***ALGORITHMS DESIGN AND ANALYSIS LAB***

***Paper code – ETCS-351***

Faculty Name: varun goel sir Student Name: Aman dixit

Roll No.: 14614803119

Semester: 5th

Group: I-7



***Maharaja Agrasen Institute Of Technology***

***PSP Area, Sector-22, Rohini,***

***Delhi-110085***

**PRACTICAL DETAILS**

Experiments according to the list provided by GGSIPU

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **EXP NO.** | **EXPERIMENT NAME** | **DATE** | **REMARKS** | **MARKS** |
| 1 | To implement following algorithm using array as a data structure and analyse its time complexity :-  a. Merge Sort  b. Quick Sort  c. Bubble Sort  d. Bucket Sort  e. Radix Sort  f. Shell Sort  g. Selection Sort  h. Heap Sort |  |  |  |
| 2 | To implement following algorithms and analyse its time complexity :-  a. Linear Search  b. Binary Search |  |  |  |
| 3 | To analyse matrix multiplication and analyse its time complexity. |  |  |  |
| 4 | To implement Longest Common Subsequence problem and analyse its time complexity. |  |  |  |
| 5 | To implement Optimal Binary Search problem and analyse its time complexity. |  |  |  |
| 6 | To implement Huffman Coding and analyse its time complexity |  |  |  |
| 7 | To implement Dijkstra’s algorithm and analyse its time complexity. |  |  |  |
| 8 | To implement Bellman Ford algorithm and analyse its time complexity. |  |  |  |
| 9 | To implement following algorithms and analyse its time complexity :-  a. Naive String Matching algorithm  b. Rabin Karp algorithm  c. Knuth Morris Pratt algorithm |  |  |  |

**EXPERIMENT-01**

**AIM:-** To implement following algorithm using array as a data structure and analyse its time complexity :-

a. Merge Sort

b. Quick Sort

c. Bubble Sort

d. Bucket Sort

e. Radix Sort

f. Shell Sort

g. Selection Sort

h. Heap Sort

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.

**SOURCE CODE:-**

#include <bits/stdc++.h>

using namespace std;

void merge(int arr[], int s, int mid, int e) {

int n1, n2, i, k, j;

n1 = mid - s + 1;

n2 = e - mid;

int left[n1], right[n2];

for (i = 0; i < n1; i++) {

left[i] = arr[s + i];

}

for (i = 0; i < n2; i++) {

right[i] = arr[mid + i + 1];

}

i = 0;

j = 0;

k = s;

while (i < n1 && j < n2) {

if (left[i] <= right[j]) {

arr[k] = left[i];

i++;

}

else {

arr[k] = right[j];

j++;

}

k++;

}

while (i < n1) {

arr[k] = left[i];

i++;

k++;

}

while (j < n2) {

arr[k] = right[j];

j++;

k++;

}

}

void mergeSort(int arr[], int s, int e) {

int mid;

if (s < e) {

mid = (s + e) / 2;

mergeSort(arr, s, mid);

mergeSort(arr, mid + 1, e);

merge(arr, s, mid, e);

}

}

void bestCase(int n) { //Insert in sorted order

int \*arr = new int[n];

for (int i = 0; i < n; i++) {

arr[i] = i;

}

typedef std::chrono::high\_resolution\_clock Time;

typedef std::chrono::milliseconds ms;

typedef std::chrono::duration fsec;

auto t0 = Time::now();

mergeSort(arr, 0, n - 1);

auto t1 = Time::now();

fsec fs = t1 - t0;

cout << "Time taken : " << fs.count() << "s " << endl;

}

void worstCase(int n) { //Insert in Decreasing order

int \*arr = new int[n];

for (int i = 0; i < n; i++) {

arr[i] = n - i;

}

typedef std::chrono::high\_resolution\_clock Time;

typedef std::chrono::milliseconds ms;

typedef std::chrono::duration fsec;

auto t0 = Time::now();

mergeSort(arr, 0, n - 1);

auto t1 = Time::now();

fsec fs = t1 - t0;

cout << "Time taken : " << fs.count() << "s " << endl;

}

void averageCase(int n) {

int \*arr = new int[n];

for (int i = 0; i < n; i++) {

arr[i] = rand() % n;

}

typedef std::chrono::high\_resolution\_clock Time;

typedef std::chrono::milliseconds ms;

typedef std::chrono::duration fsec;

auto t0 = Time::now();

mergeSort(arr, 0, n - 1);

auto t1 = Time::now(); fsec fs = t1 - t0;

cout << "Time taken : " << fs.count() << "s " << endl;

}

int main() {

cout << "Best Case Time Complexity" << "\n";

for (int i = 1; i <= 3; i++) {

cout << "For " << i \* 10000 << " elements, ";

bestCase(i \* 10000);

}

cout << "Worst Case Time Complexity" << "\n";

for (int i = 1; i <= 3; i++) {

cout << "For " << i \* 10000 << " elements, ";

worstCase(i \* 10000);

}

cout << "Average Case Time Complexity" << "\n";

for (int i = 1; i <= 3; i++) {

cout << "For " << i \* 10000 << " elements, ";

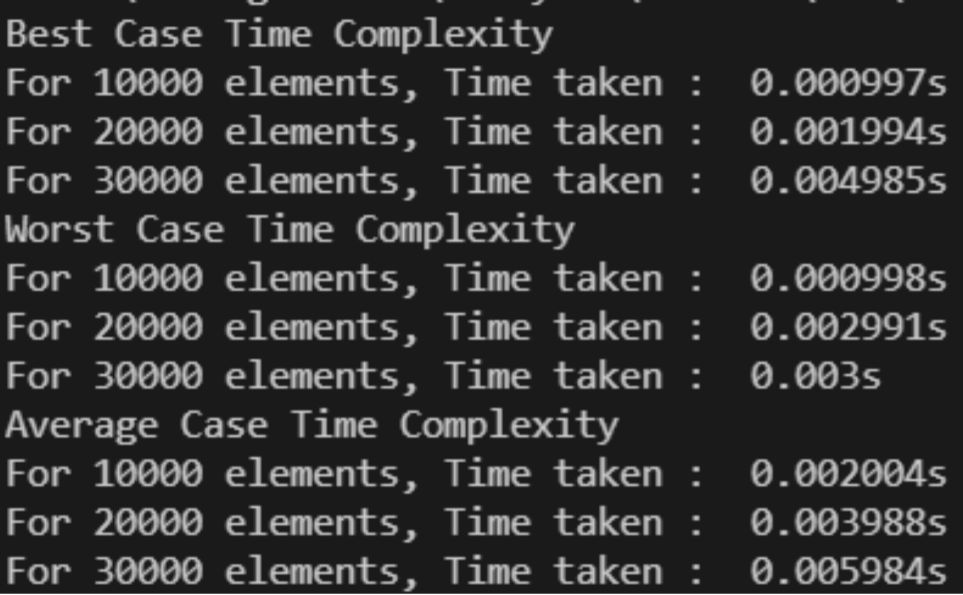
averageCase(i \* 10000);

