklsnmd"

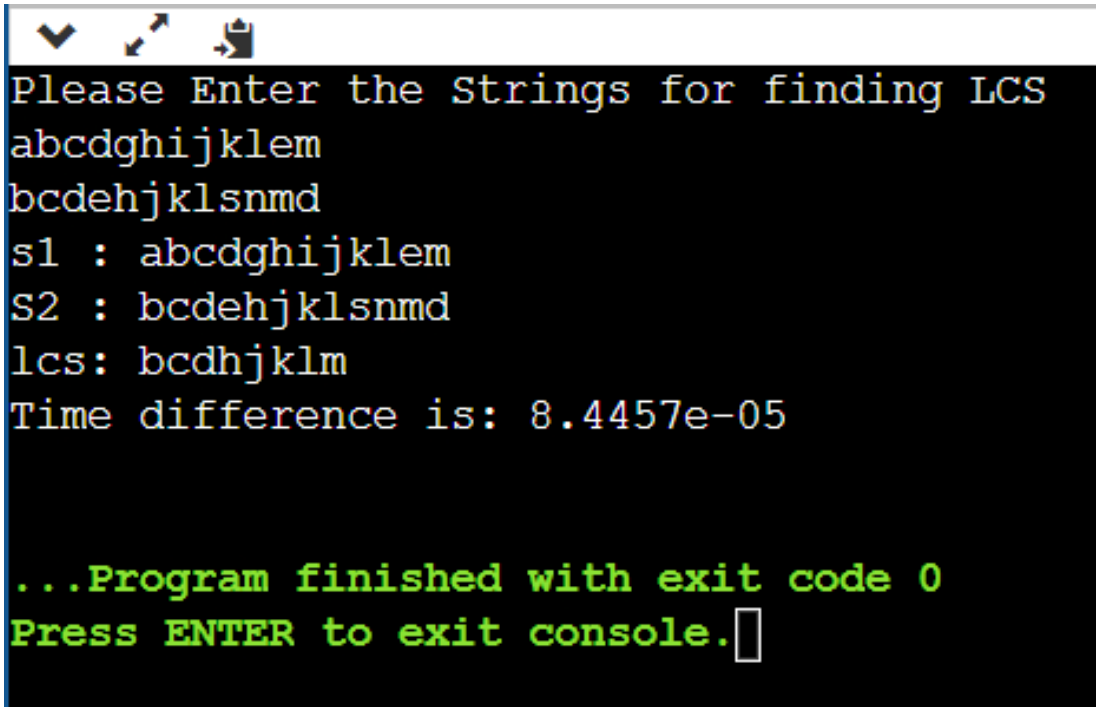
Ans.

s1 : abcdghijklem

S2 : bcdehjklsnmd

lcs: bcdhjklm

Time difference is: 8.4457e-05



Matrix Chain Multiplication:

Given a sequence of matrices, find the most efficient way to multiply these matrices together. The problem is not actually to perform the multiplications, but merely to decide in which order to perform the multiplications.

We have many options to multiply a chain of matrices because matrix multiplication is associative. In other words, no matter how we parenthesize the product, the result will be the same. For example, if we had four matrices A, B, C, and D, we would have:

(ABC)D = (AB)(CD) = A(BCD) = ...

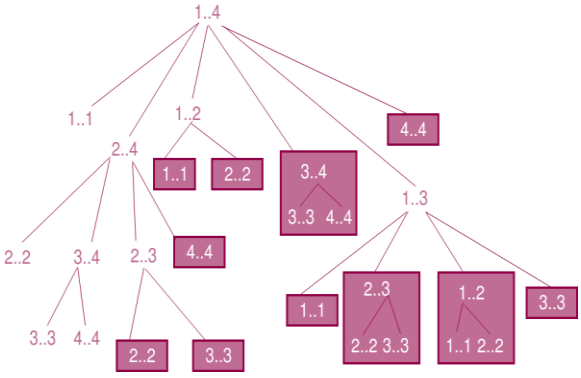
However, the order in which we parenthesize the product affects the number of simple arithmetic operations needed to compute the product, or the efficiency. For example, suppose A is a 10 × 30 matrix, B is a 30 × 5 matrix, and C is a 5 × 60 matrix. Then,

(AB)C = (10×30×5) + (10×5×60) = 1500 + 3000 = 4500 operations

A(BC) = (30×5×60) + (10×30×60) = 9000 + 18000 = 27000 operations.

Clearly the first parenthesization requires less number of operations.

Given an array p[] which represents the chain of matrices such that the ith matrix Ai is of dimension p[i-1] x p[i]. We need to write a function MatrixChainOrder() that should return the minimum number of multiplications needed to multiply the chain.



(AB)C = (10×30×5) + (10×5×60) = 1500 + 3000 = 4500 operations

A(BC) = (30×5×60) + (10×30×60) = 9000 + 18000 = 27000 operations.

Clearly the first parenthesization requires less number of operations.

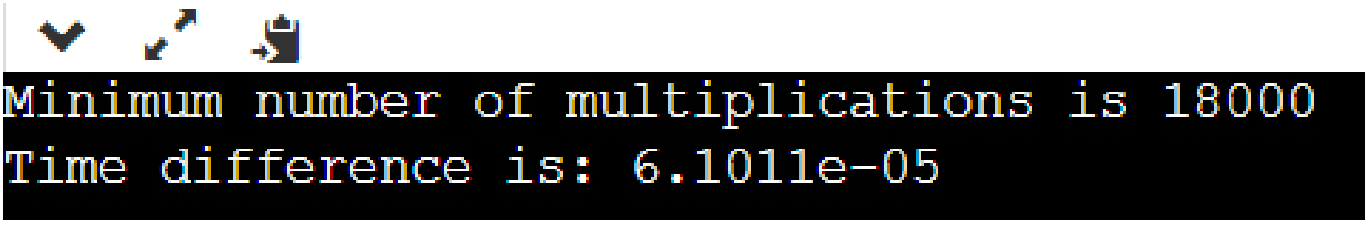
Given an array p[] which represents the chain of matrices such that the ith matrix Ai is of dimension p[i-1] x p[i]. We need to write a function MatrixChainOrder() that should return the minimum number of multiplications needed to multiply the chain.

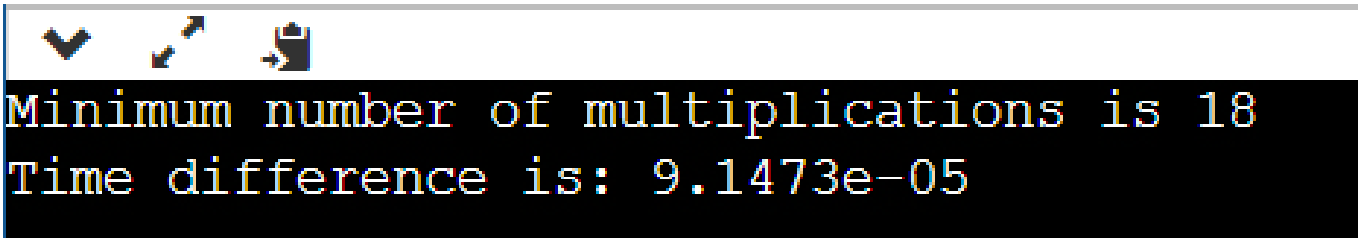
Source Code:

```
#include <bits/stdc++.h>
#include <chrono>

using namespace std;

int dp[100][100];

int matrixChainMemo(int *p, int i, int j){
    if (i == j)
        return 0;

    if (dp[i][j] != -1)
        return dp[i][j];

    dp[i][j] = INT_MAX;

    for (int k = i; k < j; k++) {
        dp[i][j] = min(dp[i][j], matrixChainMemo(p, i, k) +
matrixChainMemo(p, k + 1, j) + p[i - 1] * p[k] * p[j]);
    }
    return dp[i][j];
}

int MatrixChainOrder(int *p, int n){
    return matrixChainMemo(p, 1, n - 1);
}

int main(){
    int arr[] = {10, 20, 30, 40};
```

```
    int n = sizeof(arr) / sizeof(arr[0]);
    memset(dp, -1, sizeof dp);

    auto start = chrono::high_resolution_clock::now();

    ios_base::sync_with_stdio(false);

    cout << "Minimum number of multiplications is " << MatrixChainOrder(arr,
n) << endl;

    auto end = chrono::high_resolution_clock::now();

    double time_taken = chrono::duration_cast<chrono::nanoseconds>(end -
start).count();
    time_taken *= 1e-9;

    cout << "Time difference is: " << time_taken << setprecision(6) << endl;

    return 0;
}
```

Output:





Batch Analysis

Source Code

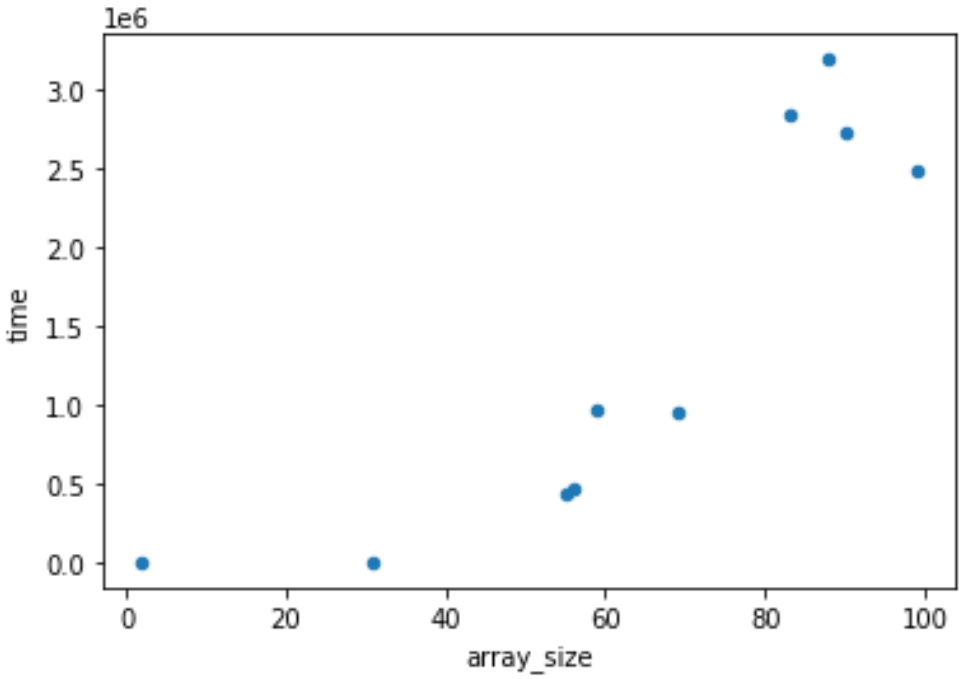
```
#include <bits/stdc++.h>
#include <chrono>

using namespace std;

int dp[100][100];

int matrixChainMemo(int *p, int i, int j){
    if (i == j)
        return 0;

    if (dp[i][j] != -1)
        return dp[i][j];

    dp[i][j] = INT_MAX;

    for (int k = i; k < j; k++) {
        dp[i][j] = min(
            dp[i][j], matrixChainMemo(p, i, k) + matrixChainMemo(p, k + 1, j) +
p[i - 1] * p[k] * p[j]);
    }
    return dp[i][j];
}

int MatrixChainOrder(int *p, int n){
    int i = 1, j = n - 1;
    return matrixChainMemo(p, i, j);
}

void printArray(double arr[], int size)
{
    for (int i = 0; i < size; i++)
        cout << arr[i] << ", ";
    cout << endl;
}

void printArray(int arr[], int size)
{
    for (int i = 0; i < size; i++)
        cout << arr[i] << ", ";
    cout << endl;
```

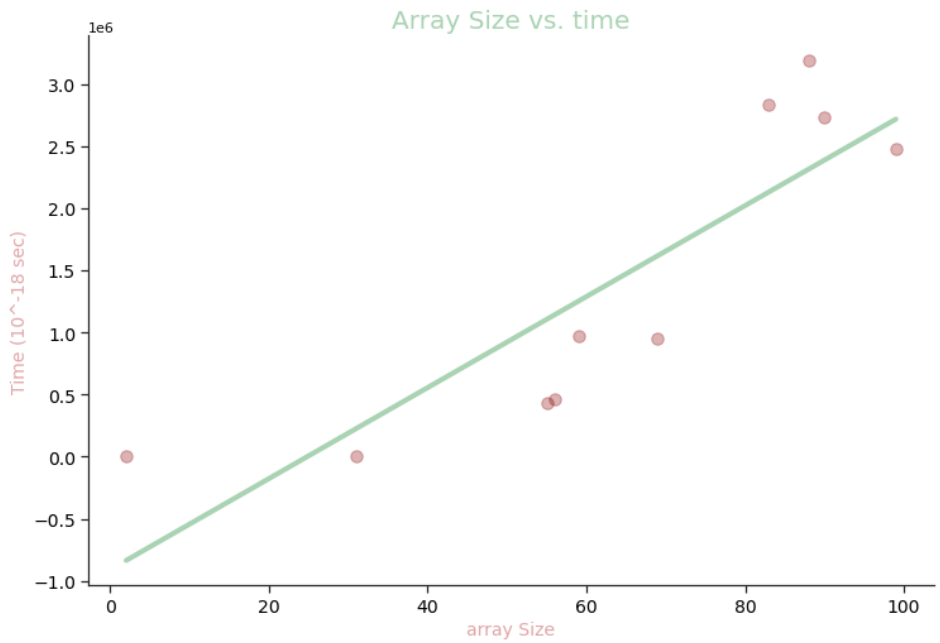
```
}

void printArray(float arr[], int size)
{
    for (int i = 0; i < size; i++)
        cout << arr[i] << ", ";
    cout << endl;
}

int main(){
    double times[10];
    int ns[10];
    for (int x = 0; x < 10; x++)    {
        memset(dp, -1, sizeof dp);
        int n = rand() % 100;
        ns[x] = n;

        int arr[n];
        for (int i = 0; i < n; i++) {
            arr[i] = rand() % 100;
        }

        auto start = chrono::high_resolution_clock::now();
        ios_base::sync_with_stdio(false);

        MatrixChainOrder(arr, n);

        auto end = chrono::high_resolution_clock::now();

        double time_taken = chrono::duration_cast<chrono::nanoseconds>(end -
start).count();

        times[x] = time_taken;
    }

    cout << "value of n's: " << endl;
    printArray(ns, 10);

    cout << "time for each n: " << endl;
    printArray(times, 10);

    return 0;
}
```

Output:

input

```
value of n's:
83, 88, 56, 69, 90, 59, 2, 99, 55, 31,
time for each n:
2.83879e+06, 3.19062e+06, 460596, 953485, 2.72973e+06, 974584, 235, 2.48403e+06, 432541, 76248,
```





Viva Questions

1. What is the recurrence relation for MCM problem?

Ans.

B(i, j) = B(i, k) + B(k + 1, j) + p[i - 1] * p[k] * p[j]

2. Let A1, A2, A3,A4, A5 be five matrices of dimensions 2x3, 3x5,5x2,2x4,4x3 respectively. Find the minimum number of scalar multiplications required to find the product A1 A2 A3 A4 A5 using the basic matrix multiplication method?

Ans.

78

3. Consider the two matrices P and Q which are 10 x 20 and 20 x 30 matrices respectively. What is the number of multiplications required to multiply the two matrices?

Ans.

The number of multiplications required is 10*20*30.

4. Consider the matrices P, Q and R which are 10 x 20, 20 x 30 and 30 x 40 matrices respectively. What is the minimum number of multiplications required to multiply the three matrices?

Ans.

The minimum number of multiplications are 18000. This is the case when the matrices are parenthesized as (P*Q)*R.

5. Can this problem be solved by greedy approach. Justify?

Ans.

Unfortunately, **there is no good "greedy choice"** for Matrix Chain Multiplication, meaning that for any choice that's easy to compute, there is always some input sequence of matrices for which your greedy algorithm will not find the optimum parenthesization.

At each step only least value is selected among all elements of in the array p[x, y], so that the multiplication cost is kept minimum at each step. This greedy approach ensures that the solution is optimal with least cost involved and the output is a fully parenthesized product of matrices.

Optimal Binary Search

an **optimal binary search tree (OBST)**, sometimes called a **weight-balanced binary tree**, is a binary search tree which provides the smallest possible search time for a given sequence of accesses (or access probabilities).

Optimal BSTs are generally divided into two types:

- Static
- Dynamic

In the **static optimality** problem, the tree cannot be modified after it has been constructed. In this case, there exists some particular layout of the nodes of the tree which provides the smallest expected search time for the given access probabilities. Various algorithms exist to construct or approximate the statically optimal tree given the information on the access probabilities of the elements.

- OBST is one special kind of advanced tree.
- It focus on how to reduce the cost of the search of the BST.
- It may not have the lowest height !
- It needs 3 tables to record probabilities, cost, and root.
- It has n keys (representation $k_1,k_2,\ldots,k_n$) in sorted order (so that $k_1<k_2<\ldots<k_n$), and we wish to build a binary search tree from these keys. For each k_i ,we have a probability p_i that a search will be for k_i.
- In contrast of, some searches may be for values not in k_i, and so we also have n+1 "dummy keys" $d_0,d_1,\ldots,d_n$ representing not in k_i.
- In particular, d_0 represents all values less than k_1, and d_n represents all values greater than k_n, and for i=1,2,…,n-1, the dummy key d_i represents all values between k_i and k_{i+1}.

∗The dummy keys are leaves (external nodes), and the data keys mean internal nodes.

Step1:The structure of an OBST

To characterize the optimal substructure of OBST, we start with an observation about sub trees. Consider any sub tree of a BST. It must contain keys in a contiguous range ki,…,kj, for some 1≦i ≦j ≦n. In addition, a sub tree that contains keys ki,…,kj must also have as its leaves the dummy keys d_{i-1} ,…, d_j

- We need to use the optimal substructure to show that we can construct an optimal solution to the problem from optimal solutions to sub problems. Given keys k_i ,…, k_j, one of these keys, say k_r (I ≦r ≦j), will be the root of an optimal sub tree containing these keys. The left sub tree of the root k_r will contain the keys (k_i ,…, k_{r-1}) and the dummy keys(d_{i-1} ,…, d_{r-1}), and the right sub tree will contain the keys (k_{r+1} ,…, k_j) and the dummy keys(d_r ,…, d_j). As long as we examine all candidate roots k_r, wherel ≦r ≦j, and we determine all optimal binary search trees containing k_i ,…, k_{r-1} and those containing k_{r+1} ,…, k_j , we are guaranteed that we will find an OBST.
- There is one detail worth nothing about "empty" sub trees. Suppose that in a sub tree with keys k_i,...,k_j, we select k_i as the root. By the above argument, k_i 's left sub tree contains the keys k_i,…, k_{i-1}. It is natural to interpret this sequence as containing no keys. It is easy to know that sub trees also contain dummy keys. The sequence has no actual keys but does contain the single dummy key d_{i-1}. Symmetrically, if we select k_j as the root, then k_j's right sub tree contains the keys, k_{j+1} …,k_j; this right sub tree contains no actual keys, but it does contain the dummy key d_j.

Step2: A recursive solution

- We need to define the value of an optimal solution recursively. We pick our sub problem domain as finding an OBST containing the keys k_i,…,k_j, where i≧1, j ≦n, and j ≧ i-1. (It is when j=i-1 that there are no actual keys; we have just the dummy key d_{i-1}.)
- Let us define e [i,j] as the expected cost of searching an OBST containing the keys k_i,…, k_j. Ultimately, we wish to compute e[1,n].
- The easy case occurs when j=i-1. Then we have just the dummy key d_{i-1}. The expected search cost is e [i,i-1]= q_{i-1}.
- When j≧1, we need to select a root k_r from among k_i,…,k_j and then make an OBST with keys k_i,…,k_{r-1} its left sub tree and an OBST with keys k_{r+1},…,k_j its right sub tree. By the time, what happens to the expected search cost of a sub tree when it becomes a sub tree of a node? The answer is that the depth of each node in the sub tree increases by 1.