}

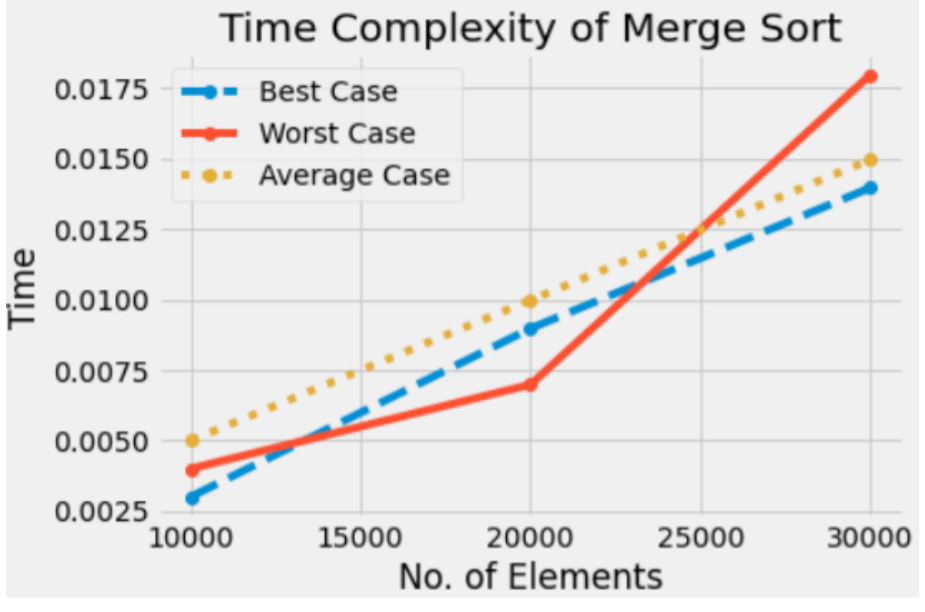
return 0;

}

**OUTPUT :-**



**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 log(N) parts, and the merging of all sub lists into a single list takes O(N) time. Time complexity of Merge Sort is O (n log(n)) in all 3 cases (worst, average and best) as merge sort always divides the array in two halves and takes linear time to merge two halves. The recurrence equation used is :

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

1. **Quick Sort**

The basic steps in this sort are:

**Divide**: The array A [p… r] is partitioned (rearranged) into two nonempty sub arrays A [p... q] and A [q + 1... r] such that each element of A [p . . q] is less than or equal to each element of A [q + 1... r]. The index q is computed as part of this partitioning procedure.

**Conquer**: The two sub arrays A [p… q] and A [q + 1… r] are sorted by recursive calls to quick sort.

**Combine**: Since the sub arrays are sorted in place, no work is needed to combine them: the entire array A [p.. r] is now sorted.

**SOURCE CODE:-**

#include <bits/stdc++.h>

using namespace std;

int partition(int \*arr, int low, int high) {

int pvt = arr[high];

int idx = low - 1;

for (int j = low; j <= high - 1; j++) {

if (pvt > arr[j]) {

idx++; swap(arr[idx], arr[j]);

}

} swap(arr[idx + 1],

arr[high]);

return (idx + 1);

}

void quickSort(int \*arr, int l, int h) {

if (l < h) {

int pvt = partition(arr, l, h);

quickSort(arr, l, pvt - 1);

quickSort(arr, pvt + 1, h);

}

}

void bestCase(int n) {

int \*arr = new int[n];

for (int i = 0; i < n; i++) {

arr[i] = i;

}

typedef std::chrono::high\_resolution\_clock Time;

typedef std::chrono::milliseconds ms;

typedef std::chrono::duration fsec;

auto t0 = Time::now();

quickSort(arr, 0, n - 1);

auto t1 = Time::now();

fsec fs = t1 - t0;

cout << "Time taken : " << fs.count() << "s " << endl;

}

void worstCase(int n) { //Insert in Decreasing order

int \*arr = new int[n];

for (int i = 0; i < n; i++) {

arr[i] = n - i;

}

typedef std::chrono::high\_resolution\_clock Time;

typedef std::chrono::milliseconds ms;

typedef std::chrono::duration fsec;

auto t0 = Time::now();

quickSort(arr, 0, n - 1);

auto t1 = Time::now();

fsec fs = t1 - t0;

cout << "Time taken : " << fs.count() << "s " << endl;

}

void averageCase(int n) {

int \*arr = new int[n];

for (int i = 0; i < n; i++) {

arr[i] = rand() % n;

}

typedef std::chrono::high\_resolution\_clock Time;

typedef std::chrono::milliseconds ms;

typedef std::chrono::duration fsec;

auto t0 = Time::now();

quickSort(arr, 0, n - 1);

auto t1 = Time::now();

fsec fs = t1 - t0;

cout << "Time taken : " << fs.count() << "s " << endl;

}

int main() {

cout << "Best Case Time Complexity" << "\n";

for (int i = 1; i <= 3; i++) {

cout << "For " << i \* 10000 << " elements, ";

bestCase(i \* 10000); } cout << "Worst Case Time Complexity" << "\n";

}

for (int i = 1; i <= 3; i++) {

cout << "For " << i \* 10000 << " elements, ";

worstCase(i \* 10000); } cout << "Average Case Time Complexity" << "\n";

}

for (int i = 1; i <= 3; i++) {

cout << "For " << i \* 10000 << " elements, ";

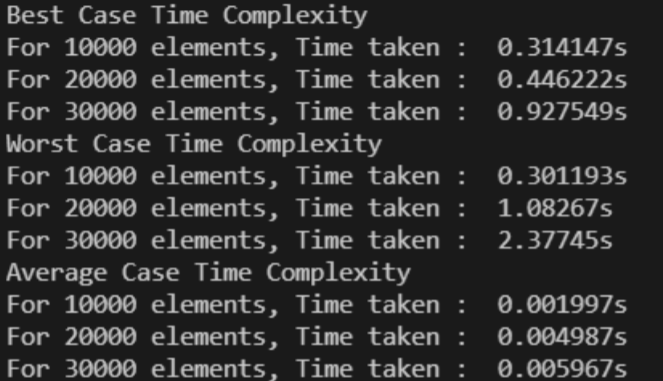
averageCase(i \* 10000);

}

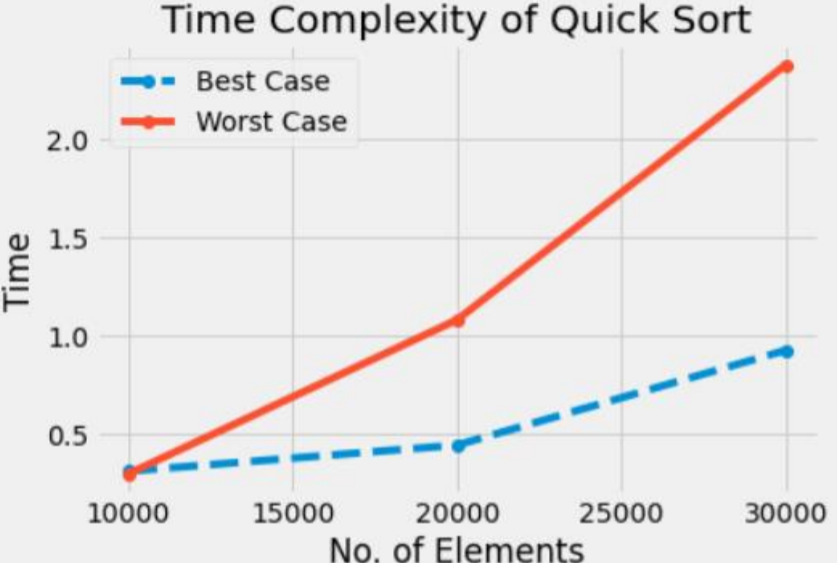
return 0;

}

**OUTPUT :-**



**RESULT AND ANALYSIS:-**

****

The running time of Quick sort depends whether Partition is balanced or not.

**Best case Partitioning**:If the procedure 'Partition' produces two regions of size n/2. The recurrence relation is then

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

And from case 2 of Master Theorem. T (n) = O(n lg n)

**Worst case Partitioning**:The worst-case occurs if given array A [1... n] is already sorted. The PARTITION (A, p, r) call always return p so successive calls to partition will split arrays of length n, n-1, n-2, . . ., 2 and running time proportional to n + (n-1) + (n-2) + . . . + 2 = [(n+2) (n-1)]/2 = O(n^2 ). The worst-case also occurs if A [1... n] starts out in reverse order.

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.

**SOURCE CODE:-**

#include <bits/stdc++.h>

using namespace std;

void bubbleSort(int \*arr, int n) {

typedef std::chrono::high\_resolution\_clock Time;

typedef std::chrono::milliseconds ms;

typedef std::chrono::duration fsec;

auto t0 = Time::now(); //Bubble Sort

bool stopFlag;

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

stopFlag = false;

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

if (arr[j] > arr[j + 1]) {

swap(arr[j], arr[j + 1]);

stopFlag = true;

}

}

if (stopFlag == false){

break;

}

}

auto t1 = Time::now();

fsec fs = t1 - t0;

cout << "Time taken : " << fs.count() << "s " << endl;

}

void bestCase(int n) { //Insert in sorted order

int \*arr = new int[n];

for (int i = 0; i < n; i++) {

arr[i] = i;

}

bubbleSort(arr, n);

}

void worstCase(int values) { //Insert in Decreasing order

int \*arr = new int[values];

for (int i = 0; i < values; i++) {

arr[i] = values - i;

}

bubbleSort(arr, values);

}

void averageCase(int values) {

int \*arr = new int[values];

for (int i = 0; i < values; i++) {

arr[i] = rand() % values;

}

bubbleSort(arr, values);

}

int main() {

cout << "Best Case Time Complexity" << "\n";

for (int i = 1; i <= 3; i++) {

cout << "For " << i \* 10000 << " elements, ";

bestCase(i \* 10000);

}

cout << "Worst Case Time Complexity" << "\n";

for (int i = 1; i <= 3; i++) {

cout << "For " << i \* 10000 << " elements, ";

worstCase(i \* 10000);

}

cout << "Average Case Time Complexity" << "\n";

for (int i = 1; i <= 3; i++) {

cout << "For " << i \* 10000 << " elements, ";

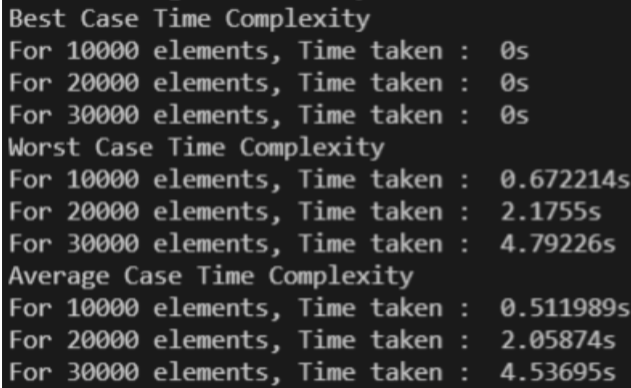
averageCase(i \* 10000);

}

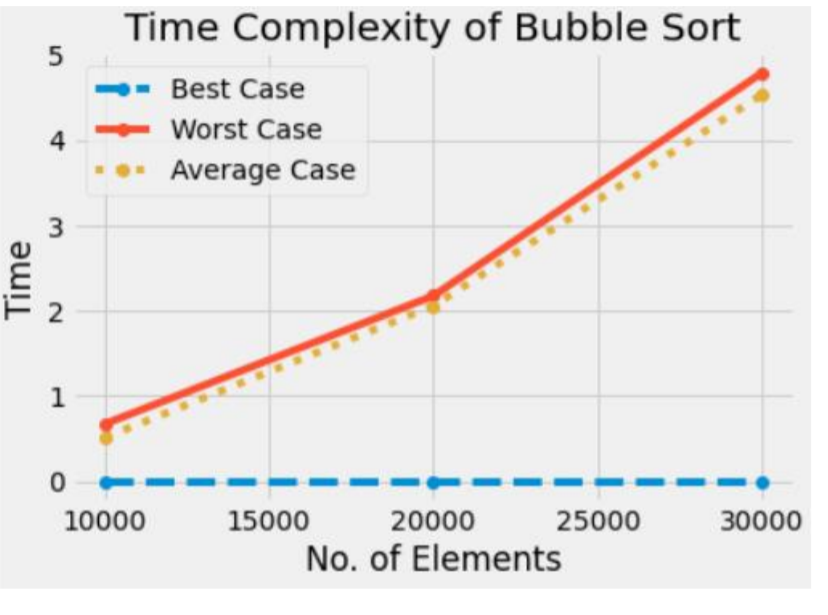
return 0;

}

**OUTPUT :-**

****

**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 the array is already sorted and its time complexity is O(n) (linear time). This sort takes just O(1) extra space.

1. **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.

**SOURCE CODE:-**

#include <bits/stdc++.h>

using namespace std;

void bucketSort(float \*arr, int n) {

typedef std::chrono::high\_resolution\_clock Time;

typedef std::chrono::milliseconds ms;

typedef std::chrono::duration fsec;

auto t0 = Time::now();

vector bucket[n];

for (int i = 0; i < n; i++) {

int bidx = n \* arr[i];

bucket[bidx].push\_back(arr[i]);

}

for (int i = 0; i < n; i++){

sort(bucket[i].begin(), bucket[i].end());

int idx = 0;

}

for (int i = 0; i < n; i++) {

for (int j = 0; j < bucket[i].size(); j++) {

arr[idx++] = bucket[i][j];

}

}

auto t1 = Time::now();

fsec fs = t1 - t0;

cout << "Time taken : " << fs.count() << "s " << endl;

}

void bestCase(int values) { //Insert in sorted order

float arr[values];

float val = values;

for (int i = 0; i < values;i++){

float x = i; arr[i] = 0 + (x / val);

}

bucketSort(arr, values);

}

void worstCase(int values) { //Insert in Decreasing order

float arr[values];

float val = values;

for (int i = 0; i < values; i++) {

float x = i;

arr[i] = 1 - (x / val);

}

bucketSort(arr, values);

}

void averageCase(int values) {

float \*arr = new float[values];

for (int i = 0; i < values; i++) {

arr[i] = ((double)rand() / (RAND\_MAX));

}

bucketSort(arr, values);

}

int main() {

cout << "Best Case Time Complexity"<< "\n";

for (int i = 1; i <= 3;i++) {

cout << "For " << i \* 10000 << " elements, ";

bestCase(i \* 10000);

}

cout << "Worst Case Time Complexity" << "\n";

for (int i = 1; i <= 3; i++) {

cout << "For " << i \* 5000 << " elements, ";

worstCase(i \* 5000); } cout << "Average Case Time Complexity" << "\n";

}

cout << "Worst Case Time Complexity" << "\n";

for (int i = 1; i <= 3; i++) {

cout << "For " << i \* 5000 << " elements, ";

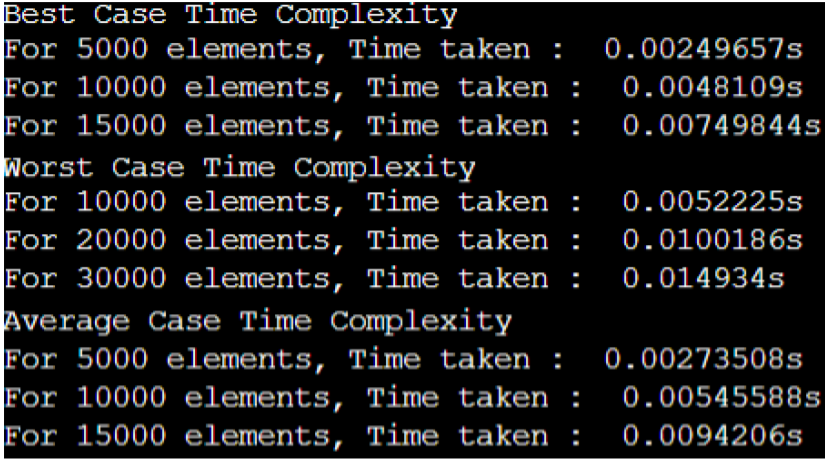
averageCase(i \* 5000);