- By the second statement, the excepted search cost of this sub tree increases by the sum of all the probabilities in the sub tree. For a sub tree with keys $k_i,\ldots,k_j$ let us denote this sum of probabilities as

$w(i,j) = (l=i\sim j)\sum p_l + (l=i-1\sim j)\sum q_l$

Thus, if k_r is the root of an optimal sub tree containing keys $k_i,\ldots,k_j$, we have

$E[i,j]= p_r + (e[i,r-1]+w(i,r-1))+(e[r+1,j]+w(r+1,j))$

Nothing that $w(i,j) = w(i,r-1)+ p_r +w(r+1,j)$

- We rewrite e[i,j] as

  $e[i,j]= e[i,r-1] + e[r+1,j]+w(i,j)$

The recursive equation as above assumes that we know which node k_r to use as the root. We choose the root that gives the lowest expected search cost, giving us our final recursive formulation:

E[i,j]=

case1: if $i \leqq j, i \leqq r \leqq j$

$E[i,j]=\min\{e[i,r-1]+e[r+1,j]+w(i,j)\}$

case2: if $j=i-1$; $E[i,j]= q_{i-1}$

- The e[i,j] values give the expected search costs in OBST. To help us keep track of the structure of OBST, we define root[i,j], for $1 \leqq i \leqq j \leqq n$, to be the index r for which k_r is the root of an OBST containing keys $k_i,\ldots,k_j$.

Step3: Computing the expected search cost of an OBST

- We store the e[i.j] values in a table e[1..n+1, 0..n]. The first index needs to run to n+1rather than n because in order to have a sub tree containing only the dummy key d_n, we will need to compute and store e[n+1,n]. The second index needs to start from 0 because in order to have a sub tree containing only the dummy key d_0, we will need to compute and store e[1,0]. We will use only the entries e[i,j] for which $j \geqq i-1$. we also use a table root[i,j], for recording the root of the sub tree containing keys $k_i,\ldots, k_j$. This table uses only the entries for which $1 \leqq i \leqq j \leqq n$.

- We will need one other table for efficiency. Rather than compute the value of w(i,j) from scratch every time we are computing e[i,j] . We keep these values in a table w[1..n+1,0..n]. For the base case, we compute $w[i,i-1] = q_{i-1}$ for $1 \leqq i \leqq n$.

- For $j \geqq l$, we compute :

  $Ww[i,j]=w[i,j-1]+p_{i+}q_i$

Pseudo code:

OPTIMAL—BST(p,q,n)

For i 1 to n+1

do e[i,i-1] <- q_{i-1}

do w[i,i-1] <- q_{i-1}

For l 1 to n

do for i <- 1 to n-l +1

do j <- i+l-1

e[i,j] <- ∞

w[i,j] <- w[i,j-1]+p_j+q_j

For r <- i to j

do t <- e[i,r-1]+e[r+1,j]+w[i,j]

if t<e[i,j]

then e[i,j] <- t

root [i,j] <- r

Return e and root

Sample Example:

key	*A*	*B*	*C*	*D*
probability	0.1	0.2	0.4	0.3

The initial tables look like this:

main table

	0	1	2	3	4
1	0	0.1			
2		0	0.2		
3			0	0.4	
4				0	0.3
5					0

root table

	0	1	2	3	4
1		1			
2			2		
3				3	
4					4
5					

Let us compute $C(1, 2)$:

$$C(1,2) = \min \left\{ \begin{array}{ll} k=1: & C(1,0)+C(2,2)+\sum_{s=1}^{2} p_s = 0+0.2+0.3 = 0.5 \\ k=2: & C(1,1)+C(3,2)+\sum_{s=1}^{2} p_s = 0.1+0+0.3 = 0.4 \end{array} \right\}$$
$$= 0.4.$$

Thus, out of two possible binary trees containing the first two keys, ***A*** and ***B***, the root of the optimal tree has index 2 (i.e., it contains ***B),*** and the average number of comparisons in a successful search in this tree is 0.4.

We will arrive at the following final tables using the recurrence equations:

main table

	0	1	2	3	4
1	0	0.1	0.4	1.1	1.7
2		0	0.2	0.8	1.4
3			0	0.4	1.0
4				0	0.3
5					0

root table

	0	1	2	3	4
1		1	2	3	3
2			2	3	3
3				3	3
4					4
5					

Thus, the average number of key comparisons in the optimal tree is equal to 1.7. Since ***R(***1, 4***)*** = 3, the root of the optimal tree contains the third key, i.e., ***C.*** Its left sub tree is made up of keys ***A*** and ***B***, and its right sub tree contains just key ***D***. To find the specific structure of these sub trees, we find first their roots by consulting the root table again. Since ***R(***1, 2***)*** = 2, the root of the optimal tree containing ***A*** and ***B*** is ***B,*** with ***A*** being its left child (and the root of the one-node tree: ***R(***1, 1***)*** = 1***)***. Since ***R (***4, 4***)*** = 4, the root of this one-node optimal tree is its only key ***D.***

Result and Analysis:

Every time we work on an entry e[i, j] with j − i = k, we know that all the entries e[i 0 , j 0] with j 0 − i 0 < k have already been computed. Note that the recursive formula we use to compute e[i, j] only involves entries e[i 0 , j 0] with j 0 − i 0 < k. So they are all ready, and we can compute e[i, j] in time O(j − i). It is easy to check that the total running time is Θ (n 3).

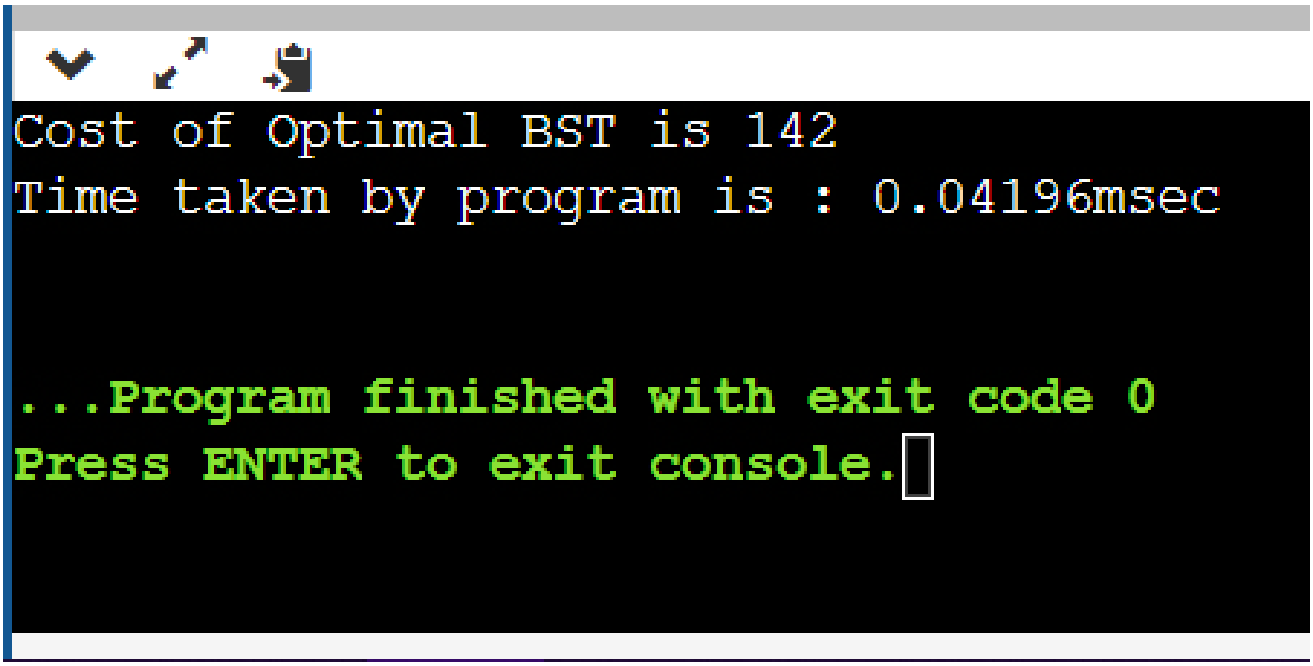
Source Code:

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

int sum(int freq[], int i, int j);

int optimalSearchTree(int keys[], int freq[], int n){
```

```
    int cost[n][n];

    // For a single key, cost is equal to frequency of the key
    for (int i = 0; i < n; i++)
        cost[i][i] = freq[i];

    // Now we need to consider chains of length 2, 3, ... L is chain length.
    for (int L = 2; L <= n; L++) {

        // i is row number in cost[][]
        for (int i = 0; i <= n - L + 1; i++) {

            // Get column number j from row number i and chain length L
            int j = i + L - 1;
            cost[i][j] = INT_MAX;

            // Try making all keys in interval keys[i..j] as root
            for (int r = i; r <= j; r++) {
                // c = cost when keys[r] becomes root of this subtree
                int c = ((r > i) ? cost[i][r - 1] : 0) + ((r < j) ? cost[r +
1][j] : 0) + sum(freq, i, j);

                if (c < cost[i][j])
                    cost[i][j] = c;
            }
        }
    }
    return cost[0][n - 1];
}

int sum(int freq[], int i, int j){
    int s = 0;
    for (int k = i; k <= j; k++)
        s += freq[k];

    return s;
}

int main(){

    srand((unsigned)time(0));

    auto start = chrono::high_resolution_clock::now();
    ios_base::sync_with_stdio(false);
```

```
    int keys[] = {10, 12, 20};
    int freq[] = {34, 8, 50};

    int n = sizeof(keys) / sizeof(keys[0]);

    cout << "Cost of Optimal BST is " << optimalSearchTree(keys, freq, n);

    auto end = chrono::high_resolution_clock::now();
    double time_taken = chrono::duration_cast<chrono::nanoseconds>(end -
start).count();
    time_taken *= 1e-9 * 1000;

    cout << "\nTime taken by program is : " << time_taken << setprecision(6);
    cout << "msec" << endl;

    return 0;
}
```

Output:



Viva Questions

1. What is OBST?

Ans.

As we know that in binary search tree, the nodes in the left subtree have lesser value than the root node and the nodes in the right subtree have greater value than the root node. The frequency and key-value determine the overall cost of searching a node.

2. Give any application of OBST.

Ans.

A binary search tree is one of the most important data structures in computer science. One of its principal applications is to implement a dictionary, a set of elements with the operations of searching, insertion, and deletion

3. Can OBST be solved using Greedy Approach?

Ans.

Yes, we try to optimize the cost.

4. What is balanced BST?

Ans.

A balanced binary tree, also referred to as a height-balanced binary tree, is defined as a binary tree in which the height of the left and right subtree of any node differ by not more than 1.

To check if a tree is height-balanced, get the height of left and right subtrees. Return true if difference between heights is not more than 1 and left and right subtrees are balanced, otherwise return false.

5. What is the best case for OBST?

Ans.

In best case, The binary search tree is a balanced binary search tree. Height of the binary search tree becomes log(n). So, Time complexity of BST Operations **= O(logn)**.

Binomial Coefficient

A binomial coefficient C(n, k) also gives the number of ways, disregarding order, that k objects can be chosen from among n objects more formally, the number of k-element subsets (or k-combinations) of a n-element set.

1) Optimal Substructure
The value of C(n, k) can be recursively calculated using the following standard formula for Binomial Coefficients.

C(n, k) = C(n-1, k-1) + C(n-1, k)

C(n, 0) = C(n, n) = 1

2) Overlapping Subproblems
It should be noted that the above function computes the same subproblems again and again.

$$\binom{i}{n} = \frac{i(i-1)(i-2)\cdots(i-(n-1))}{n(n-1)(n-2)(n-3)\cdots 2\cdot 1} = \frac{1}{n!}\prod_{k=0}^{n-1}(i-k) = \left(\prod_{k=1}^{n}\frac{1}{k}\right)\prod_{k=0}^{n-1}(i-k)$$

$$\left(\prod_{k=1}^{n}\frac{1}{k}\right)\prod_{k=0}^{n-1}(i-k) = \left(\prod_{k=1}^{n}\frac{1}{k}\right)\prod_{k=1}^{n}(i-(k-1)) = \prod_{k=1}^{n}\frac{i+1-k}{k} \Rightarrow \binom{i}{n} = \prod_{k=1}^{n}\frac{i-k+1}{k}$$

1) **Binomial Coefficients**

In probability and statistics applications, you often need to know the total possible number of certain outcome combinations. For example, you may want to know how many ways a 2-card BlackJack hand can be dealt from a 52 card deck, or you may need to know the number of possible committees of 3 people that can be formed from a 12-person department, etc. The *binomial coefficient* (often referred to as "*n* choose *k*") will provide the number of combinations of *k* things that can be formed from a set of *n* things. The binomial coefficient is written mathematically as:

$$\binom{n}{k}$$

which we refer to as "*n* choose *k*".
Binomial coefficients can be defined recursively:

$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k} \qquad \binom{n}{0} = \binom{n}{n} = 1$$

Individually, write a *recursive* function named `choose(int n,int k)` that will compute and return the value of the binomial coefficient. Then compare your function to your partner's, and together (i) come up with a function implementation you both agree on, and (ii) write it as a C++ function on the computer.

Instructions:

Compared to a brute force recursive algorithm that could run exponential, the dynamic programming algorithm runs typically in quadratic time. The recursive algorithm ran in exponential time while the iterative algorithm ran in linear time. The space cost does increase, which is typically the size of the table. Frequently, the whole table does not have to store.

Computing a Binomial Coefficient

Computing binomial coefficients is non-optimization problem but can be solved using dynamic programming.

Binomial coefficients are represented by $C(n, k)$ or $\binom{n}{k}$ and can be used to represent the coefficients of a binomial:

$(a + b)^n = C(n, 0)a^n + \ldots + C(n, k)a^{n-k}b^k + \ldots + C(n, n)b^n$

The recursive relation is defined by the prior power

$C(n, k) = C(n\text{-}1, k\text{-}1) + C(n\text{-}1, k)$ for $n > k > 0$

IC $C(n, 0) = C(n, n) = 1$

Dynamic algorithm constructs a nxk table, with the first column and diagonal filled out using the IC.

Construct the table:

		k					
		0	1	2	...	*k*-1	*k*
	0	1					
	1	1	1				
	2	1	2	1			
n	.						
	.						
	.						
	k	1					1
	.						
	.						

.			
n-1	1	*C*(*n*-1, *k*-1)	
n	1		*C*(*n*, *k*)

The table is then filled out iteratively, row by row using the recursive relation.

Pseudo code:

Binomial(n, k)

for i ← 0 to n do // fill out the table row wise

for i = 0 to min(i, k) do

if j==0 or j==i then C[i, j] ← 1 // IC

else C[i, j] ← C[i-1, j-1] + C[i-1, j] // recursive relation

return C[n, k]

Sample Example:

Ex: C(3,2) Answer should be 3.

i=0

j= 0 to 0

j=0 :C[0,0]=1

i=1

j= 0 to 1

j=0 :C[2,0]=1

j=1: C[1,1]=1

i=2

j= 0 to 2

j=0 :C[2,0]=1

j=1: C[2,1]=C[1,0]+ C[1,1] =1+1=2

j=1: C[2,1]=1

i=3

j= 0 to 2 (min(i,k) => k=2,i=3)

j=0 : C[3,0]=1

j=1: C[3,1]= C[2,0]+ C[2,1]= 1+2=3

j=2: C[3,2]= C[2,1]+ C[2,2]= 2+1=3

return C[3,2] =>3

Result and Analysis:

The cost of the algorithm is filing out the table. Addition is the basic operation. Because $k \leq n$, the sum needs to be split into two parts because only the half the table needs to be filled out for $i < k$ and remaining part of the table is filled out across the entire row.

$A(n, k)$ = sum for upper triangle + sum for the lower rectangle

$= \sum_{i=1}^{k} \sum_{j=1}^{i-1} 1 + \sum_{i=1}^{n} \sum_{j=1}^{k} 1$

$= \sum_{i=1}^{k} (i-1) + \sum_{i=1}^{n} k$

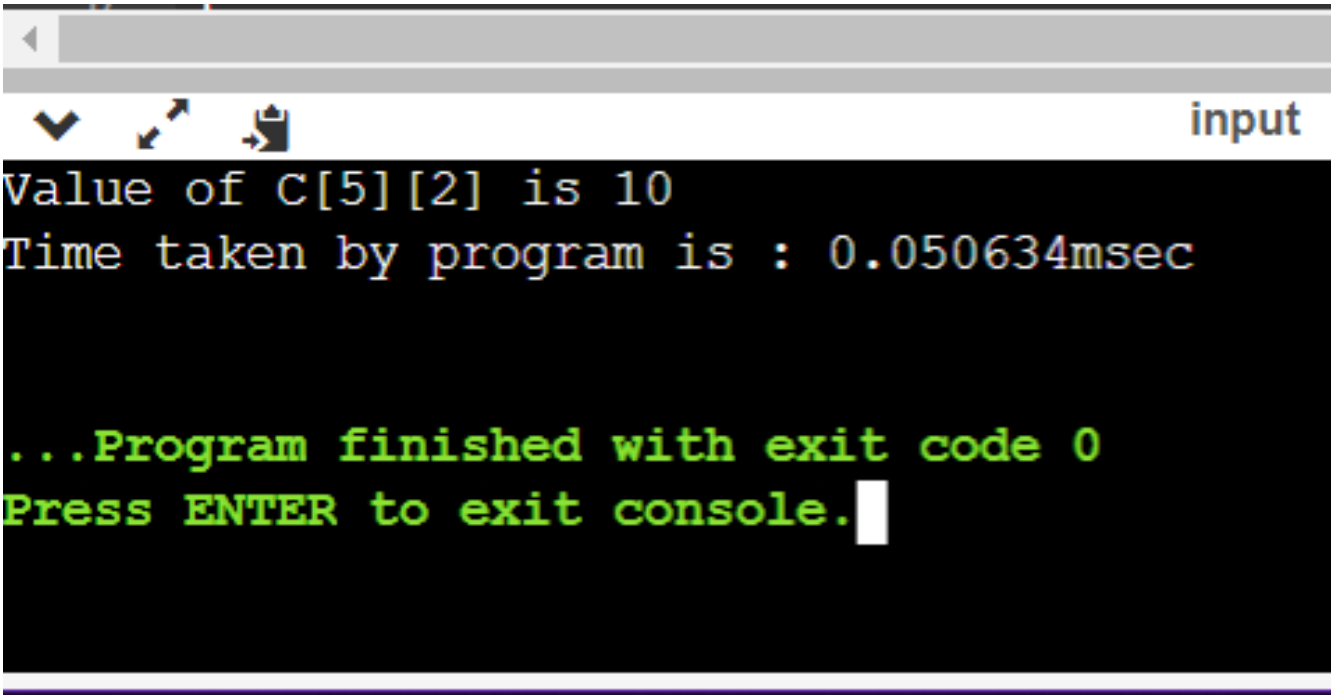
$= (k-1)k/2 + k(n-k) \in \Theta(nk)$

Source Code:

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

int min(int a, int b);

int binomialCoeff(int n, int k){

    int C[n + 1][k + 1];
```

```
    int i, j;

    // Calculate value of Binomial Coefficient in bottom up manner
    for (i = 0; i <= n; i++) {

        for (j = 0; j <= min(i, k); j++) {

            // Base Cases
            if (j == 0 || j == i)
                C[i][j] = 1;

            // Calculate value using previously stored values
            else
                C[i][j] = C[i - 1][j - 1] + C[i - 1][j];
        }
    }
    return C[n][k];
}

// A utility function to return minimum of two integers
int min(int a, int b) {
    return (a < b) ? a : b;
}

int main(){

    srand((unsigned)time(0));

    auto start = chrono::high_resolution_clock::now();
    // unsync the I/O of C and C++.
    ios_base::sync_with_stdio(false);

    int n = 5, k = 2;

    cout << "Value of C[" << n << "][" << k << "] is " << binomialCoeff(n, k);

    auto end = chrono::high_resolution_clock::now();
    double time_taken = chrono::duration_cast<chrono::nanoseconds>(end -
start).count();
    time_taken *= 1e-9 * 1000;
    cout << "\nTime taken by program is : " << time_taken << setprecision(6);
    cout << "msec" << endl;
    return 0;
}
```

Output:



Viva Questions

1. What is Time Complexity of Binomial Coefficient?

Ans.