}

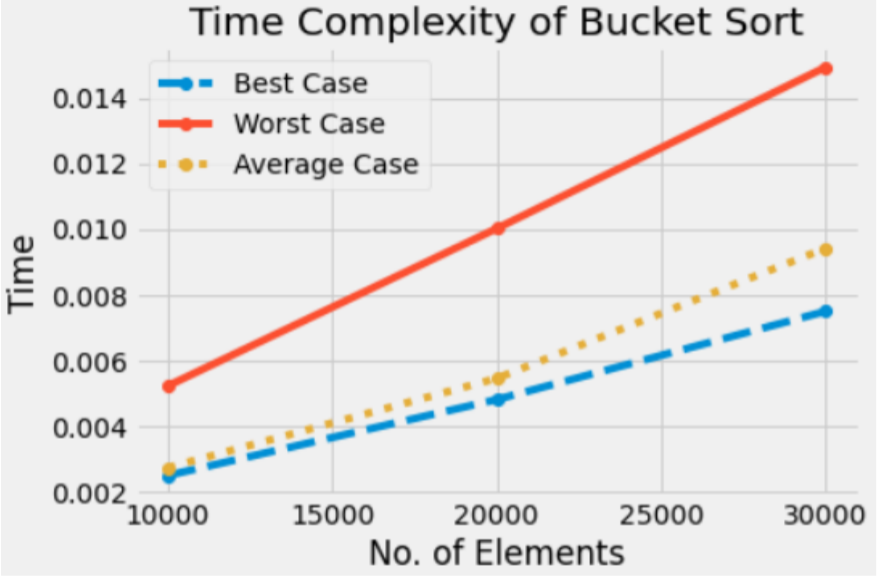
return 0;

}

**OUTPUT :-**

****

**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).

1. **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 the last digit (least significant digit). Then the result is again sorted by the second digit, continuing this process for all digits until we reach the most significant digit. We use counting sort to sort elements of every digit.

**SOURCE CODE:-**

#include <bits/stdc++.h>

using namespace std;

int getMax(int arr[], int n) {

int max = arr[0];

for (int i = 1; i < n; i++){

if (arr[i] > max) {

max = arr[i];

}

}

return max;

}

void count(int arr[], int n, int exp) {

int out[n];

int i, count[10] = {0};

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

count[(arr[i] / exp) % 10]++;

}

for (i = 1; i < 10; i++){

count[i] += count[i - 1];

}

for (i = n - 1; i >= 0; i--) {

out[count[(arr[i] / exp) % 10] - 1] = arr[i];

count[(arr[i] / exp) % 10]--;

}

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

arr[i] = out[i];

}

}

void radixSort(int arr[], int n) {

typedef std::chrono::high\_resolution\_clock Time;

typedef std::chrono::milliseconds ms;

typedef std::chrono::duration fsec;

auto t0 = Time::now();

int m = getMax(arr, n);

for (int i = 1; m / i > 0; i \*= 10) {

count(arr, n, i);

}

auto t1 = Time::now();

fsec fs = t1 - t0;

cout << "Time taken : " << fs.count() << "s " << endl;

}

void bestCase(int values) { //Insert in Decreasing order

int \*arr = new int[values];

for (int i = 0; i < values; i++) {

arr[i] = i;

}

radixSort(arr, values);

}

void worstCase(int values) { //Insert in Decreasing order

int \*arr = new int[values];

for (int i = 0; i < values; i++) {

arr[i] = values - i;

}

radixSort(arr, values);

}

void averageCase(int values) {

int \*arr = new int[values];

for (int i = 0; i < values; i++) {

arr[i] = rand() % values;

}radixSort(arr, values);

}

int main() {

cout << "Best Case Time Complexity" << "\n";

for (int i = 1; i <= 3; i++) {

cout << "For " << i \* 10000 << " elements, ";

bestCase(i \* 10000);

}

cout << "Worst Case Time Complexity" << "\n";

for (int i = 1; i <= 3; i++) {

cout << "For " << i \* 10000 << " elements, ";

worstCase(i \* 10000);

}

cout << "Average Case Time Complexity" << "\n";

for (int i = 1; i <= 3; i++) {

cout << "For " << i \* 10000 << " elements, ";

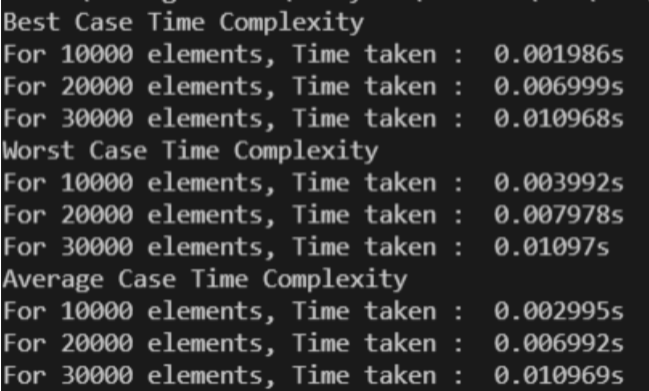
averageCase(i \* 10000);

}

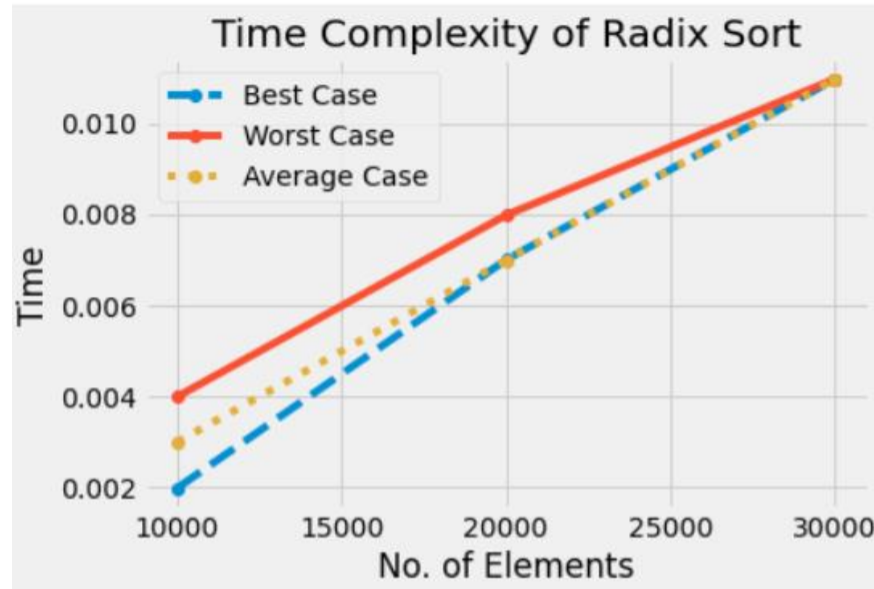
return 0;

}

**OUTPUT :-**

****

**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 .

1. **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, …, 2k – 1 ;

Knuth’s increments: 1, 4, 13, …, (3k – 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.