O(N^2 + Q), because we are precomputing the binomial coefficients up to **nCn**. This operation takes **O(N^2)** time and then **O(1)** time to answer each query.

2. What is Space Complexity of Binomial Coefficient?

Ans.

O(N^2), for storing the precomputed results of binomial coeffcients.

3. What is Binomial Coefficient?

Ans.

Binomial Coefficient is used heavily to solve combinatorics problems. Let's say you have some n different elements and you need to pick k elements. So, if you want to solve this problem you can easily write all the cases of choosing k elements out of n elements.

4. Why is Binomial Coefficient required?

Ans.

This problem can be easily solved using binomial coefficient. More than that, this problem of choosing k elements out of n different elements is one of the way to define binomial coefficient **n C k.** Binomial coefficient can be easily calculated using the given formula:

$$\binom{n}{k} = \frac{n!}{k!(n-k)!}.$$

Since now we are good at the basics, we should find ways to calculate this efficiently.

5. What is Naive Approach for finding Binomial Coefficient?

Ans.

This approach isn't too **naive** at all. Consider you are asked to find the number of ways of choosing 3 elements out of 5 elements. So you can easily find **n!, k!** and **(n-k)!** and put the values in the given formula. This solution takes only **O(N) time** and **O(1) space**. But sometimes your factorial values may overflow so we need to take care of that. This approach is fine if we want to calculate a single binomial coefficient. But many times we need to calculate many binomial coefficients. So, it's better to have them precomputed. We will find out how to find the binomial coefficients efficiently.

EXPERIMENT - 5

Algorithms Design and Analysis Lab

Aim

To implement Algorithms using Greedy Approach and analyse its time complexity.

Syeda Reeha Quasar
14114802719
4C7

EXPERIMENT – 5

Aim:

To implement Algorithms using Greedy Approach and analyse its time complexity.

Theory:

A greedy algorithm is any algorithm that follows the problem-solving heuristic of making the locally optimal choice at each stage.[1] In many problems, a greedy strategy does not produce an optimal solution, but a greedy heuristic can yield locally optimal solutions that approximate a globally optimal solution in a reasonable amount of time.

To solve a problem based on the greedy approach, there are two stages

1. Scanning the list of items
2. Optimization

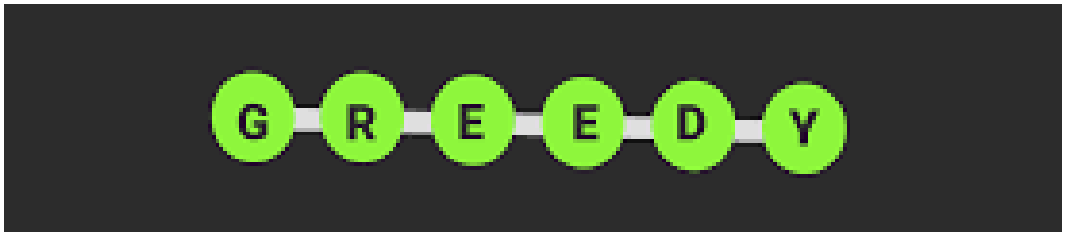
These stages are covered parallelly in this Greedy algorithm tutorial, on course of division of the array.

To understand the greedy approach, you will need to have a working knowledge of recursion and context switching. This helps you to understand how to trace the code. You can define the greedy paradigm in terms of your own necessary and sufficient statements.

Two conditions define the greedy paradigm.

- Each stepwise solution must structure a problem towards its best-accepted solution.
- It is sufficient if the structuring of the problem can halt in a finite number of greedy steps.

With the theorizing continued, let us describe the history associated with the Greedy search approach.



Knapsack problem

Instructions:

To solve 0-1 Knapsack, Dynamic Programming approach is required as Greedy approach gives an optimal solution for Fractional Knapsack. In 0-1 Knapsack, items cannot be broken which means the thief should take the item as a whole or should leave it. This is reason behind calling it as 0-1 Knapsack. Hence, in case of 0-1 Knapsack, the value of x_i can be either *0* or *1*

Problem Statement:

A thief is robbing a store and can carry a max$_i$mal weight of *W* into his knapsack. There are *n* items and weight of ith item is w_i and the profit of selecting this item is p_i. What items should the thief take?

Dynamic-Programming Approach:

Let *i* be the highest-numbered item in an optimal solution S for W dollars. Then $S' = S - \{i\}$ is an optimal solution for $W - w_i$ dollars and the value to the solution *S* is V_i plus the value of the sub-problem. We can express this fact in the following formula: define c[i, w] to be the solution for items 1,2, … , i and the max$_i$mum weight w.

The algorithm takes the following inputs

- The max$_i$mum weight W
- The number of items n
- The two sequences v = <v_1, v_2, …, v_n> and w = <w_1, w_2, …, w_n>
- Let *i* be the highest-numbered item in an optimal solution *S* for *W* pounds. Then *S`* = *S* - {*i*} is an optimal solution for *W* - w_i pounds and the value to the solution *S* is V_i plus the value of the sub problem.
- We can express this fact in the following formula: define *c*[*i*, *w*] to be the solution for items 1,2, . . . , *i* and maximum weight *w*. Then

$$c[i,w] = \begin{cases} 0 & \text{if } i = 0 \text{ or } w = 0 \\ c[i\text{-}1, w] & \text{if } w_i \geq 0 \\ \max [v_i + c[i\text{-}1, w\text{-}w_i], c[i\text{-}1, w]\} & \text{if } i>0 \text{ and } w \geq w_i \end{cases}$$

This says that the value of the solution to *i* items either include i^{th} item, in which case it is v_i plus a sub problem solution for (*i* - 1) items and the weight excluding w_i, or does not include i^{th} item, in which case it is a sub problem's solution for (*i* - 1) items and the same weight. That is, if the thief picks item *i*, thief takes v_i value, and thief can choose from items *w* - w_i, and get *c*[*i* -

1, $w - w_i$] additional value. On other hand, if thief decides not to take item *i*, thief can choose from item 1,2, . . . , *i*- 1 upto the weight limit *w*, and get *c*[*i* - 1, *w*] value. The better of these two choices should be made.

Pseudo code:

The first row of *c* is filled in from left to right, then the second row, and so on. At the end of the computation, *c*[*n*, *w*] contains the maximum value that can be picked into the knapsack.

Dynamic-0-1-knapsack (v, w, n, W)

```
for w = 0 to W do
  c [0, w] = 0
for i = 1 to n do
  c [i, 0] = 0
  for w = 1 to W do
    if wi ≤ w then
      if vi + c[i-1, w-wi] then
        c [i, w] = vi + c [i-1, w-wi]
      else c [i, w] = c[i-1, w]
    else
      c [i, w] = c[i-1, w]
```

The set of items to take can be deduced from the table, starting at c[n, w]and tracing backwards where the optimal values came from. If *c [i, w] = c[i-1, w]*, then item *i* is not part of the solution, and we continue tracing with c[i-1, w]. Otherwise, item *i* is part of the solution, and we continue tracing with c[i-1, w-W].

Sample Example:

Assume that we have a knapsack with max weight capacity W = 5 Our objective is to fill the knapsack with items such that the benefit (value or profit) is maximum.

Following table contains the items along with their value and weight.

Item	i	1	2	3	4
value	val	100	20	60	40
weight	wt.	3	2	4	1

Total items n = 4
Total capacity of the knapsack W = 5

Now we create a value table V [i,w] where, i denotes number of items and w denotes the weight of the items. Rows denote the items and columns denote the weight. As there are 4 items so, we have 5 rows from 0 to 4.
And the weight limit of the knapsack is W = 5 so, we have 6 columns from 0 to 5

After calculation, the value table V

V[i,w]	w = 0	1	2	3	4	5
i = 0	0	0	0	0	0	0
1	0	0	0	100	100	100
2	0	0	20	100	100	120
3	0	0	20	100	100	120
4	0	40	40	100	140	140

Maximum value earned
Max Value = V [n,W] = V[4,5] = 140

Result and Analysis:

This algorithm takes $\theta(n, w)$ times as table c has $(n + 1).(w + 1)$ entries, where each entry requires $\theta(1)$ time to compute.

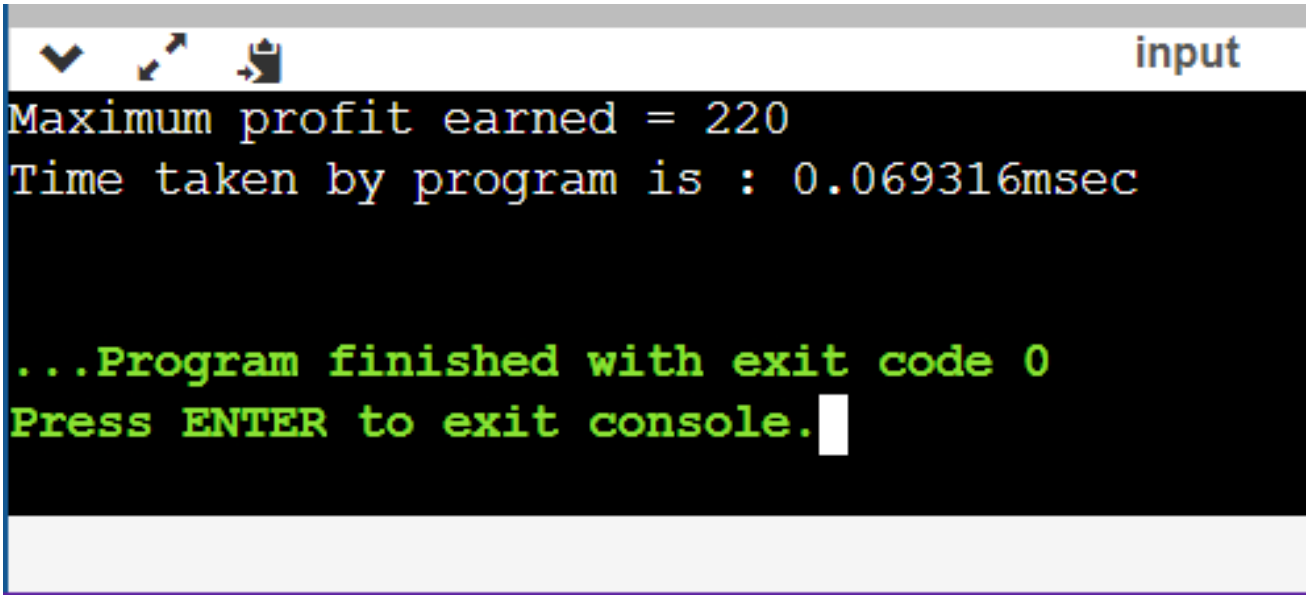
Source Code:

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

int knapSackRec(int W, int wt[], int val[], int i, int** dp)
{
    if (i < 0)
```

```
        return 0;
    if (dp[i][W] != -1)
        return dp[i][W];
    if (wt[i] > W) {

        dp[i][W] = knapSackRec(W, wt, val, i - 1, dp);
        return dp[i][W];
    }
    else {
        dp[i][W] = max(val[i] + knapSackRec(W - wt[i], wt, val, i - 1, dp),
knapSackRec(W, wt, val, i - 1, dp));
        return dp[i][W];
    }
}

int knapSack(int W, int wt[], int val[], int n)
{
    int** dp;
    dp = new int*[n];

    for (int i = 0; i < n; i++)
        dp[i] = new int[W + 1];

    for (int i = 0; i < n; i++)
        for (int j = 0; j < W + 1; j++)
            dp[i][j] = -1;
    return knapSackRec(W, wt, val, n - 1, dp);
}

int main()
{
    int val[] = { 60, 100, 120 };
    int wt[] = { 10, 20, 30 };
    int W = 50;
    int n = sizeof(val) / sizeof(val[0]);

    auto start = chrono::high_resolution_clock::now();
    ios_base::sync_with_stdio(false);

    cout << "Maximum profit earned = " << knapSack(W, wt, val, n);

    auto end = chrono::high_resolution_clock::now();
    double time_taken = chrono::duration_cast<chrono::nanoseconds>(end -
start).count();
```

```
    time_taken *= 1e-9*1000;

    cout << "\nTime taken by program is : " << time_taken << setprecision(6);
    cout << "msec" << endl;

    return 0;
}
```

Output:



Fractional Knapsack problem

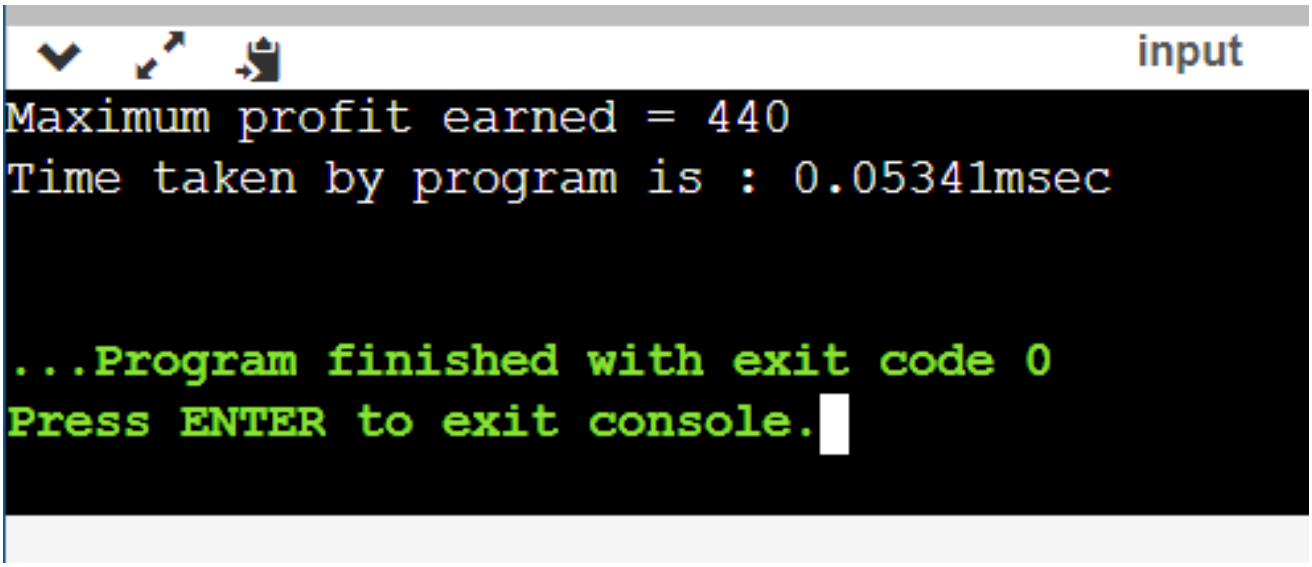
Source Code

```
#include <bits/stdc++.h>

using namespace std;

struct Item
{
    int value, weight;
```

```
    Item(int value, int weight) : value(value), weight(weight) {}
};

bool cmp(struct Item a, struct Item b)
{
    double r1 = (double)a.value / a.weight;
    double r2 = (double)b.value / b.weight;
    return r1 > r2;
}

double fractionalKnapsack(struct Item arr[], int N, int size)
{
    sort(arr, arr + size, cmp);
    int curWeight = 0;
    double finalvalue = 0.0;

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

        if (curWeight + arr[i].weight <= N)
        {
            curWeight += arr[i].weight;
            finalvalue += arr[i].value;
        }
        else
        {
            int remain = N - curWeight;
            finalvalue += arr[i].value * ((double)remain / arr[i].weight);

            break;
        }
    }
    return finalvalue;
}

int main()
{
    srand((unsigned)time(0));
    int N = 60;

    Item arr[] = {{100, 10}, {280, 40}, {120, 20}, {120, 24}};
    int size = sizeof(arr) / sizeof(arr[0]);

    auto start = chrono::high_resolution_clock::now();
    ios_base::sync_with_stdio(false);
```

```
    cout << "Maximum profit earned = " << fractionalKnapsack(arr, N, size);

    auto end = chrono::high_resolution_clock::now();
    double time_taken = chrono::duration_cast<chrono::nanoseconds>(end -
start).count();

    time_taken *= 1e-9 * 1000;

    cout << "\nTime taken by program is : " << time_taken << setprecision(6) <<
"msec" << endl;
    return 0;
}
```

Output



Viva Questions

1. Why is the greedy approach not suitable for 0-1 Knapsack Problem

Ans.

In 0-1 Knapsack, items cannot be broken which means the thief should take the item as a whole or should leave it. This is reason behind calling it as 0-1 Knapsack.

Hence, in case of 0-1 Knapsack, the value of x_i can be either ***0*** or ***1***, where other constraints remain the same.

0-1 Knapsack cannot be solved by Greedy approach. Greedy approach does not ensure an optimal solution. In many instances, Greedy approach may give an optimal solution.

2. What is the difference between fractional and 0-1 Knapsack problem

Ans.

0–1 Knapsack problem

In the 0–1 Knapsack problem, we are not allowed to break items. We either take the whole item or don't take it.

```
1. Input:
2.   Items as (value, weight) pairs
3.   arr[] = {{60, 10}, {100, 20}, {120, 30}}
4.   Knapsack Capacity, W = 50;
5. Output:
6.   Maximum possible value = 220
7.   by taking items of weight 20 and 30 kg
```

Fractional Knapsack

In Fractional Knapsack, we can break items for maximizing the total value of knapsack. This problem in which we can break item also called fractional knapsack problem.