**SOURCE CODE:-**

#include <bits/sydc++.h>

using namespace std;

int shellSort(int arr[], int n) {

typedef std::chrono::high\_resolution\_clock Time;

typedef std::chrono::milliseconds ms;

typedef std::chrono::duration fsec;

auto t0 = Time::now();

for (int spc = n / 2; spc > 0; spc /= 2) {

for (int i = spc; i < n; i += 1) {

int tmp = arr[i];

int j;

for (j = i; j >= spc && arr[j - spc] > tmp; j -= spc){

arr[j] = arr[j - spc]; arr[j] = tmp;

}

}

}

auto t1 = Time::now();

fsec fs = t1 - t0;

cout << "Time taken : " << fs.count() << "s " << endl;

return 0;

}

void bestCase(int n) { //Insert in sorted order

int \*arr = new int[n];

for (int i = 0; i < n; i++) {

arr[i] = i;

}

shellSort(arr, n);

}

void worstCase(int n) { //Insert in Decreasing order

int \*arr = new int[n];

for (int i = 0; i < n; i++) {

arr[i] = n - i;

}

shellSort(arr, n);

}

void averageCase(int n) {

int \*arr = new int[n];

for (int i = 0; i < n; i++) {

arr[i] = (rand() % static\_cast(n - 1));

}

shellSort(arr, n);

}

int main() {

cout << "Best Case Time Complexity" << "\n";

for (int i = 1; i <= 3; i++) {

cout << "For " << i \* 10000 << " elements, ";

bestCase(i \* 10000);

}

cout << "Worst Case Time Complexity" << "\n";

for (int i = 1; i <= 3; i++) {

cout << "For " << i \* 10000 << " elements, ";

worstCase(i \* 10000);

}

cout << "Average Case Time Complexity" << "\n";

for (int i = 1; i <= 3; i++) {

cout << "For " << i \* 10000 << " elements, ";

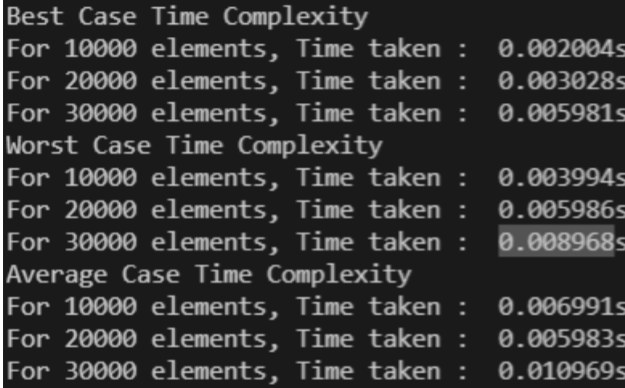
averageCase(i \* 10000);

}

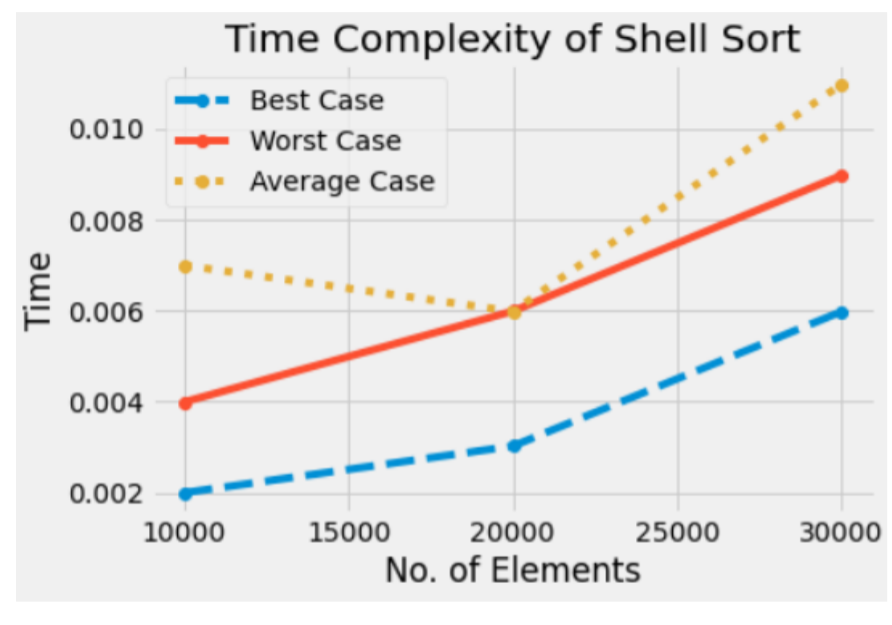
return 0;

}

**OUTPUT :-**

****

**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) .

1. **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.

**SOURCE CODE:-**

#include <bits/stdc++.h>

using namespace std;

void selectionSort(int \*arr, int n) {

typedef std::chrono::high\_resolution\_clock Time;

typedef std::chrono::duration fsec;

auto t0 = Time::now(); //SelectionSort

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

int minIndex = i;

for (int j = i + 1; j < n; j++) {

if (arr[j] < arr[minIndex]) {

minIndex = j;

}

swap(arr[minIndex], arr[i]);

}

}

auto t1 = Time::now();

fsec fs = t1 - t0;

cout << "Time taken : " << fs.count() << "s" << endl;

}

void bestCase(int values) { //Insert in Decreasing order

int \*arr = new int[values];

for (int i = 0; i < values; i++) {

arr[i] = i;

}

selectionSort(arr, values);

}

void worstCase(int values) { //Insert in Decreasing order

int \*arr = new int[values];

for (int i = 0; i < values; i++) {

arr[i] = values - i;

}

selectionSort(arr, values);

}

void averageCase(int values) {

int \*arr = new int[values];

for (int i = 0; i < values; i++) {

arr[i] = rand() % values;

}

selectionSort(arr, values);

}

int main() {

cout << "Best Case Time Complexity" << "\n";

for (int i = 1; i <= 3; i++) {

cout << "For " << i \* 10000 << " elements, ";

bestCase(i \* 10000);

}

cout << "Worst Case Time Complexity" << "\n";

for (int i = 1; i <= 3; i++) {

cout << "For " << i \* 10000 << " elements, ";

worstCase(i \* 10000);

}

cout << "Average Case Time Complexity"<< "\n";

for (int i = 1; i <= 3; i++) {

cout << "For " << i \* 10000 << " elements, ";

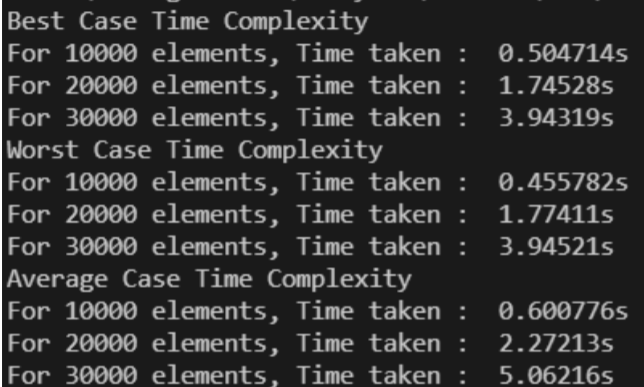
averageCase(i \* 10000);

}

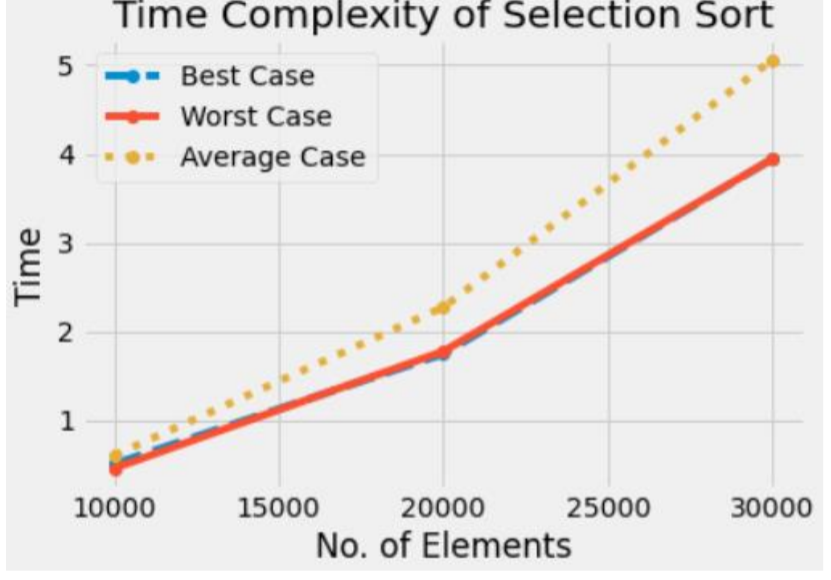
return 0;

}

**OUTPUT :-**

****

**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 (n^2).

1. **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.

**SOURCE CODE:-**

#include <bits/stdc++.h>

using namespace std;

void heapify(int arr[], int n, int i) {

int largest = i;

int l = 2 \* i + 1;

int r = 2 \* i + 2;

if (l < n && arr[l] > arr[largest]){

largest = i;

}

if (r < n && arr[r] > arr[largest]){

largest = r;

}

if (largest != i) {

swap(arr[i], arr[largest]);

heapify(arr, n, largest);

}

}

void heapSort(int arr[], int n) { // main function to do heap sort

typedef std::chrono::high\_resolution\_clock Time;

typedef std::chrono::milliseconds ms;

typedef std::chrono::duration fsec;

auto t0 = Time::now();

for (int i = n / 2 - 1; i >= 0; i--){

heapify(arr, n, i);

}

for (int i = n - 1; i > 0; i--) {

swap(arr[0], arr[i]);

heapify(arr, i, 0);

}

auto t1 = Time::now();

fsec fs = t1 - t0;

cout << "Time taken : " << fs.count() << "s " << endl;

}

void bestCase(int n) { //Insert in sorted order

int \*arr = new int[n];

for (int i = 0; i < n; i++) {

arr[i] = i;

}

heapSort(arr, n);

}

void worstCase(int n) { //Insert in Decreasing order

int \*arr = new int[n];

for (int i = 0; i < n; i++) {

arr[i] = n - i;

}

heapSort(arr, n);

}

void averageCase(int n){

int \*arr = new int[n];

for (int i = 0; i < n; i++) {

arr[i] = rand() % n;

}

heapSort(arr, n);

}

int main() {

cout << "Best Case Time Complexity" << "\n";

for (int i = 1; i <= 3; i++) {

cout << "For " << i \* 10000 << " elements, ";

bestCase(i \* 10000);

}

cout << "Worst Case Time Complexity" << "\n";

for (int i = 1; i <= 3; i++) {

cout << "For " << i \* 10000 << " elements, ";

worstCase(i \* 10000);

}

cout << "Average Case Time Complexity" << "\n";

for (int i = 1; i <= 3; i++) {

cout << "For " << i \* 10000 << " elements, ";

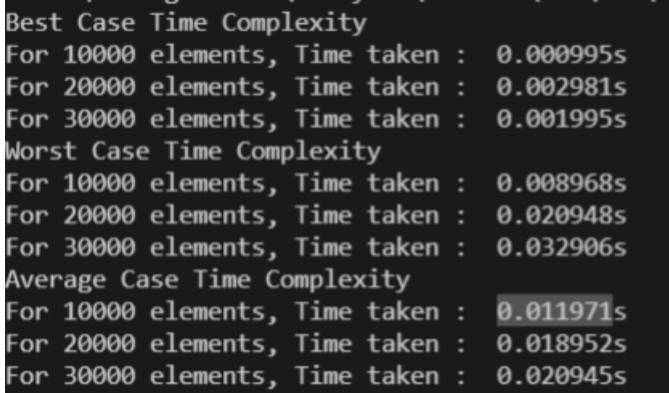
averageCase(i \* 10000);

}

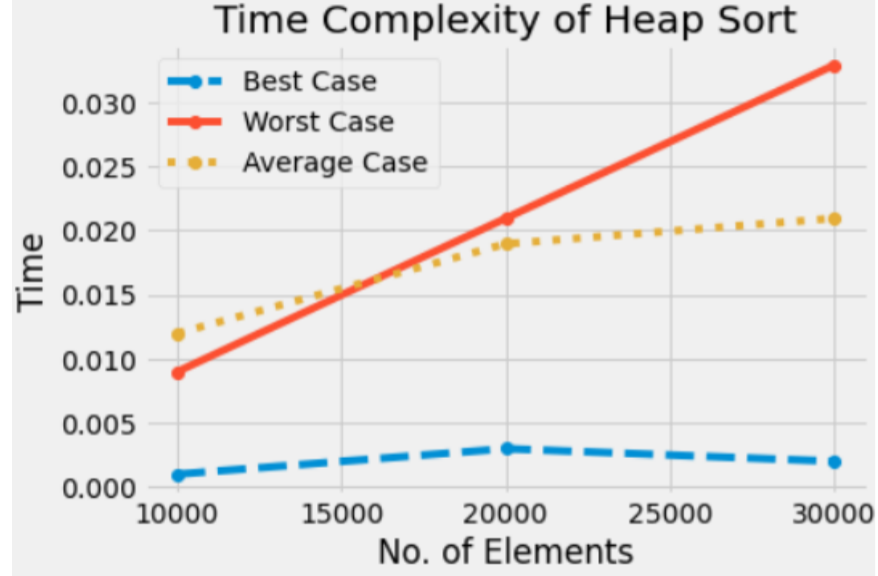
return 0;

}

**OUTPUT :-**

****

**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=2h+1 -1 nodes in an almost complete binary tree of height h, so Heapify runs in time O (ln n). BuildHeap calls Heapify n/2 times, so it takes O (n h) = O (n ln n).

**EXPERIMENT-02**

**AIM:-** To implement following algorithms and analyse its time complexity :-

a. Linear Search

b. Binary Search

1. **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.

**SOURCE CODE:-**

#include <bits/stdc++.h>

using namespace std;

void LinearSearch(int arr[], int n, int key) {

typedef std::chrono::high\_resolution\_clock Time;

typedef std::chrono::milliseconds ms;

typedef std::chrono::duration fsec;

auto t0 = Time::now();

for (int i = 0; i < n; i++) {

if (arr[i] == key) {

break;

}

}

auto t1 = Time::now();

fsec fs = t1 - t0;

ms d = std::chrono::duration\_cast(d);

cout << "Time taken : " << fs.count() << "s " << endl;

cout<< d.count() << "ms "

}

void bestCase(int values) { //Insert in Decreasing order

int \*arr = new int[values];

for (int i = 0; i < values; i++) {

arr[i] = i;

}

int key = 0 + rand() % (values + 1);

LinearSearch(arr, values, key);

}

void worstCase(int values) { //Insert in Decreasing order

int \*arr = new int[values];

for (int i = 0; i < values; i++) {

arr[i] = values - i;

}

int key = 0 + rand() % (values + 1);

LinearSearch(arr, values, key);

}

void averageCase(int values) {

int \*arr = new int[values];

for (int i = 0; i < values; i++) {

arr[i] = rand() % values;

}

int key = 0 + rand() % (values + 1);

LinearSearch(arr, values, key);

}

int main() {

cout << "Best Case Time Complexity"<< "\n";

for (int i = 1; i <= 3; i++) {

cout << "For " << i \* 100000 << " elements, ";

bestCase(i \* 100000);

}

cout << "Worst Case Time Complexity" << "\n";

for (int i = 1; i <= 3; i++) {

cout << "For " << i \* 100000 << " elements, ";

worstCase(i \* 100000);

}

cout << "Average Case Time Complexity" << "\n";

for (int i = 1; i <= 3; i++) {

cout << "For " << i \* 100000 << " elements, ";