```
1. Input :
2.    Same as above
3. Output :
```

```
4.    Maximum possible value = 240
5.    By taking full items of 10 kg, 20 kg and
6.    2/3rd of last item of 30 kg
```

3. What is memorization?

Ans.

In computing, memoization or memoisation is an optimization technique used primarily to speed up computer programs by storing the results of expensive function calls and returning the cached result when the same inputs occur again.

Memory in your C++ program is divided into two parts –

- The stack − All variables declared inside the function will take up memory from the stack.
- The heap − This is unused memory of the program and can be used to allocate the memory dynamically when program runs.

4. Find out maximum profit for fractional knapsack with maximum allowable weight =14 and n=5 (p1,p2,p3,p4,p5)={70,25,15,29,38}; (w1,w2,w3,w4,w5)={10,1,2,2,6}

Ans.

Maximum profit earned = 177 Time taken by program is : 0.056174msec

5. Solve the following instance of the 0/1 knapsack problem using dynamic programming

Weight 1 2 3 2

Profit 10 15 25 12

Ans.

Maximum profit earned = 7.8

Time taken by program is : 0.064123msec

Activity Selection

To implement Activity Selection Problem.

Problem Statement:

You are given n activities with their start and finish times. Select the maximum number of activities that can be performed by a single person, assuming that a person can only work on a single activity at a time.

Example 1 : Consider the following 3 activities sorted by

by finish time.

start[] = {10, 12, 20};

finish[] = {20, 25, 30};

A person can perform at most **two** activities. The

maximum set of activities that can be executed

is {0, 2} [These are indexes in start[] and

finish[]]

Example 2 : Consider the following 6 activities

sorted by by finish time.

start[] = {1, 3, 0, 5, 8, 5};

finish[] = {2, 4, 6, 7, 9, 9};

A person can perform at most **four** activities. The

maximum set of activities that can be executed

is {0, 1, 3, 4} [These are indexes in start[] and

finish[]]

The greedy choice is to always pick the next activity whose finish time is least among the remaining activities and the start time is more than or equal to the finish time of the previously selected activity. We can sort the activities according to their finishing time so that we always consider the next activity as minimum finishing time activity.

1) Sort the activities according to their finishing time
2) Select the first activity from the sorted array and print it.
3) Do the following for the remaining activities in the sorted array.

.....a) If the start time of this activity is greater than or equal to the finish time of the previously selected activity then select this activity and print it.

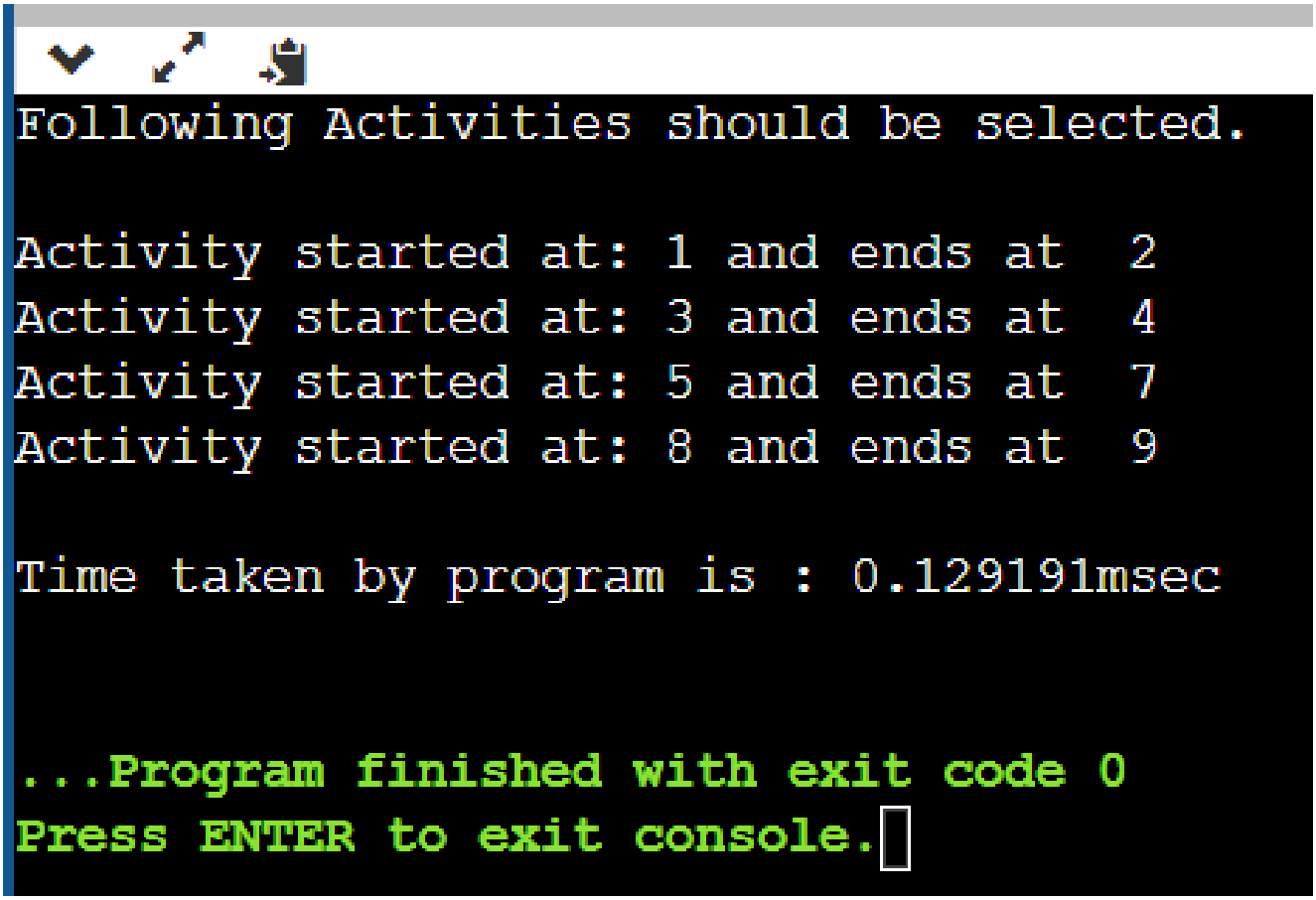
Source Code

```
#include <bits/stdc++.h>

using namespace std;


void SelectActivities(vector<int>s,vector<int>f){

// Vector to store results.

    vector<pair<int,int>>ans;


// Minimum Priority Queue to sort activities in ascending order of finishing time
(f[i]).


    priority_queue<pair<int,int>,vector<pair<int,int>>,greater<pair<int,int>>>p;


    for(int i=0;i<s.size();i++){

        // Pushing elements in priority queue where the key is f[i]

        p.push(make_pair(f[i],s[i]));

    }


    auto it = p.top();

    int start = it.second;

    int end = it.first;

    p.pop();

    ans.push_back(make_pair(start,end));


    while(!p.empty()){

        auto itr = p.top();

        p.pop();

        if(itr.second >= end){

            start = itr.second;
```

```
            end = itr.first;
            ans.push_back(make_pair(start,end));
        }
    }
    cout << "Following Activities should be selected. " << endl << endl;

    for(auto itr=ans.begin();itr!=ans.end();itr++){
        cout << "Activity started at: " << (*itr).first << " and ends at  " <<
(*itr).second << endl;
    }
}

int main()
{
    vector<int>s = {1, 3, 0, 5, 8, 5};
    vector<int>f = {2, 4, 6, 7, 9, 9};

    auto start = chrono::high_resolution_clock::now();
    ios_base::sync_with_stdio(false);

    SelectActivities(s,f);

    auto end = chrono::high_resolution_clock::now();
    double time_taken = chrono::duration_cast<chrono::nanoseconds>(end -
start).count();

    time_taken *= 1e-9*1000;

    cout << "\nTime taken by program is : " << time_taken << setprecision(6);
    cout << "msec" << endl;
```

```
    return 0;
}
```

Output



Huffman Encoding

To implement Huffman Coding and analyze its time complexity.

Huffman coding is a lossless data compression algorithm. The idea is to assign variable-length codes to input characters, lengths of the assigned codes are based on the frequencies of corresponding characters. The most frequent character gets the smallest code and the least frequent character gets the largest code.
The variable-length codes assigned to input characters are Prefix Codes, means the codes (bit sequences) are assigned in such a way that the code assigned to one character is not the prefix of code assigned to any other character. This is how Huffman Coding makes sure that there is no ambiguity when decoding the generated bitstream.
Let us understand prefix codes with a counter example. Let there be four characters a, b, c and d, and their corresponding variable length codes be 00, 01, 0 and 1. This coding leads to ambiguity because code assigned to c is the prefix of codes assigned to a and b. If the compressed bit stream is 0001, the de-compressed output may be "cccd" or "ccb" or "acd" or "ab".
See this for applications of Huffman Coding.
There are mainly two major parts in Huffman Coding

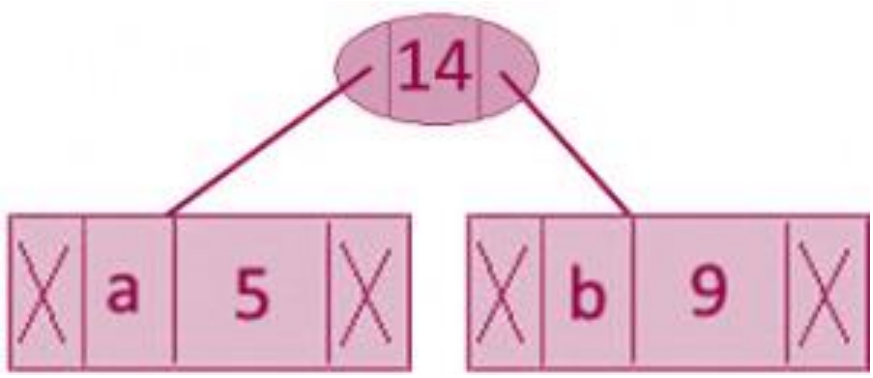
1. Build a Huffman Tree from input characters.
2. Traverse the Huffman Tree and assign codes to characters.

Steps to build Huffman Tree

Input is an array of unique characters along with their frequency of occurrences and output is Huffman Tree.

1. Create a leaf node for each unique character and build a min heap of all leaf nodes (Min Heap is used as a priority queue. The value of frequency field is used to compare two nodes in min heap. Initially, the least frequent character is at root)
2. Extract two nodes with the minimum frequency from the min heap.
3. Create a new internal node with a frequency equal to the sum of the two nodes frequencies. Make the first extracted node as its left child and the other extracted node as its right child. Add this node to the min heap.
4. Repeat steps#2 and #3 until the heap contains only one node. The remaining node is the root node and the tree is complete.
   Let us understand the algorithm with an example:

character	Frequency
a	5
b	9
c	12
d	13
e	16
f	45

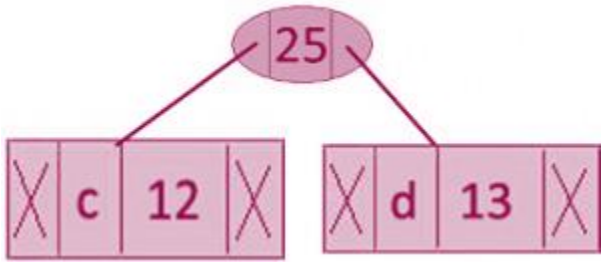


Step 1. Build a min heap that contains 6 nodes where each node represents root of a tree with single node.
Step 2 Extract two minimum frequency nodes from min heap. Add a new internal node with frequency 5 + 9 = 14.

Now min heap contains 5 nodes where 4 nodes are roots of trees with single element each, and one heap node is root of tree with 3 elements

character	Frequency	Internal Node	14
c	12	e	16
d	13	f	45

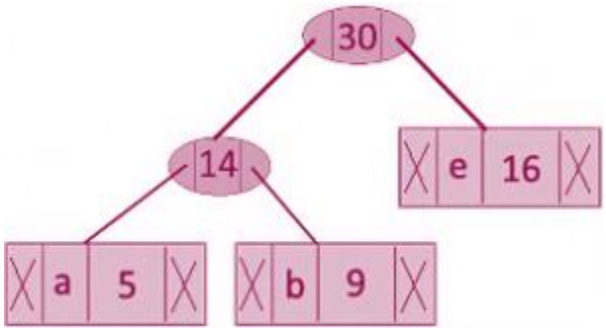
Step 3: Extract two minimum frequency nodes from heap. Add a new internal node with frequency 12 + 13 = 25



Now min heap contains 4 nodes where 2 nodes are roots of trees with single element each, and two heap nodes are root of tree with more than one nodes

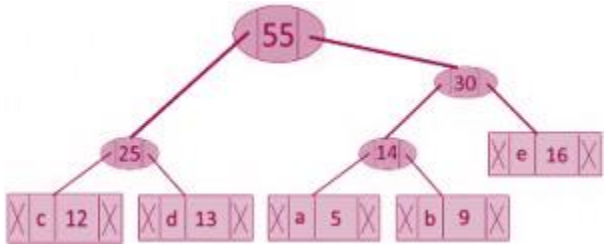
character	Frequency
Internal Node	14
E	16
Internal Node	25
f	45

Step 4: Extract two minimum frequency nodes. Add a new internal node with frequency 14 + 16 = 30



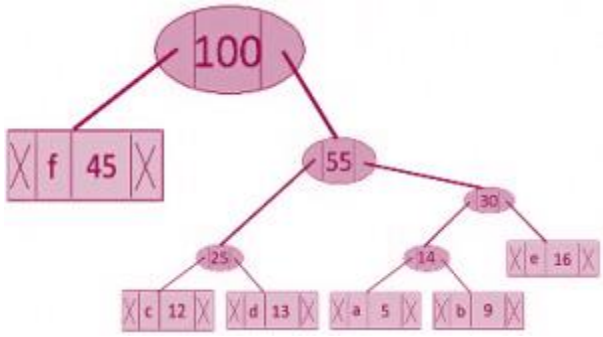
Now min heap contains 3 nodes.

Step 5: Extract two minimum frequency nodes. Add a new internal node with frequency 25 + 30 = 55



Now min heap contains 2 nodes.

Step 6: Extract two minimum frequency nodes. Add a new internal node with frequency 45 + 55 = 100



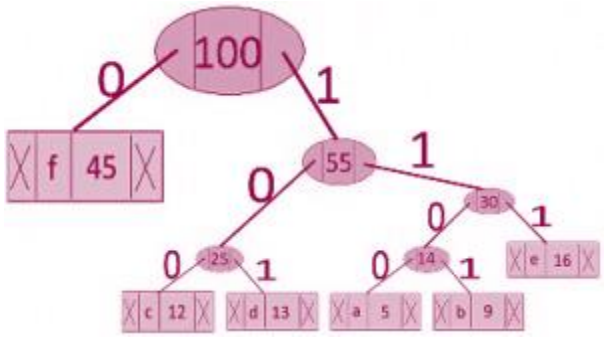
Now min heap contains only one node.

Since the heap contains only one node, the algorithm stops here.

Steps to print codes from Huffman Tree:

Traverse the tree formed starting from the root. Maintain an auxiliary array. While moving to the left child, write 0 to the array. While moving to the right child, write 1 to

the array. Print the array when a leaf node is encountered.



character	code-word
f	0
c	100
d	101
a	1100
b	1101
e	111

Source Code

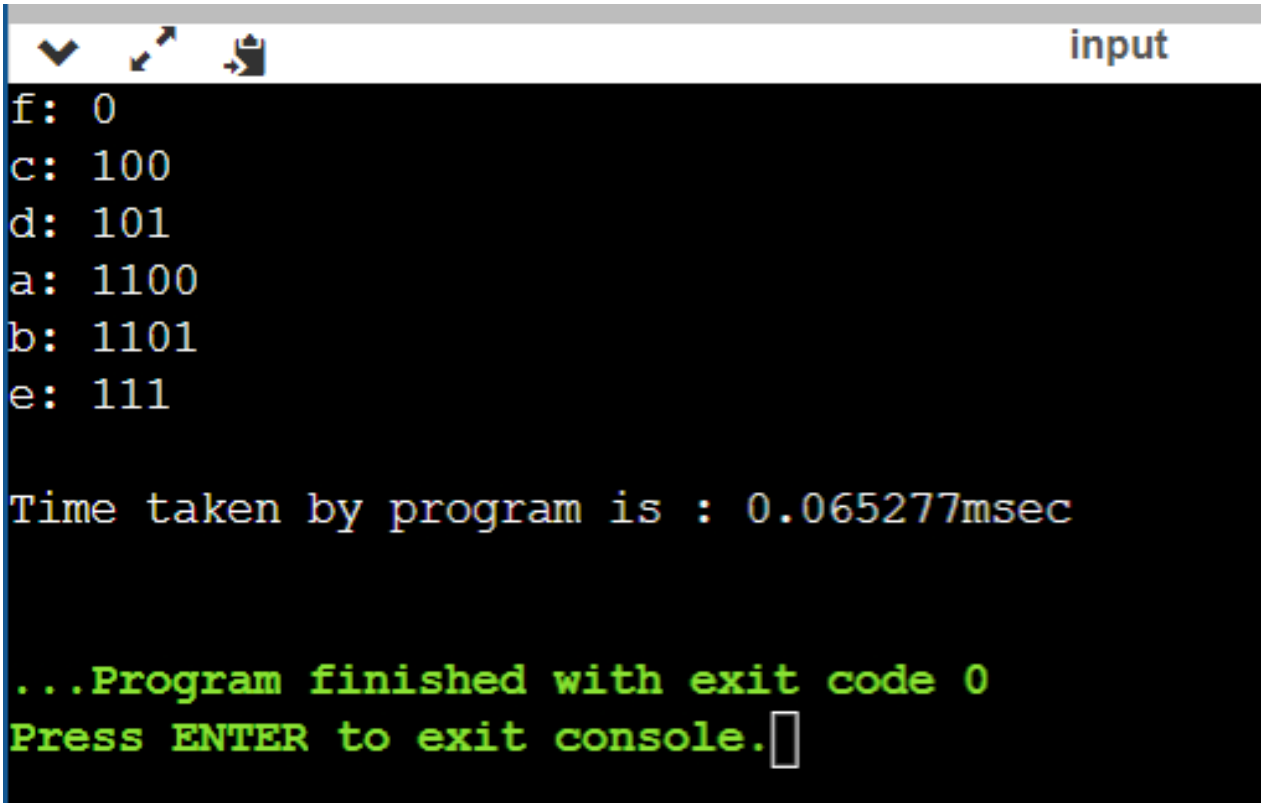
```
#include <bits/stdc++.h>
using namespace std;

struct MinHeapNode{
	char data;
	unsigned freq;
	MinHeapNode *left, *right;
	MinHeapNode(char data, unsigned freq){
		left = right = NULL;
		this->data = data;
```

```
            this->freq = freq;
      }
};

struct compare{
      bool operator()(MinHeapNode* l, MinHeapNode* r){
            return (l->freq > r->freq);
      }
};

void printCodes(struct MinHeapNode* root, string str){
    if (!root)
            return;

      if (root->data != '#')
            cout << root->data << ": " << str << "\n";

      printCodes(root->left, str + "0");
      printCodes(root->right, str + "1");
}

void HuffmanCodes(char data[], int freq[], int size){
      struct MinHeapNode *left, *right, *top;

      priority_queue<MinHeapNode*, vector<MinHeapNode*>, compare> minHeap;

      for (int i = 0; i < size; ++i)
            minHeap.push(new MinHeapNode(data[i], freq[i]));

      while (minHeap.size() != 1) {
```

```
            left = minHeap.top();
            minHeap.pop();

            right = minHeap.top();
            minHeap.pop();

            top = new MinHeapNode('#', left->freq + right->freq);

            top->left = left;
            top->right = right;

            minHeap.push(top);
      }

      printCodes(minHeap.top(), "");
}

int main(){

      char arr[] = { 'a', 'b', 'c', 'd', 'e', 'f' };
      int freq[] = { 5, 9, 12, 13, 16, 45 };

      int size = sizeof(arr) / sizeof(arr[0]);

    auto start = chrono::high_resolution_clock::now();
    ios_base::sync_with_stdio(false);