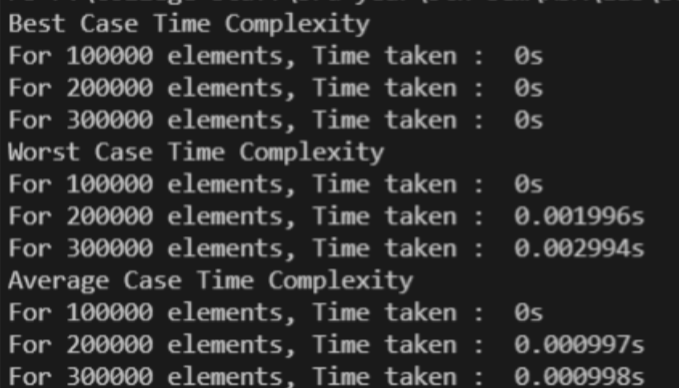
averageCase(i \* 100000);

}

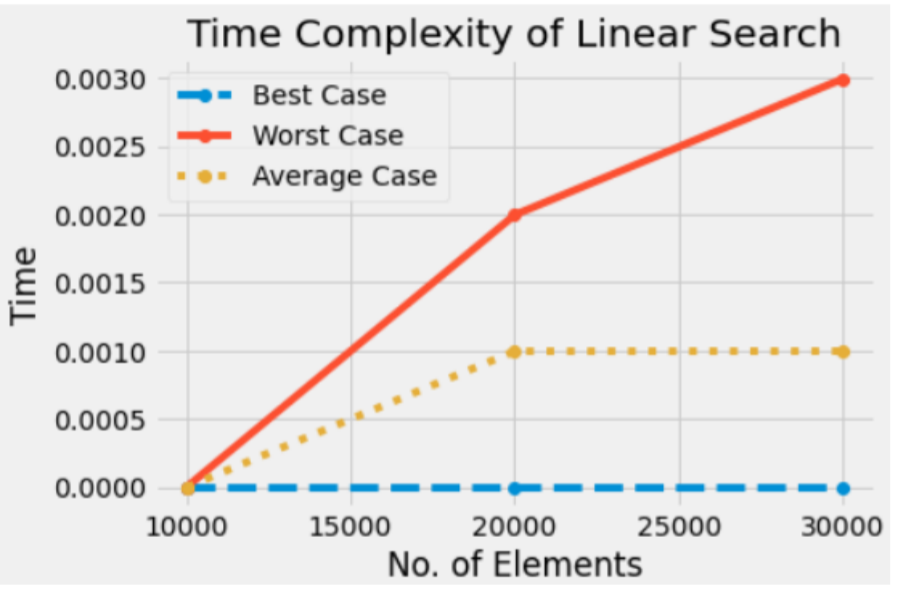
return 0;

}

**OUTPUT :-**

****

**RESULT AND ANALYSIS:-**

****

1. **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.

**SOURCE CODE:-**

#include <bits/stdc++.h>

using namespace std;

bool BinarySearch(int arr[], int s, int e, int key) {

if (e >= s) {

int m = s + (e - s) / 2;

if (arr[m] == key) {

return true;

}

if (arr[m] < key) {

BinarySearch(arr, m + 1, e, key);

}

else {

BinarySearch(arr, s, m - 1, key);

}

}

else {

return false;

}

}

void bestCase(int values) {

typedef std::chrono::high\_resolution\_clock Time;

typedef std::chrono::milliseconds ms;

typedef std::chrono::duration fsec;

auto t0 = Time::now(); //Insert in Decreasing order

int \*arr = new int[values];

for (int i = 0; i < values; i++) {

arr[i] = i;

}

int key = 0 + rand() % (values + 1);

BinarySearch(arr, 0, values, key);

auto t1 = Time::now();

fsec fs = t1 - t0;

ms d = std::chrono::duration\_cast(d);

cout << "Time taken : " << fs.count() << "s " << endl;

}

void worstCase(int values) {

typedef std::chrono::high\_resolution\_clock Time;

typedef std::chrono::milliseconds ms;

typedef std::chrono::duration fsec;

auto t0 = Time::now(); //Insert in Decreasing order

int \*arr = new int[values];

for (int i = 0; i < values; i++) {

arr[i] = values - i;

}

int key = 0 + rand() % (values + 1);

BinarySearch(arr, 0, values, key);

auto t1 = Time::now();

fsec fs = t1 - t0;

ms d = std::chrono::duration\_cast(d);

cout << "Time taken : " << fs.count() << "s " << endl;

}

void averageCase(int values) {

typedef std::chrono::high\_resolution\_clock Time;

typedef std::chrono::milliseconds ms;

typedef std::chrono::duration fsec;

auto t0 = Time::now();

int \*arr = new int[values];

for (int i = 0; i < values; i++) {

arr[i] = rand() % values;

}

int key = 0 + rand() % (values + 1);

BinarySearch(arr, 0, values, key);

auto t1 = Time::now();

fsec fs = t1 - t0;

ms d = std::chrono::duration\_cast(d);

cout << "Time taken : " << fs.count() << "s " << endl;

}

int main() {

cout << "Best Case Time Complexity" << "\n";

for (int i = 1; i <= 3; i++) {

cout << "For " << i \* 10000 << " elements, ";

bestCase(i \* 10000);

}

cout << "Worst Case Time Complexity" << "\n";

for (int i = 1; i <= 3; i++) {

cout << "For " << i \* 10000 << " elements, ";

worstCase(i \* 10000);

}

cout << "Average Case Time Complexity" << "\n";

for (int i = 1; i <= 3; i++) {

cout << "For " << i \* 10000 << " elements, ";

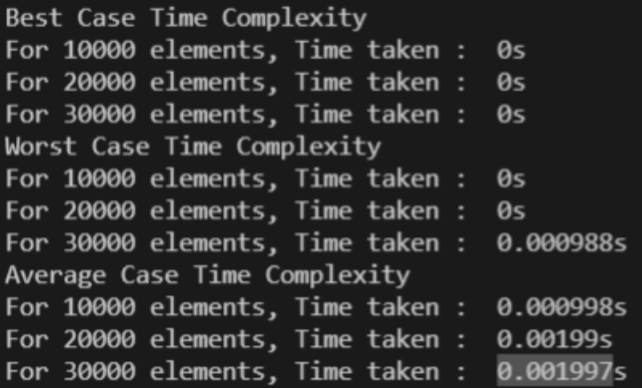
averageCase(i \* 10000);

}

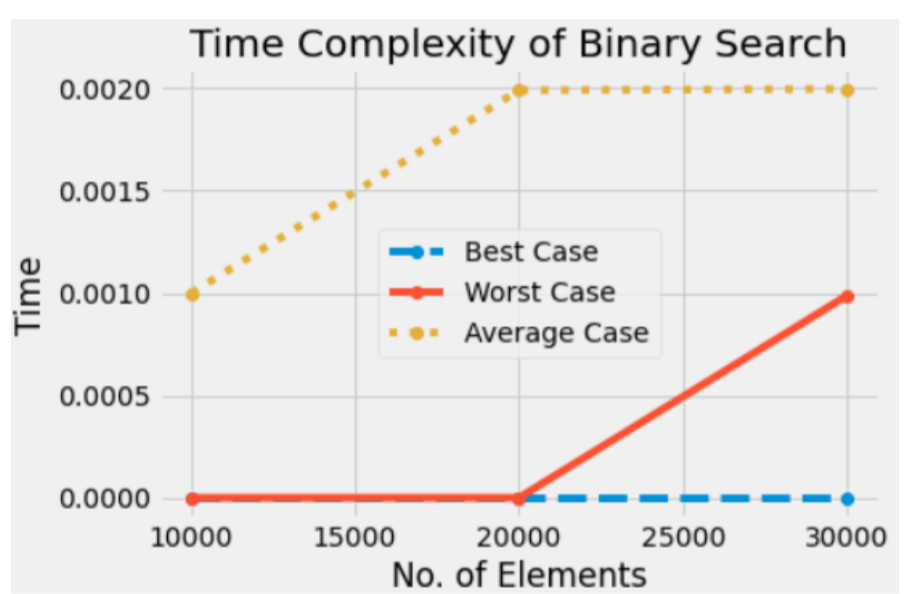
return 0;

}

**OUTPUT :-**

****

**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.

**EXPERIMENT-03**

**AIM:-** To analyse matrix multiplication and analyse its time complexity.

Following is simple Divide and Conquer method to multiply two square matrices.