      HuffmanCodes(arr, freq, size);

    auto end = chrono::high_resolution_clock::now();
```

```
    double time_taken = chrono::duration_cast<chrono::nanoseconds>(end -
start).count();


    time_taken *= 1e-9*1000;


    cout << "\nTime taken by program is : " << time_taken << setprecision(6);
    cout << "msec" << endl;


      return 0;
}
```

Output



Task Scheduling Problem

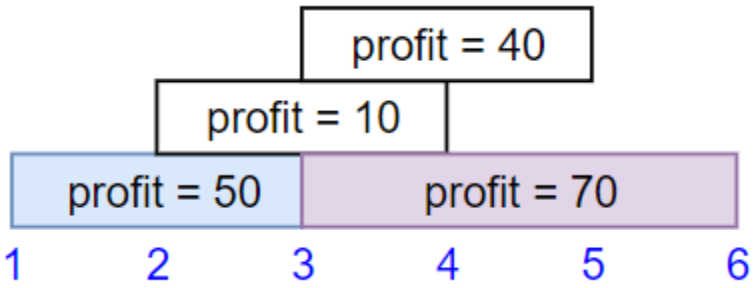
To implement Task Scheduling Problem.

We have n jobs, where every job is scheduled to be done
from startTime[i] to endTime[i], obtaining a profit of profit[i].

You're given the startTime, endTime and profit arrays, return the maximum profit you can take such that there are no two jobs in the subset with overlapping time range.

If you choose a job that ends at time X you will be able to start another job that starts at time X.

Example 1:



Input: startTime = [1,2,3,3], endTime = [3,4,5,6], profit = [50,10,40,70]

Output: 120

Explanation: The subset chosen is the first and fourth job.

Time range [1-3]+[3-6] , we get profit of 120 = 50 + 70.

Source Code

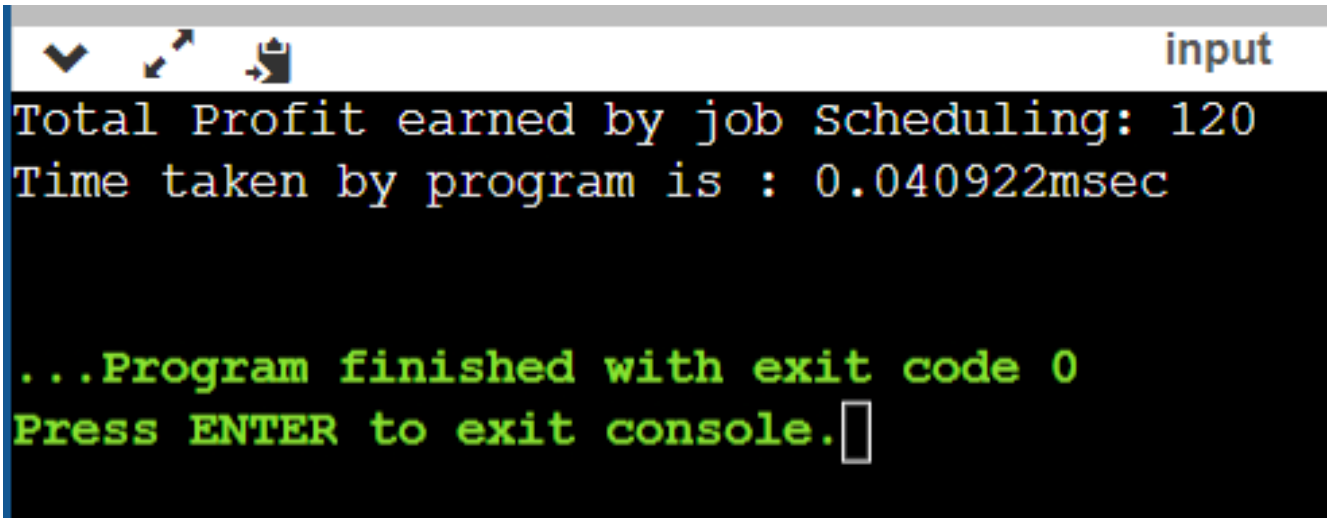
```
#include <bits/stdc++.h>

using namespace std;


int jobScheduling(vector<int> &startTime, vector<int> &endTime, vector<int>
&profit)

{

    int n = startTime.size(), i = 0, ans = 0;
```

```
    vector<pair<int, int>> v;

    for (int i = 0; i < n; i++)
    {
        v.push_back({startTime[i], i});
    }
    sort(v.begin(), v.end());
    vector<int> dp(n, 0);

    for (i = n - 1; i >= 0; i--)
    {
        int f = v[i].first;
        int ii = v[i].second;
        int val = 0;

        val += profit[ii];

        int x = endTime[ii];

        auto it = lower_bound(v.begin(), v.end(), x, [](const pair<int, int> &p,
const int &value)
                              { return p.first < value; }); // searching for next
Starttime which is greater than the current endtime

        if (it != v.end())
        {
            int j = it - v.begin();

            val += dp[j];
        }
```

```
        if (i == n - 1)
            dp[i] = max(dp[i], val);

        else
        {
            dp[i] = max(dp[i], max(val, dp[i + 1])); // either take the ith job or skip it and take the next one
        }

        ans = max(ans, dp[i]);
    }

    return ans;
}

int main()
{
    vector<int> startTime{1, 2, 3, 3};
    vector<int> endTime{3, 4, 5, 6};
    vector<int> profit{50, 10, 40, 70};

    auto start = chrono::high_resolution_clock::now();
    ios_base::sync_with_stdio(false);

    cout << "Total Profit earned by job Scheduling: " << jobScheduling(startTime, endTime, profit);

            auto end = chrono::high_resolution_clock::now();
    double time_taken = chrono::duration_cast<chrono::nanoseconds>(end - start).count();

    time_taken *= 1e-9 * 1000;
```

```
    cout << "\nTime taken by program is : " << time_taken << setprecision(6);
    cout << "msec" << endl;

    return 0;
}
```

Output



Viva Questions

1. What is a Greedy Algorithm?

Ans.

We call algorithms *greedy* when they utilise the greedy property. The greedy property is:

At that exact moment in time, what is the optimal choice to make?

Greedy algorithms are *greedy*. They do not look into the future to decide the global optimal solution. They are only concerned with the optimal solution *locally*. This means that **the overall optimal solution may differ from the solution the greedy algorithm chooses.**

They never look backwards at what they've done to see if they could optimise globally. This is the main difference between Greedy Algorithms and Dynamic Programming.

2. What Are Greedy Algorithms Used For?

Ans.
Greedy algorithms are quick. A lot faster than the two other alternatives (Divide & Conquer, and Dynamic Programming). They're used because they're fast. Sometimes, Greedy algorithms give the global optimal solution every time. Some of these algorithms are:

- Dijkstra's Algorithm
- Kruskal's algorithm
- Prim's algorithm
- Huffman trees

These algorithms are Greedy, and their Greedy solution gives the optimal solution.

3. What are the various criteria used to improve the effectiveness of the algorithm?

Ans.

Input- Zero or more quantities are externally provided.

Output- At least one quantity is composed

Definiteness- Each instruction is simple and unambiguous

Finiteness- If we trace out the instructions of an algorithm, then for all step the algorithm complete after a finite number of steps

Effectiveness- Every instruction must be elementary.

4. List the advantage of Huffman's encoding?

Ans.

Huffman's encoding is one of the essential file compression techniques.

1. It is easy
2. It is flexibility
3. It implements optimal and minimum length encoding

5. Write the difference between the Dynamic programming and Greedy method.

Ans.

Dynamic programming

1. Many numbers of decisions are generated.
2. It gives an optimal solution always

Greedy method

1. Only one sequence of decision is generated.
2. It does not require to provide an optimal solution always.

EXPERIMENT - 6

Algorithms Design and Analysis Lab

Aim

To implement Dijkstra's Algorithm and analyse its time complexity.

Syeda Reeha Quasar
14114802719
4C7

EXPERIMENT – 6

Aim:

To implement Dijkstra's Algorithm and analyse its time complexity.

Theory:

Dijkstra's algorithm solves the single-source shortest-path problem when all edges have non-negative weights. It is a greedy algorithm and similar to Prim's algorithm. Algorithm starts at the source vertex, s, it grows a tree, T, that ultimately spans all vertices reachable from S. Vertices are added to T in order of distance i.e., first S, then the vertex closest to S, then the next closest, and so on.

Following implementation assumes that graph G is represented by adjacency lists.

DIJKSTRA (G, w, s)

INITIALIZE SINGLE-SOURCE (G, s)

S ← { } // S will ultimately contains vertices of final shortest-path weights from s

Initialize priority queue Q i.e., Q ← V[G]

while priority queue Q is not empty do

u ← EXTRACT_MIN(Q) // Pull out new vertex

S ← S È {u} // Perform relaxation for each vertex v adjacent to u

for each vertex v in Adj[u] do

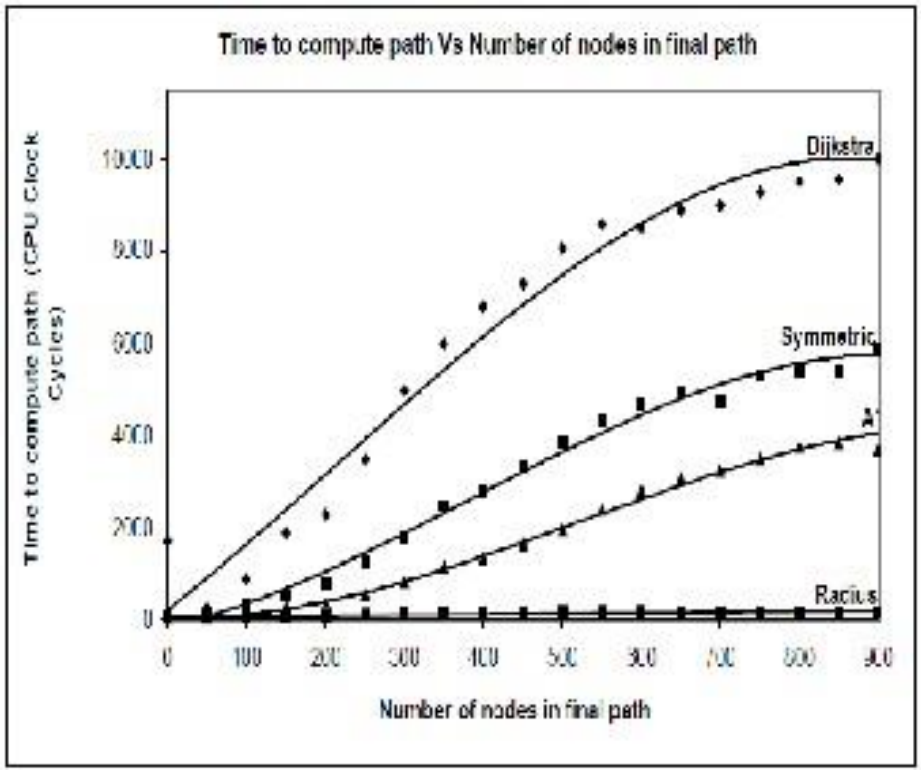
Relax (u, v, w)

Implementation Steps:

- Enter a graph using adjacency list.
- Enter the cost of each edge.
- Enter the source vertex
- Find shortest distances for all vertices reachable from source.

Result and Analysis:

Like Prim's algorithm, Dijkstra's algorithm runs in O(|E|lg|V|) time.



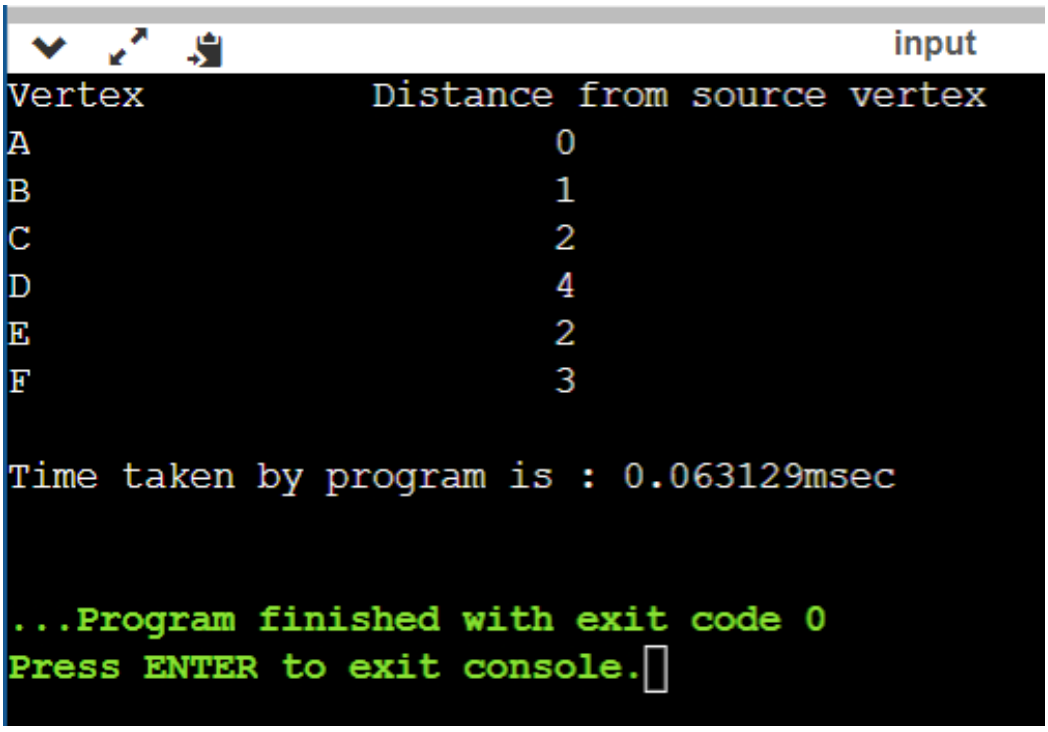
Source Code:

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

int miniDist(int distance[], bool Tset[]) // finding minimum distance
{
    int minimum = INT_MAX, ind;

    for (int k = 0; k < 6; k++)
    {
        if (Tset[k] == false && distance[k] <= minimum)
        {
            minimum = distance[k];
            ind = k;
```

```
        }
    }
    return ind;
}

void DijkstraAlgo(int graph[6][6], int src) // adjacency matrix
{
    int distance[6]; // // array to calculate the minimum distance for each node
    bool Tset[6];    // boolean array to mark visited and unvisited for each node

    for (int k = 0; k < 6; k++)
    {
        distance[k] = INT_MAX;
        Tset[k] = false;
    }

    distance[src] = 0; // Source vertex distance is set 0

    for (int k = 0; k < 6; k++)
    {
        int m = miniDist(distance, Tset);
        Tset[m] = true;
        for (int k = 0; k < 6; k++)
        {
            // updating the distance of neighbouring vertex
            if (!Tset[k] && graph[m][k] && distance[m] != INT_MAX && distance[m]
+ graph[m][k] < distance[k])
                distance[k] = distance[m] + graph[m][k];
        }
    }
    cout << "Vertex\t\tDistance from source vertex" << endl;
    for (int k = 0; k < 6; k++)
    {
        char str = 65 + k;
        cout << str << "\t\t\t" << distance[k] << endl;
    }
}

int main()
{
    int graph[6][6] = {
        {0, 1, 2, 0, 0, 0},
        {1, 0, 0, 5, 1, 0},
        {2, 0, 0, 2, 3, 0},
        {0, 5, 2, 0, 2, 2},
```

```
        {0, 1, 3, 2, 0, 1},
        {0, 0, 0, 2, 1, 0}};

    auto start = chrono::high_resolution_clock::now();
    ios_base::sync_with_stdio(false);

    DijkstraAlgo(graph, 0);

    auto end = chrono::high_resolution_clock::now();
    double time_taken = chrono::duration_cast<chrono::nanoseconds>(end -
start).count();

    time_taken *= 1e-9 * 1000;

    cout << "\nTime taken by program is : " << time_taken << setprecision(6);
    cout << "msec" << endl;

    return 0;
}
```

Output:



Viva Questions

1. What is the use of Dijkstra's algorithm?

Ans.

Dijkstra's procedure is used to solve the single-source shortest-paths method: for a given vertex called the source in a weighted linked graph, find the shortest path to all its other vertices. The single-source shortest-paths process asks for a family of paths, each leading from the source to various vertex in the graph, though some direction may have edges in common.

2. Give some applications of Dijkstra Algoritm.

Ans.

1. It is most widely used in finding shortest possible distance and show directions between 2 geographical locations such as in Google Maps.
2. This is also widely used in routing of data in networking and telecommunication domains for minimizing the delay occurred for transmission.
3. Wherever you encounter the need for shortest path solutions be it in robotics, transportation, embedded systems, factory or production plants to detect faults, etc this algorithm is used.

3. What is the most commonly used data structure for implementing Dijkstra's Algorithm?

Ans.

Minimum priority queue is the most commonly used data structure for implementing Dijkstra's Algorithm because the required operations to be performed in Dijkstra's Algorithm match with specialty of a minimum priority queue.

4. What is the time complexity of Dijikstra's algorithm?

Ans.

Time complexity of Dijkstra's algorithm is $O(N^2)$ because of the use of doubly nested for loops. It depends on how the table is manipulated.

5. The maximum number of times the decrease key operation performed in Dijkstra's algorithm will be equal to _____

Ans.

If the total number of edges in all adjacency list is E, then there will be a total of E number of iterations, hence there will be a total of at most E decrease key operations.

EXPERIMENT - 7

Algorithms Design and Analysis Lab

Aim

To implement minimum spanning trees algorithm and analyse its time complexity.

Syeda Reeha Quasar
14114802719
4C7

EXPERIMENT – 7

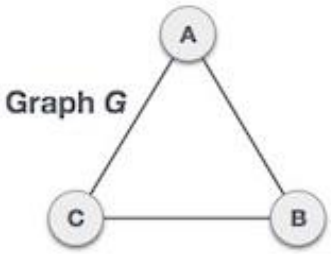
Aim:

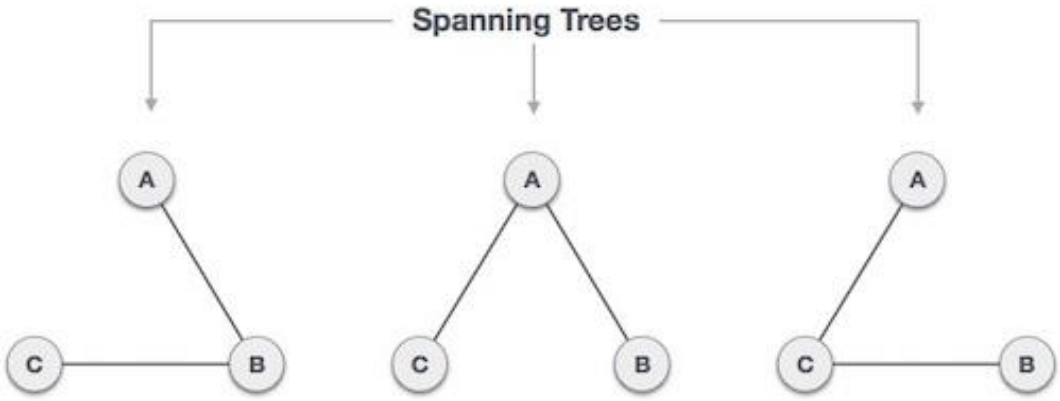
To implement minimum spanning trees algorithm and analyse its time complexity.

Theory:

A spanning tree is a subset of Graph G, which has all the vertices covered with minimum possible number of edges. Hence, a spanning tree does not have cycles and it cannot be disconnected.

By this definition, we can draw a conclusion that every connected and undirected Graph G has at least one spanning tree. A disconnected graph does not have any spanning tree, as it cannot be spanned to all its vertices.





We found three spanning trees off one complete graph. A complete undirected graph can have maximum $\mathbf{n^{n-2}}$ number of spanning trees, where **n** is the number of nodes. In the above addressed example, **n is 3**, hence $\mathbf{3^{3-2} = 3}$ spanning trees are possible.

General Properties of Spanning Tree

We now understand that one graph can have more than one spanning tree. Following are a few properties of the spanning tree connected to graph G −

- A connected graph G can have more than one spanning tree.
- All possible spanning trees of graph G, have the same number of edges and vertices.
- The spanning tree does not have any cycle (loops).
- Removing one edge from the spanning tree will make the graph disconnected, i.e. the spanning tree is **minimally connected**.
- Adding one edge to the spanning tree will create a circuit or loop, i.e. the spanning tree is **maximally acyclic**.

Mathematical Properties of Spanning Tree

- Spanning tree has **n-1** edges, where **n** is the number of nodes (vertices).
- From a complete graph, by removing maximum **e - n + 1** edges, we can construct a spanning tree.
- A complete graph can have maximum $\mathbf{n^{n-2}}$ number of spanning trees.

Thus, we can conclude that spanning trees are a subset of connected Graph G and disconnected graphs do not have spanning tree.

Application of Spanning Tree

Spanning tree is basically used to find a minimum path to connect all nodes in a graph. Common application of spanning trees are −

- **Civil Network Planning**
- **Computer Network Routing Protocol**
- **Cluster Analysis**

Let us understand this through a small example. Consider, city network as a huge graph and now plans to deploy telephone lines in such a way that in minimum lines we can connect to all city nodes. This is where the spanning tree comes into picture.

Minimum Spanning Tree (MST)

In a weighted graph, a minimum spanning tree is a spanning tree that has minimum weight than all other spanning trees of the same graph. In real-world situations, this weight can be measured as distance, congestion, traffic load or any arbitrary value denoted to the edges.

Minimum Spanning-Tree Algorithm

We shall learn about two most important spanning tree algorithms here −

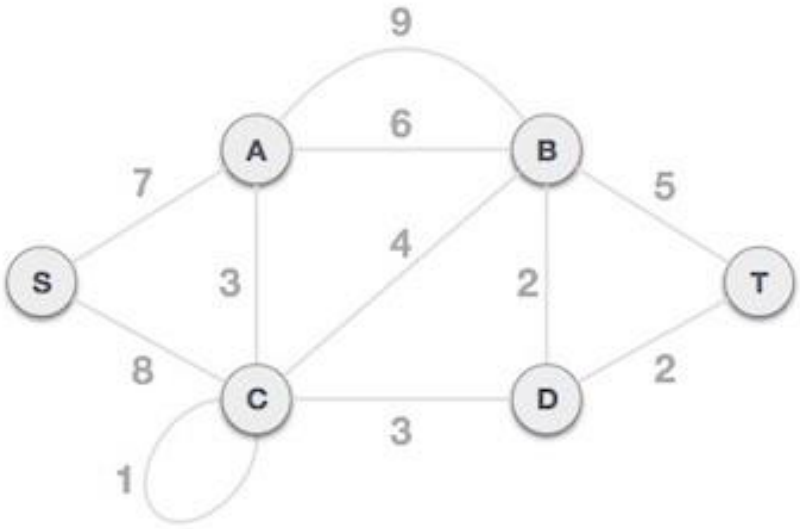
- Kruskal's Algorithm
- Prim's Algorithm

Both are greedy algorithms.

KRUSHKAL'S ALGORITHM

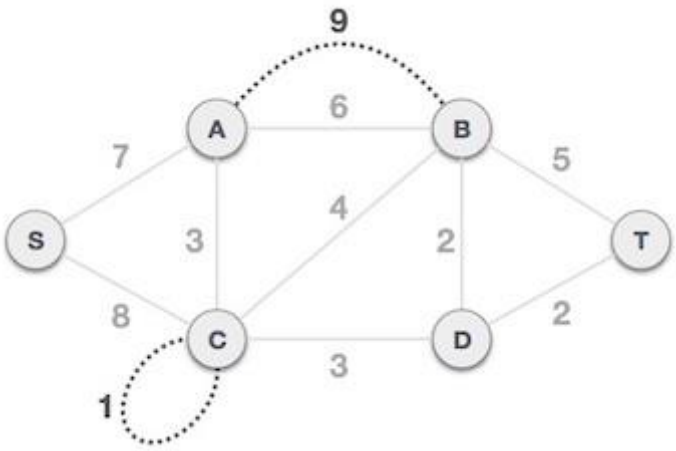
Kruskal's algorithm to find the minimum cost spanning tree uses the greedy approach. This algorithm treats the graph as a forest and every node it has as an individual tree. A tree connects to another only and only if, it has the least cost among all available options and does not violate MST properties.

To understand Kruskal's algorithm let us consider the following example −

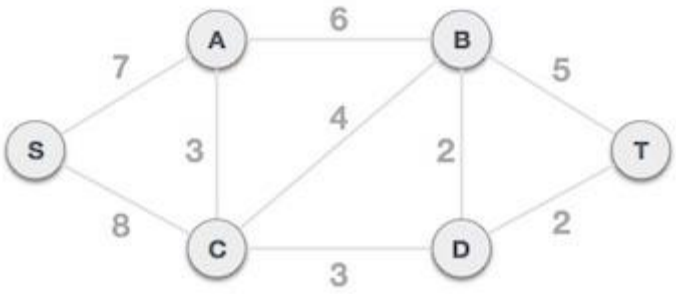


Step 1 - Remove all loops and Parallel Edges

Remove all loops and parallel edges from the given graph.



In case of parallel edges, keep the one which has the least cost associated and remove all others.



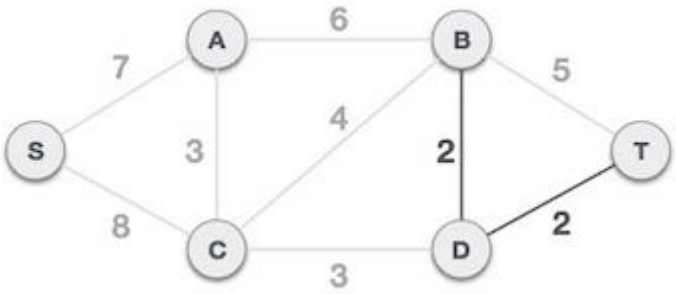
Step 2 - Arrange all edges in their increasing order of weight

The next step is to create a set of edges and weight, and arrange them in an ascending order of weightage (cost).

B, D	D, T	A, C	C, D	C, B	B, T	A, B	S, A	S, C
2	2	3	3	4	5	6	7	8

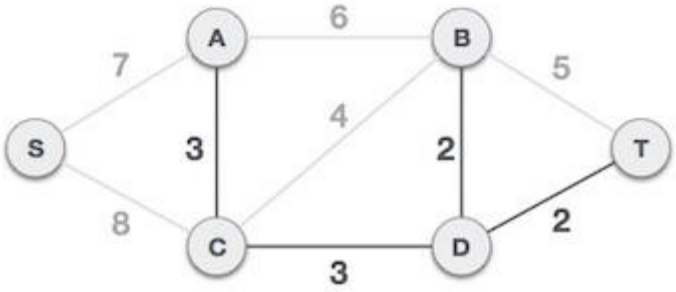
Step 3 - Add the edge which has the least weightage

Now we start adding edges to the graph beginning from the one which has the least weight. Throughout, we shall keep checking that the spanning properties remain intact. In case, by adding one edge, the spanning tree property does not hold then we shall consider not to include the edge in the graph.

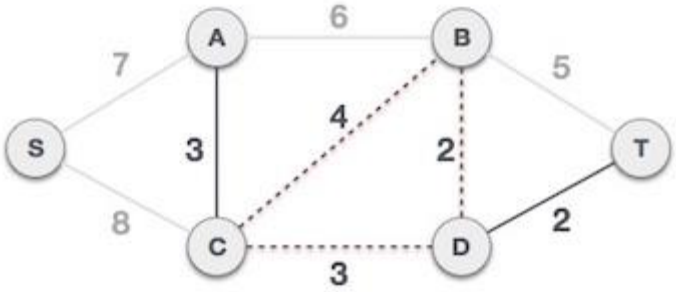


The least cost is 2 and edges involved are B,D and D,T. We add them. Adding them does not violate spanning tree properties, so we continue to our next edge selection.

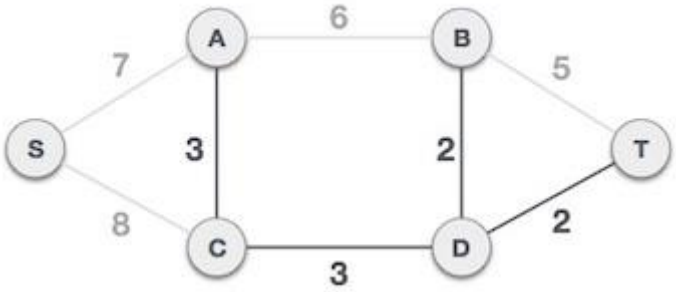
Next cost is 3, and associated edges are A,C and C,D. We add them again −



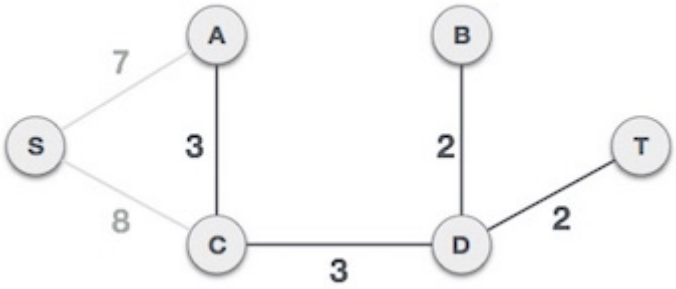
Next cost in the table is 4, and we observe that adding it will create a circuit in the graph. −



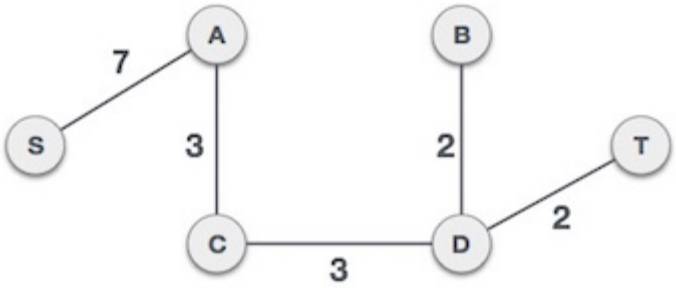
We ignore it. In the process we shall ignore/avoid all edges that create a circuit.



We observe that edges with cost 5 and 6 also create circuits. We ignore them and move on.



Now we are left with only one node to be added. Between the two least cost edges available 7 and 8, we shall add the edge with cost 7.



By adding edge S,A we have included all the nodes of the graph and we now have minimum cost spanning tree.

Source Code:

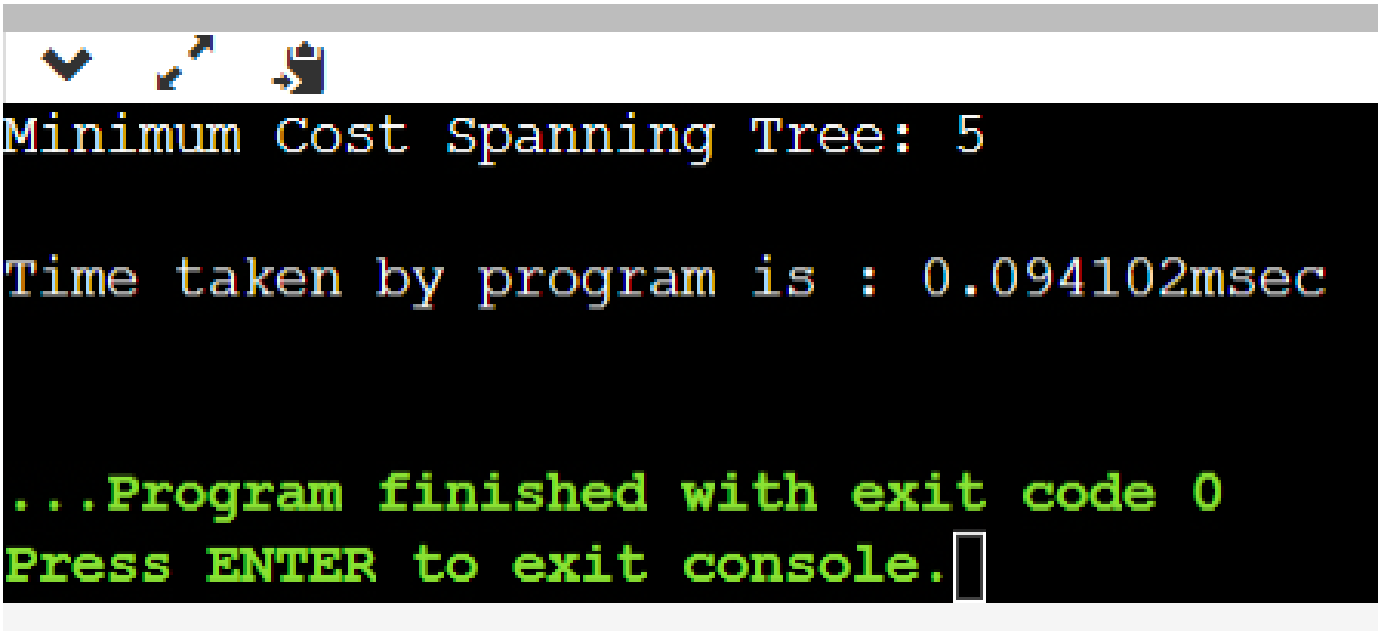
```
#include <bits/stdc++.h>
using namespace std;

class DSU
{
    int *parent;
    int *rank;

public:
    DSU(int n)
    {
        parent = new int[n];
        rank = new int[n];
```

```
        for (int i = 0; i < n; i++)
        {
            parent[i] = -1;
            rank[i] = 1;
        }
    }

    int find(int i)
    {
        if (parent[i] == -1)
            return i;

        return parent[i] = find(parent[i]);
    }

    void unite(int x, int y)
    {
        int s1 = find(x);
        int s2 = find(y);

        if (s1 != s2)
        {
            if (rank[s1] < rank[s2])
            {
                parent[s1] = s2;
                rank[s2] += rank[s1];
            }
            else
            {
                parent[s2] = s1;
                rank[s1] += rank[s2];
            }
        }
    }
};

class Graph
{
    vector<vector<int>> edgelist;
    int V;

public:
    Graph(int V)
    {
        this->V = V;
```

```
    }

    void addEdge(int x, int y, int w)
    {
        edgelist.push_back({w, x, y});
    }

    int kruskals_mst()
    {
        // 1. Sort all edges
        sort(edgelist.begin(), edgelist.end());

        // Initialize the DSU
        DSU s(V);
        int ans = 0;
        for (auto edge : edgelist)
        {
            int w = edge[0];
            int x = edge[1];
            int y = edge[2];

            // take that edge in MST if it does form a cycle
            if (s.find(x) != s.find(y))
            {
                s.unite(x, y);
                ans += w;
            }
        }
        return ans;
    }
};
int main()
{
    Graph g(4);
    g.addEdge(0, 1, 1);
    g.addEdge(1, 3, 3);
    g.addEdge(3, 2, 4);
    g.addEdge(2, 0, 2);
    g.addEdge(0, 3, 2);
    g.addEdge(1, 2, 2);

    auto start = chrono::high_resolution_clock::now();
    ios_base::sync_with_stdio(false);

    cout << "Minimum Cost Spanning Tree: " << g.kruskals_mst() << endl;
```

```
    auto end = chrono::high_resolution_clock::now();
    double time_taken = chrono::duration_cast<chrono::nanoseconds>(end -
start).count();

    time_taken *= 1e-9 * 1000;

    cout << "\nTime taken by program is : " << time_taken << setprecision(6);
    cout << "msec" << endl;

    return 0;
}
```

Output:

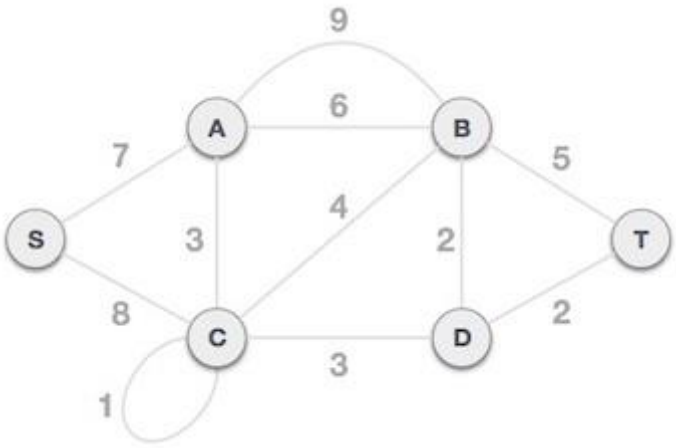


PRIM'S SPANNING TREE Algorithm

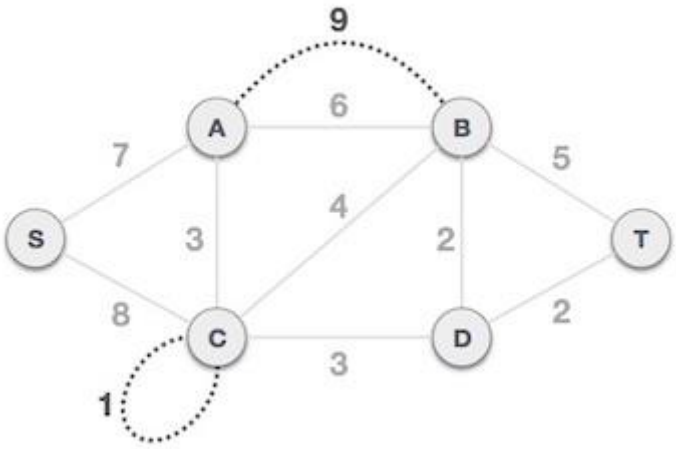
Prim's algorithm to find minimum cost spanning tree (as Kruskal's algorithm) uses the greedy approach. Prim's algorithm shares a similarity with the **shortest path first** algorithms.

Prim's algorithm, in contrast with Kruskal's algorithm, treats the nodes as a single tree and keeps on adding new nodes to the spanning tree from the given graph.

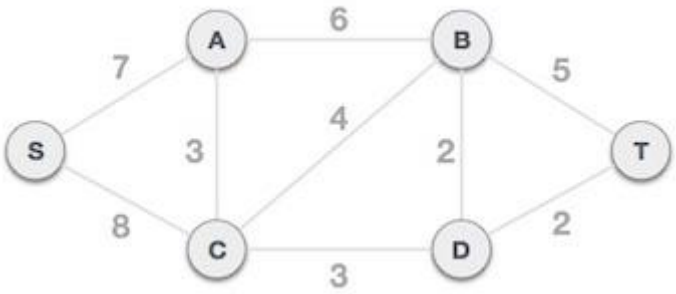
To contrast with Kruskal's algorithm and to understand Prim's algorithm better, we shall use the same example −



Step 1 - Remove all loops and parallel edges



Remove all loops and parallel edges from the given graph. In case of parallel edges, keep the one which has the least cost associated and remove all others.

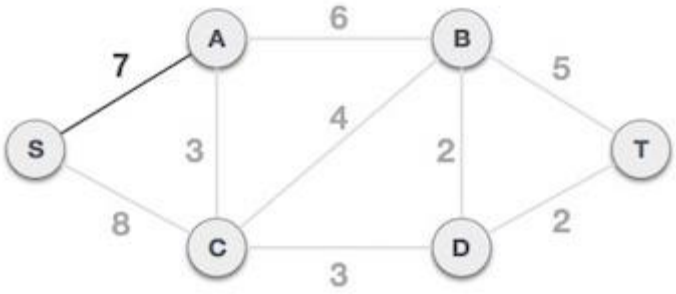


Step 2 - Choose any arbitrary node as root node

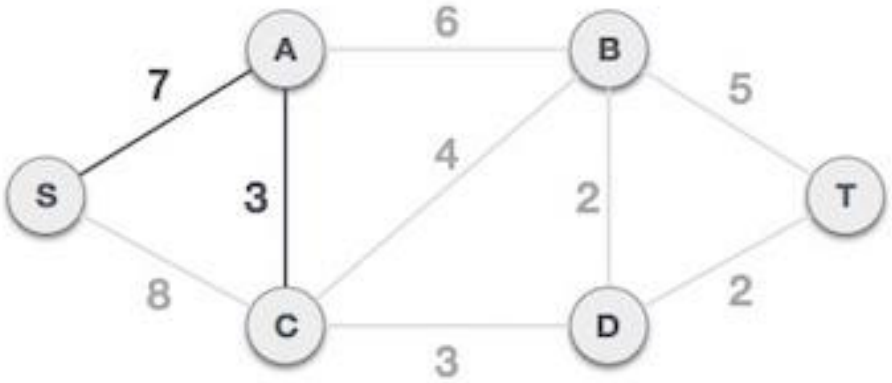
In this case, we choose **S** node as the root node of Prim's spanning tree. This node is arbitrarily chosen, so any node can be the root node. One may wonder why any video can be a root node. So the answer is, in the spanning tree all the nodes of a graph are included and because it is connected then there must be at least one edge, which will join it to the rest of the tree.

Step 3 - Check outgoing edges and select the one with less cost

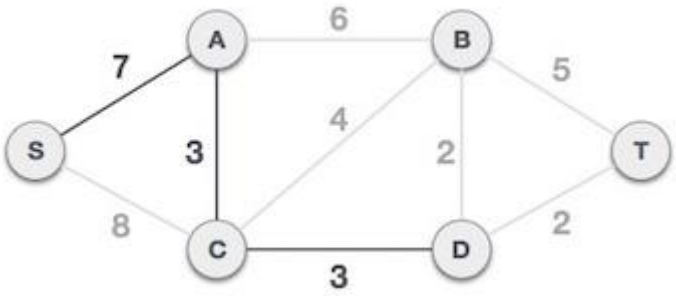
After choosing the root node **S**, we see that S,A and S,C are two edges with weight 7 and 8, respectively. We choose the edge S,A as it is lesser than the other.



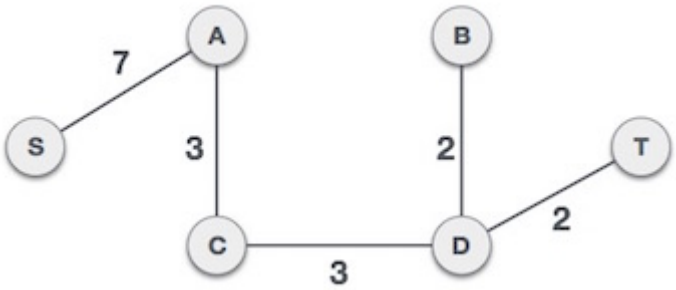
Now, the tree S-7-A is treated as one node and we check for all edges going out from it. We select the one which has the lowest cost and include it in the tree.



After this step, S-7-A-3-C tree is formed. Now we'll again treat it as a node and will check all the edges again. However, we will choose only the least cost edge. In this case, C-3-D is the new edge, which is less than other edges' cost 8, 6, 4, etc.



After adding node **D** to the spanning tree, we now have two edges going out of it having the same cost, i.e. D-2-T and D-2-B. Thus, we can add either one. But the next step will again yield edge 2 as the least cost. Hence, we are showing a spanning tree with both edges included.



We may find that the output spanning tree of the same graph using two different algorithms is same.

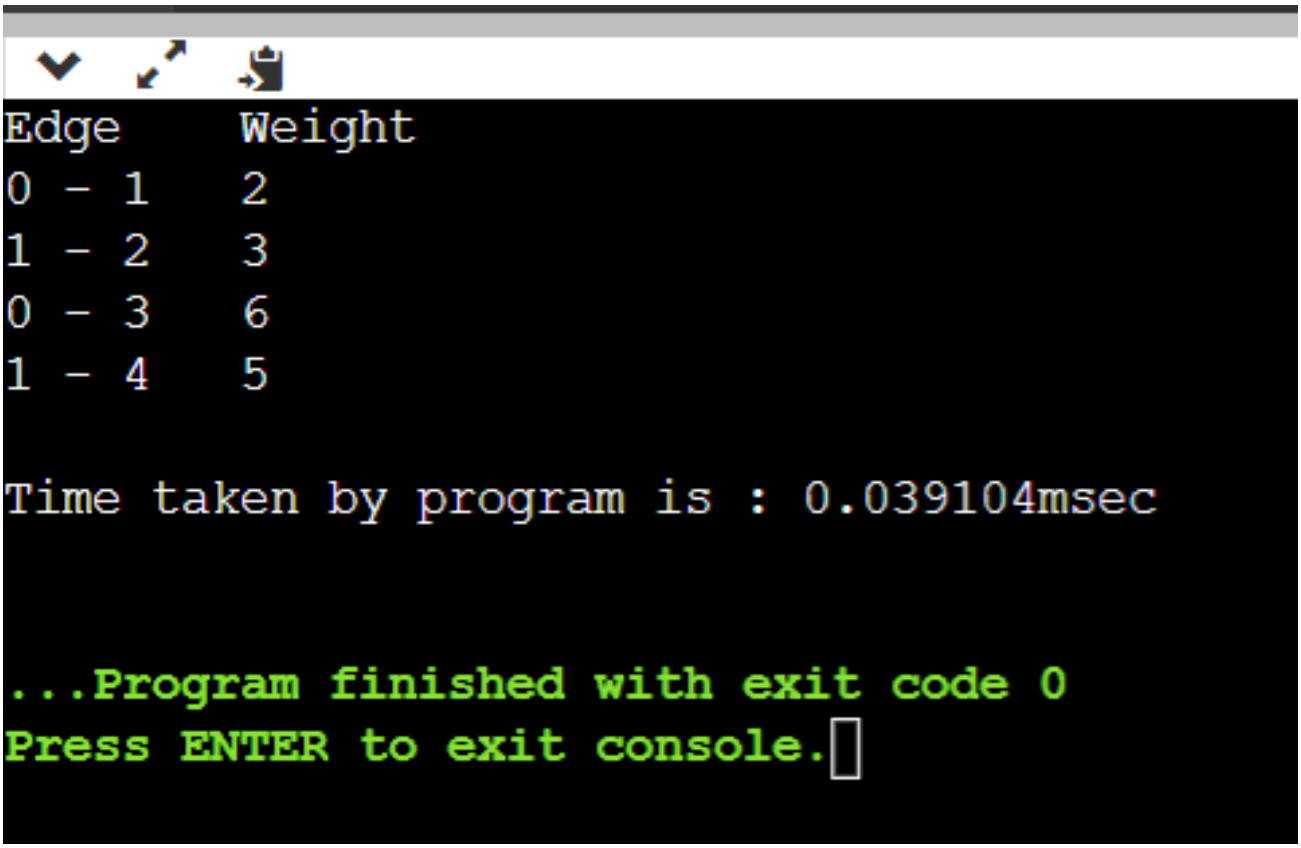
Source Code:

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

// Number of vertices in the graph
#define V 5

// A utility function to find the vertex with minimum key value, from the set of
vertices not yet included in MST
int minKey(int key[], bool mstSet[])
```

```
{
    // Initialize min value
    int min = INT_MAX, min_index;

    for (int v = 0; v < V; v++)
        if (mstSet[v] == false && key[v] < min)
            min = key[v], min_index = v;

    return min_index;
}

// A utility function to print the constructed MST stored in parent[]
void printMST(int parent[], int graph[V][V])
{
    cout << "Edge \tWeight\n";
    for (int i = 1; i < V; i++)
        cout << parent[i] << " - " << i << " \t" << graph[i][parent[i]] << " \n";
}

// Function to construct and print MST for a graph represented using adjacency
matrix representation
void primMST(int graph[V][V])
{
    // Array to store constructed MST
    int parent[V];

    // Key values used to pick minimum weight edge in cut
    int key[V];

    // To represent set of vertices included in MST
    bool mstSet[V];

    for (int i = 0; i < V; i++)
        key[i] = INT_MAX, mstSet[i] = false;

    key[0] = 0;
    parent[0] = -1; // First node is always root of MST

    // The MST will have V vertices
    for (int count = 0; count < V - 1; count++)
    {
        int u = minKey(key, mstSet);
        mstSet[u] = true;
        for (int v = 0; v < V; v++)
            if (graph[u][v] && mstSet[v] == false && graph[u][v] < key[v])
```

```
                parent[v] = u, key[v] = graph[u][v];
    }

    // print the constructed MST
    printMST(parent, graph);
}

int main()
{
    /* Let us create the following graph
        2 3
    (0)--(1)--(2)
    | / \ |
    6| 8/ \5 |7
    | / \ |
    (3)-------(4)
            9     */
    int graph[V][V] = {{0, 2, 0, 6, 0},
                       {2, 0, 3, 8, 5},
                       {0, 3, 0, 0, 7},
                       {6, 8, 0, 0, 9},
                       {0, 5, 7, 9, 0}};

    auto start = chrono::high_resolution_clock::now();
    ios_base::sync_with_stdio(false);

    primMST(graph);

    auto end = chrono::high_resolution_clock::now();
    double time_taken = chrono::duration_cast<chrono::nanoseconds>(end -
start).count();

    time_taken *= 1e-9 * 1000;

    cout << "\nTime taken by program is : " << time_taken << setprecision(6);
    cout << "msec" << endl;

    return 0;
}
```

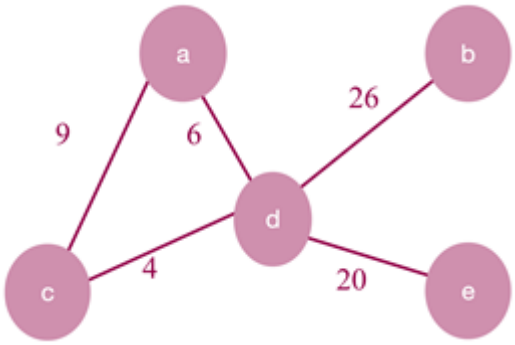
Output:


Edge Weight
0 - 1 2
1 - 2 3
0 - 3 6
1 - 4 5
Time taken by program is : 0.039104msec
...Program finished with exit code 0
Press ENTER to exit console.


input
Edge Weight
0 - 1 2
1 - 2 3
0 - 3 6
1 - 4 5
Time taken by program is : 0.039104msec
...Program finished with exit code 0
Press ENTER to exit console.

Viva Questions

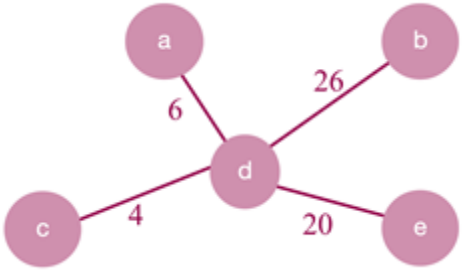
1. Consider the graph shown below. What are the edges in the MST of the given graph?



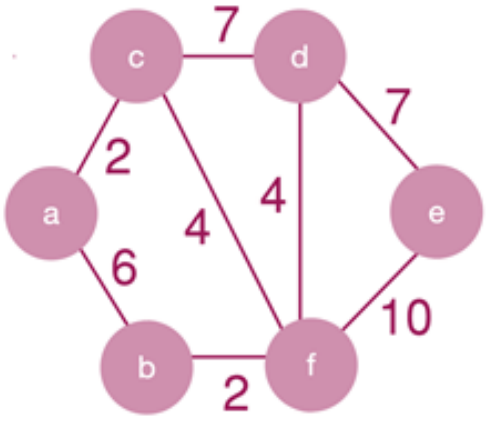
Ans.

(a-d)(d-c)(d-b)(d-e)

The minimum spanning tree of the given graph is shown below. It has cost 56.



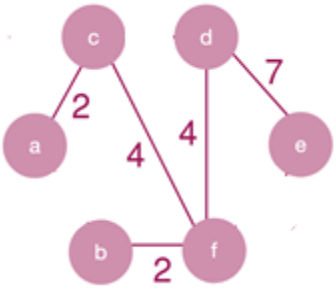
2 Consider the given graph.



What is the weight of the minimum spanning tree using the Kruskal's algorithm?

Ans.

Kruskal's algorithm constructs the minimum spanning tree by constructing by adding the edges to spanning tree one-one by one. The MST for the given graph is,



So, the weight of the MST is 19.

3. What is the time complexity of Kruskal's algorithm?

Ans.

Kruskal's algorithm involves sorting of the edges, which takes O(E logE) time, where E is a number of edges in graph and V is the number of vertices. After sorting, all edges are iterated and union-find algorithm is applied. union-find algorithm requires O(logV) time. So, overall Kruskal's algorithm requires O(E log V) time.

4. Worst case is the worst case time complexity of Prim's algorithm if adjacency matrix is used?

Ans.

Use of adjacency matrix provides the simple implementation of the Prim's algorithm. In Prim's algorithm, we need to search for the edge with a minimum for that vertex. So, worst case time complexity will be O(V2), where V is the number of vertices.

5. Prim's algorithm is also known as _____

Ans.

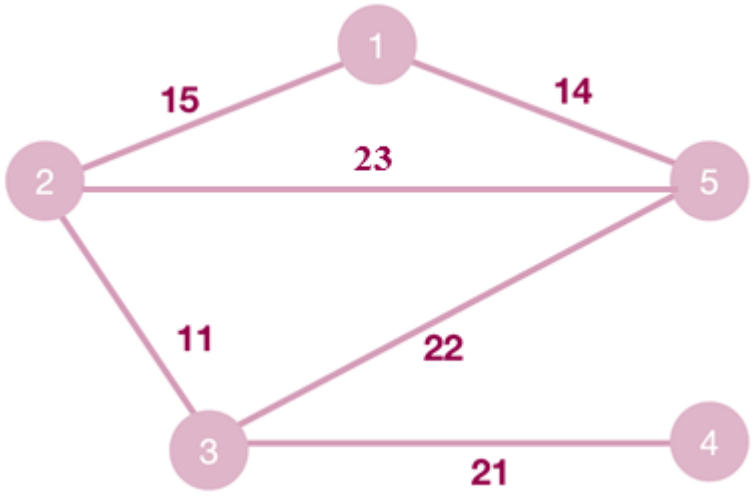
The Prim's algorithm was developed by Vojtěch Jarník and it was latter discovered by the duo Prim and Dijkstra. Therefore, Prim's algorithm is also known as DJP Algorithm.

6. Prim's algorithm can be efficiently implemented using _____ for graphs with greater density.

Ans.

In Prim's algorithm, we add the minimum weight edge for the chosen vertex which requires searching on the array of weights. This searching can be efficiently implemented using binary heap for dense graphs. And for graphs with greater density, Prim's algorithm can be made to run in linear time using d-ary heap(generalization of binary heap).

7. Consider the graph shown below.

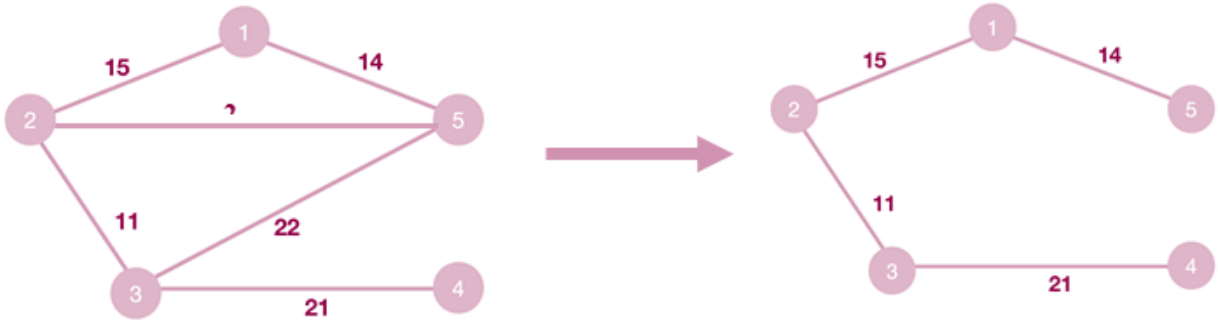


which of the following edges form the MST of the given graph using Prim'a algorithm, starting from vertex 4.

Ans.

(4-3)(3-2)(2-1)(1-5)

The MST for the given graph using Prim's algorithm starting from vertex 4 is,



So, the MST contains edges (4-3)(3-2)(2-1)(1-5).

8. Prim's algorithm is a _____

Ans.

Prim's algorithm uses a greedy algorithm approach to find the MST of the connected weighted graph. In greedy method, we attempt to find an optimal solution in stages.

EXPERIMENT - 8

Algorithms Design and Analysis Lab

Aim

To implement String matching algorithm and analyse its time complexity.

Syeda Reeha Quasar
14114802719
4C7

EXPERIMENT – 8

Aim:

To implement String matching algorithm and analyse its time complexity.

Theory:

String matching algorithms have greatly influenced computer science and play an essential role in various real-world problems. It helps in performing time-efficient tasks in multiple domains. These algorithms are useful in the case of searching a string within another string. String matching is also used in the Database schema, Network systems.

Exact string matching algorithms is to find one, several, or all occurrences of a defined string (pattern) in a large string (text or sequences) such that each matching is perfect. All alphabets of patterns must be matched to corresponding matched subsequence. These are further classified into four categories:

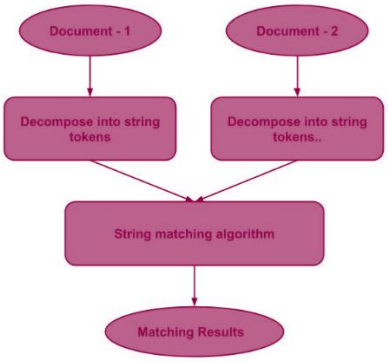
1. Algorithms based on character comparison:
    - Naive Algorithm: It slides the pattern over text one by one and check for a match. If a match is found, then slides by 1 again to check for subsequent matches.
    - KMP (Knuth Morris Pratt) Algorithm: The idea is whenever a mismatch is detected, we already know some of the characters in the text of the next window. So, we take advantage of this information to avoid matching the characters that we know will anyway match.
    - Boyer Moore Algorithm: This algorithm uses best heurestics of Naive and KMP algorithm and starts matching from the last character of the pattern.
    - Using the Trie data structure: It is used as an efficient information retrieval data structure. It stores the keys in form of a balanced BST.
2. Deterministic Finite Automaton (DFA) method:
    - Automaton Matcher Algorithm: It starts from the first state of the automata and the first character of the text. At every step, it considers next character of text, and look for the next state in the built finite automata and move to a new state.
3. Algorithms based on Bit (parallelism method):

- Aho-Corasick Algorithm: It finds all words in O(n + m + z) time where n is the length of text and m be the total number characters in all words and z is total number of occurrences of words in text. This algorithm forms the basis of the original Unix command fgrep.

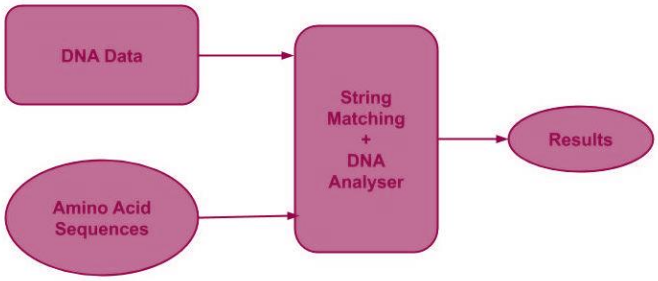
4. Hashing-string matching algorithms:
   - Rabin Karp Algorithm: It matches the hash value of the pattern with the hash value of current substring of text, and if the hash values match then only it starts matching individual characters.

Applications of String Matching Algorithms:

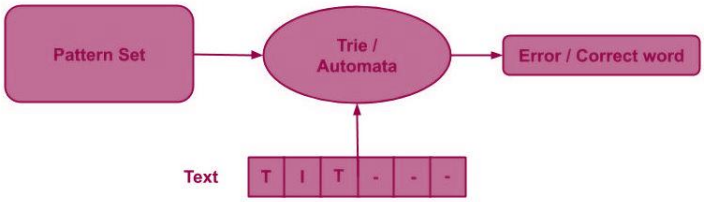
- **Plagiarism Detection:** The documents to be compared are decomposed into string tokens and compared using string matching algorithms. Thus, these algorithms are used to detect similarities between them and declare if the work is plagiarized or original.



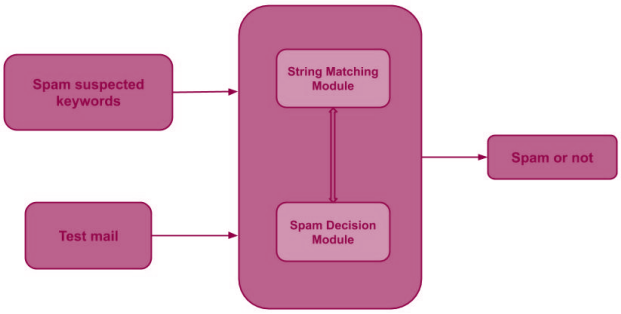
- **Bioinformatics and DNA Sequencing:** Bioinformatics involves applying information technology and computer science to problems involving genetic sequences to find DNA patterns. String matching algorithms and DNA analysis are both collectively used for finding the occurrence of the pattern set.



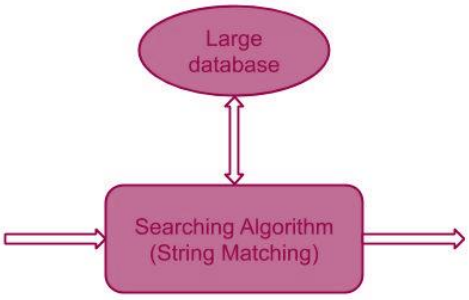
- **Digital Forensics:** String matching algorithms are used to locate specific text strings of interest in the digital forensic text, which are useful for the investigation.
- **Spelling Checker:** Trie is built based on a predefined set of patterns. Then, this trie is used for string matching. The text is taken as input, and if any such pattern occurs, it is shown by reaching the acceptance state.



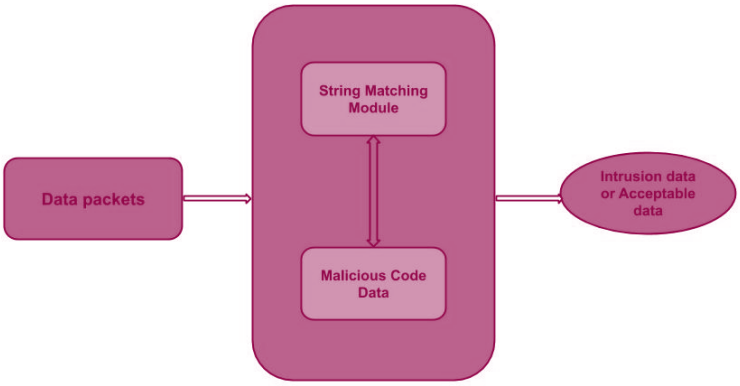
- **Spam filters:** Spam filters use string matching to discard the spam. For example, to categorize an email as spam or not, suspected spam keywords are searched in the content of the email by string matching algorithms. Hence, the content is classified as spam or not.



- **Search engines or content search in large databases:** To categorize and organize data efficiently, string matching algorithms are used. Categorization is done based on the search keywords. Thus, string matching algorithms make it easier for one to find the information they are searching for.



- **Intrusion Detection System:** The data packets containing intrusion-related keywords are found by applying string matching algorithms. All the malicious code is stored in the database, and every incoming data is compared with stored data. If a match is found, then the alarm is generated. It is based on exact string matching algorithms where each intruded packet must be detected.



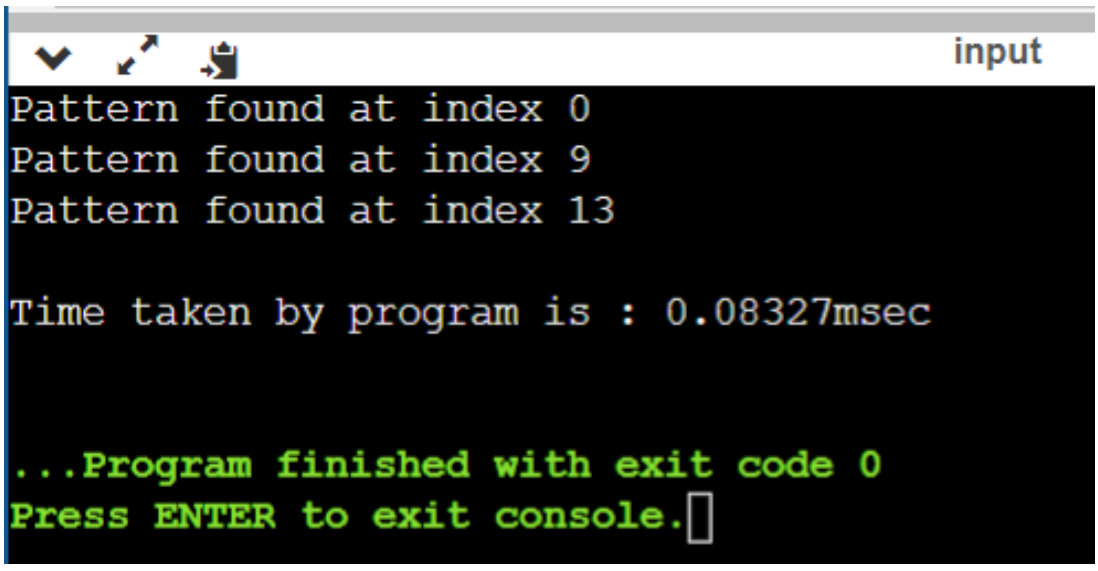
NAÏVE STRING-MATCHING ALGORITHM

Source Code:

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

void search(char *pat, char *txt)
{
    int M = strlen(pat);
    int N = strlen(txt);

    /* A loop to slide pat[] one by one */
    for (int i = 0; i <= N - M; i++)
    {
        int j;

        /* For current index i, check for pattern match */
        for (j = 0; j < M; j++)
            if (txt[i + j] != pat[j])
                break;
```

```
        if (j == M) // if pat[0...M-1] = txt[i, i+1, ...i+M-1]
            cout << "Pattern found at index "
                 << i << endl;
    }
}

int main()
{
    char txt[] = "AABAACAADAABAAABAA";
    char pat[] = "AABA";

    auto start = chrono::high_resolution_clock::now();
    ios_base::sync_with_stdio(false);

    search(pat, txt);

    auto end = chrono::high_resolution_clock::now();
    double time_taken = chrono::duration_cast<chrono::nanoseconds>(end -
start).count();

    time_taken *= 1e-9 * 1000;

    cout << "\nTime taken by program is : " << time_taken << setprecision(6);
    cout << "msec" << endl;

    return 0;
}
```

Output:



Rabin Karp String Matching Algorithm

Rabin Karp algorithm needs to calculate hash values for following strings.
1) Pattern itself.
2) All the substrings of the text of length m.

Since we need to efficiently calculate hash values for all the substrings of size m of text, we must have a hash function which has the following property.
Hash at the next shift must be efficiently computable from the current hash value and next character in text or we can say *hash(txt[s+1 .. s+m])* must be efficiently computable from *hash(txt[s .. s+m-1])* and *txt[s+m]* i.e., *hash(txt[s+1 .. s+m])= rehash(txt[s+m], hash(txt[s .. s+m-1]))* and rehash must be O(1) operation.
The hash function suggested by Rabin and Karp calculates an integer value. The integer value for a string is the numeric value of a string.

For example, if all possible characters are from 1 to 10, the numeric value of "122" will be 122. The number of possible characters is higher than 10 (256 in general) and pattern length can be large. So the numeric values cannot be practically stored as an integer. Therefore, the numeric value is calculated using modular arithmetic to make sure that the hash values can be stored in an integer variable (can fit in memory words). To do rehashing, we need to take off the most significant digit and add the new least significant digit for in hash value.

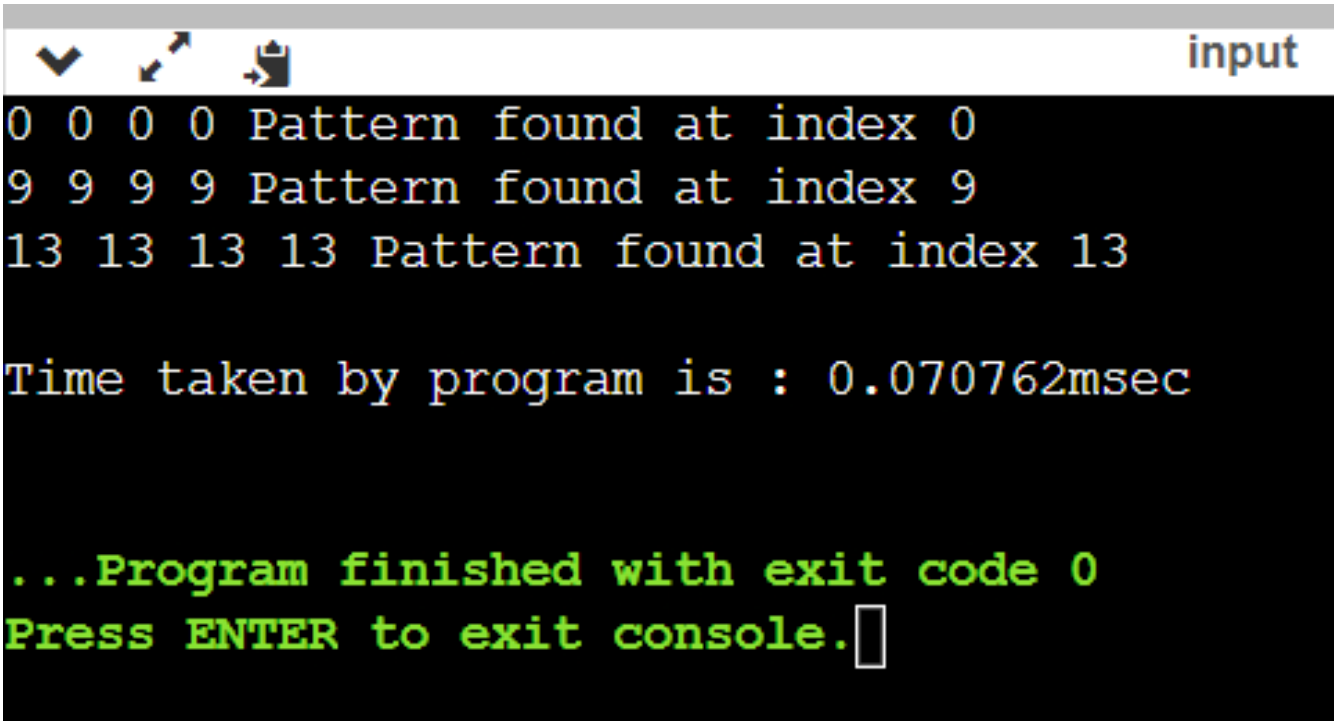
Source Code:

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

#define d 256

void search(char pat[], char txt[], int q)
{
    int M = strlen(pat);
    int N = strlen(txt);
    int i, j;
    int p = 0; // hash value for pattern
    int t = 0; // hash value for txt
    int h = 1;

    // The value of h would be "pow(d, M-1)%q"
    for (i = 0; i < M - 1; i++)
        h = (h * d) % q;

    // Calculate the hash value of pattern and first window of text
    for (i = 0; i < M; i++)
```

```
    {
        p = (d * p + pat[i]) % q;
        t = (d * t + txt[i]) % q;
    }

    // Slide the pattern over text one by one
    for (i = 0; i <= N - M; i++)
    {
        if (p == t)
        {
            bool flag = true;
            /* Check for characters one by one */
            for (j = 0; j < M; j++)
            {
                if (txt[i + j] != pat[j])
                {
                    flag = false;
                    break;
                }
                if (flag)
                    cout << i << " ";
            }

            // if p == t and pat[0...M-1] = txt[i, i+1, ...i+M-1]

            if (j == M)
                cout << "Pattern found at index " << i << endl;
        }

        // Calculate hash value for next window of text: Remove leading digit,
add trailing digit
        if (i < N - M)
        {
            t = (d * (t - txt[i] * h) + txt[i + M]) % q;

            // We might get negative value of t, converting it to positive
            if (t < 0)
                t = (t + q);
        }
    }
}

int main()
{
    char txt[] = "AABAACAADAABAAABAA";
```

```
    char pat[] = "AABA";

    // A prime number
    int q = 101;

    auto start = chrono::high_resolution_clock::now();
    ios_base::sync_with_stdio(false);

    search(pat, txt, q);

    auto end = chrono::high_resolution_clock::now();
    double time_taken = chrono::duration_cast<chrono::nanoseconds>(end -
start).count();

    time_taken *= 1e-9 * 1000;

    cout << "\nTime taken by program is : " << time_taken << setprecision(6);
    cout << "msec" << endl;

    return 0;
}
```

Output:



Knuth Morris Pratt algorithm

- KMP algorithm preprocesses pat[] and constructs an auxiliary **lps[]** of size m (same as size of pattern) which is used to skip characters while matching.
- **name lps indicates longest proper prefix which is also suffix.**. A proper prefix is prefix with whole string **not** allowed. For example, prefixes of "ABC" are "", "A", "AB" and "ABC". Proper prefixes are "", "A" and "AB". Suffixes of the string are "", "C", "BC" and "ABC".
- We search for lps in sub-patterns. More clearly we focus on sub-strings of patterns that are either prefix and suffix.
- For each sub-pattern pat[0..i] where i = 0 to m-1, lps[i] stores length of the maximum matching proper prefix which is also a suffix of the sub-pattern pat[0..i].
- lps[i] = the longest proper prefix of pat[0..i]

  which is also a suffix of pat[0..i].

Matching Overview

txt = "AAAAABAAABA"

pat = "AAAA"

We compare first window of **txt** with **pat**

txt = "**AAAA**ABAAABA"

pat = "**AAAA**" [Initial position]

We find a match. This is same as Naive String Matching.

In the next step, we compare next window of **txt** with **pat**.

txt = "**AAAAA**BAAABA"

pat = "**AAAA**" [Pattern shifted one position]

This is where KMP does optimization over Naive. In this second window, we only compare fourth A of pattern with fourth character of current window of text to decide whether current window matches or not. Since we know first three characters will anyway match, we skipped matching first three characters.

Need of Preprocessing?

An important question arises from the above explanation,

how to know how many characters to be skipped. To know this,

we pre-process pattern and prepare an integer array

lps[] that tells us the count of characters to be skipped.

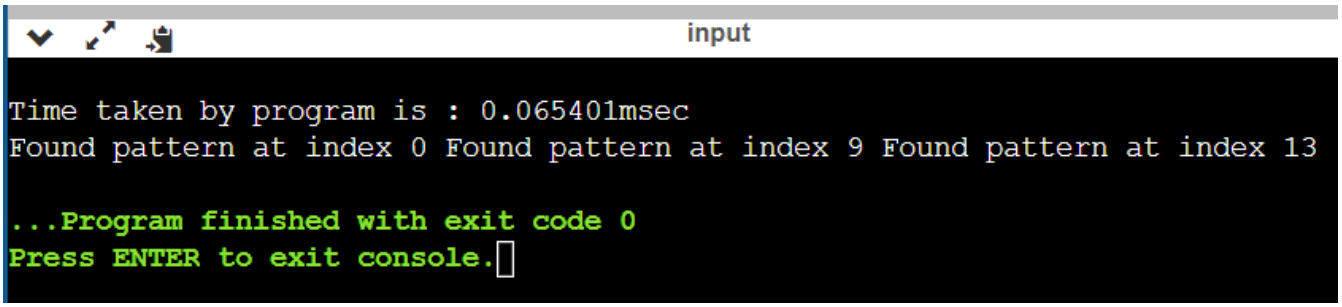
Source Code:

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

void computeLPSArray(char *pat, int M, int *lps);

void KMPSearch(char *pat, char *txt)
{
    int M = strlen(pat);
    int N = strlen(txt);

    int lps[M];

    // Preprocess the pattern (calculate lps[] array)
    computeLPSArray(pat, M, lps);

    int i = 0; // index for txt[]
    int j = 0; // index for pat[]
    while (i < N)
    {
        if (pat[j] == txt[i])
        {
            j++;
            i++;
        }

        if (j == M)
        {
            printf("Found pattern at index %d ", i - j);
            j = lps[j - 1];
        }

        // mismatch after j matches
```

```
        else if (i < N && pat[j] != txt[i])
        {
            // Do not match lps[0..lps[j-1]] characters, they will match anyway
            if (j != 0)
                j = lps[j - 1];
            else
                i = i + 1;
        }
    }
}

// Fills lps[] for given patttern pat[0..M-1]
void computeLPSArray(char *pat, int M, int *lps)
{
    // length of the previous longest prefix suffix
    int len = 0;

    lps[0] = 0; // lps[0] is always 0

    // the loop calculates lps[i] for i = 1 to M-1
    int i = 1;
    while (i < M)
    {
        if (pat[i] == pat[len])
        {
            len++;
            lps[i] = len;
            i++;
        }
        else // (pat[i] != pat[len])
        {
            if (len != 0)
            {
                len = lps[len - 1];
            }
            else // if (len == 0)
            {
                lps[i] = 0;
                i++;
            }
        }
    }
}

int main()
```

```
{
    char txt[] = "AABAACAADAABAAABAA";
    char pat[] = "AABA";

    auto start = chrono::high_resolution_clock::now();
    ios_base::sync_with_stdio(false);

    KMPSearch(pat, txt);

    auto end = chrono::high_resolution_clock::now();
    double time_taken = chrono::duration_cast<chrono::nanoseconds>(end -
start).count();

    time_taken *= 1e-9 * 1000;

    cout << "\nTime taken by program is : " << time_taken << setprecision(6);
    cout << "msec" << endl;

    return 0;
}
```

Output:



Viva Questions

1. What is the worst case time complexity of KMP algorithm for pattern searching (m = length of text, n = length of pattern)?

Ans.

KMP algorithm is an efficient pattern searching algorithm. It has a time complexity of O(m) where m is the length of text.

2. The naive pattern searching algorithm is an in place algorithm or not.

Ans.

The auxiliary space complexity required by naive pattern searching algorithm is O(1). So it qualifies as an in place algorithm.

3. Describe about Rabin Karp algorithm and naive pattern searching algorithm worst case time complexity.

Ans.

The worst case time complexity of Rabin Karp algorithm is O(m*n) but it has a linear average case time complexity. So Rabin Karp and naive pattern searching algorithm have the same worst case time complexity.

4. What is a Rabin and Karp Algorithm?

Ans.

The string matching algorithm which was proposed by Rabin and Karp, generalizes to other algorithms and for two-dimensional pattern matching problems.

5. What is the pre-processing time of Rabin and Karp Algorithm?

Ans.

The for loop in the pre-processing algorithm runs for m(length of the pattern) times. Hence the pre-processing time is Theta(m).

6. What is the basic formula applied in Rabin Karp Algorithm to get the computation time as Theta(m)?

Ans.

The pattern can be evaluated in time Theta(m) using Horner's rule:
p = P[m] + 10(P[m-1] + 10(P[m-2] +…+ 10(P[2]+10P[1])…)).

7. What happens when the modulo value(q) is taken large?

Ans.

If the modulo value(q) is large enough then the spurious hits occur infrequently enough that the cost of extra checking is low.