1) Divide matrices A and B in 4 sub-matrices of size N/2 x N/2 .

2) Calculate following values recursively. ae + bg, af + bh, ce + dg and cf + dh.

**SOURCE CODE:-**

#include <bits/stdc++.h>

using namespace std;

main() {

int a[2][2], b[2][2], c[2][2];

int m1, m2, m3, m4, m5, m6, m7, i, j;

cout << "Enter the elements of Matrix A : " << endl;

for (i = 0; i < 2; i++) {

for (j = 0; j < 2; j++) {

cin >> a[i][j];

}

}

cout << "Enter the elements of Matrix B : " << endl;

for (i = 0; i < 2; i++) {

for (j = 0; j < 2; j++) {

cin >> b[i][j];

}

}

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]);

c[0][0] = m1 + m4 - m5 + m7;

c[0][1] = m3 + m5;

c[1][0] = m2 + m4;

c[1][1] = m1 - m2 + m3 + m6;

cout << "After Multiplication :" << endl;

for (i = 0; i < 2; i++) {

for (j = 0; j < 2; j++) {

cout << c[i][j] << " ";

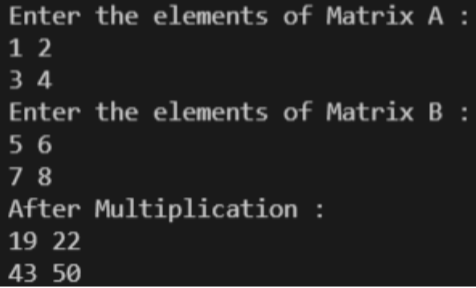
}

cout << "\n";

}

}

**OUTPUT :-**

****

**RESULT AND ANALYSIS:-**

Clearly, the space complexity of this procedure is Ο (n^2). Since the tables m and s require Ο (n^2) space. As far as the time complexity is concerned, 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 . Therefore, the running time of this procedure is Ο (n^3).

**EXPERIMENT-04**

**AIM:-** To implement Longest Common Subsequence problem and analyse its time complexity.

**Subsequence**: Let us consider a sequence S = . A sequence Z = 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 Xand 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.

**Dynamic Programming**: Let X = < x1, x2, x3,…, xm > and Y = < y1, y2, y3,…, yn > 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 an optimal solution.

**SOURCE CODE:-**

#include <bits/stdc++.h>

using namespace std;

int LCS(string s1, string s2, int i, int j) {

if (i == s1.length() or j == s2.length()) {

return 0;

}

if (s1[i] == s2[j]) {

return 1 + LCS(s1, s2, i + 1, j + 1);

}

int call1 = LCS(s1, s2, i + 1, j);

int call2 = LCS(s1, s2, i, j + 1);

return max(call1, call2);

}

int LCS\_withDP(string s1, string s2, int i, int j, vector<vector<int>>

&dp){

if (i == s1.length() or j == s2.length()){

return 0;

}

if (dp[i][j] != -1){

return dp[i][j];

}

if (s1[i] == s2[j]){

return dp[i][j] = 1 + LCS\_withDP(s1, s2, i + 1, j + 1, dp);

}

int call1 = LCS\_withDP(s1, s2, i + 1, j, dp);

int call2 = LCS\_withDP(s1, s2, i, j + 1, dp);

return dp[i][j] = max(call1, call2);

}

int main(){

string s1 = "ABCD";

string s2 = "ABEDG";

int l1 = s1.length();

int l2 = s2.length();

vector<vector<int>> dp(l1, vector<int>(l2, -1));

cout << "LCS is of length --> " << endl;

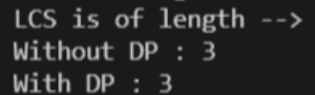
cout << "Without DP : " << LCS(s1, s2, 0, 0) << endl;

cout << "With DP : " << LCS\_withDP(s1, s2, 0, 0, dp)<<endl;

return 0;

}

**OUTPUT :-**

****

**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.

**EXPERIMENT-05**

**AIM:-** To implement Optimal Binary Search problem and analyse its time complexity.

In computer science, 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 and 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 focuses 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 k1,k2,…,kn) in sorted order (so that k1<k2<…<kn),and we wish to build a binary search tree from these keys. For each ki ,we have a probability pi that a search will be for ki.
* In contrast, some searches may be for values not in ki, and so we also have n+1 “dummy keys” d0,d1,…,dn representing not in ki.
* In particular, d0 represents all values less than k1, and dn represents all values greater than kn, and for i=1,2,…,n-1, the dummy key di represents all values between ki and ki+1.

**SOURCE CODE:-**

#include <bits/stdc++.h>

using namespace std;

int sum(int freq[], int i, int j) {

int sum1 = 0;

for (int k = i; k <= j; k++){

sum1 += freq[k];

}

return sum1;

}

int OBST(int arr[], int freq[], int n) {

int cost[n][n];

for (int i = 0; i < n; i++) {

cost[i][i] = freq[i];

}

for (int x = 2; x <= n; x++){

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

int j = i + x - 1;

cost[i][j] = INT\_MAX;

for (int r = i; r <= j; r++){

int c = 0;

if(r>i){

c += cost[i][r - 1];

}

if(r<j){

c += cost[r + 1][j];

}

c+= sum(freq, i, j);

if (c < cost[i][j]){

cost[i][j] = c;

}

}

}

}

return cost[0][n - 1];

}

int main(){

int arr[] = {5, 9, 12, 30};

int freq[] = {21, 10, 40, 30};

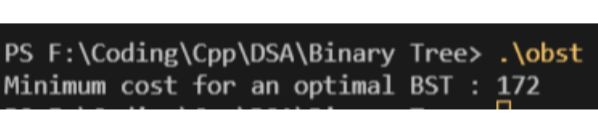
int n = sizeof(arr) / sizeof(arr[0]);

cout << "Minimum cost for an optimal BST : " << OBST(arr,

freq, n);

}

**OUTPUT :-**

****

**RESULT AND ANALYSIS:-**

Every time we work on an entry e[i, j] with j − i = k, we know that all the entries e[i0 , j0 ] with j0 − i0 < k have already been computed. Note that the recursive formula we use to compute e[i, j] only involves entries e[i0 , j0 ] with j0 − i0 < 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 O(n^3 ).

**EXPERIMENT-06**

**AIM:-** To implement Huffman Coding and analyse 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 a prefix of code assigned to any other character. This is how Huffman Coding makes sure that there is no ambiguity when decoding the generated bit stream.

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 a 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”.

**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 printHC(struct MinHeapNode \*root, string str) {

if (!root){

return;

}

if (root->data != '$'){

cout << root->data << ": " << str << "\n";

}

printHC(root->left, str + "0");

printHC(root->right, str + "1");

}

void HuffmanCoding(char data[], int freq[], int size) {

struct MinHeapNode \*left, \*right, \*top;

priority\_queue, 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);

}

printHC(minHeap.top(), "");

}

int main() {

char arr[] = {'a', 'b', 'c', 'd', 'e', 'f'};

int freq[] = {2, 5, 11, 15, 21, 39};

int size = sizeof(arr) / sizeof(arr[0]);

typedef std::chrono::high\_resolution\_clock Time;

typedef std::chrono::milliseconds ms;

typedef std::chrono::duration fsec;

auto t0 = Time::now();

HuffmanCoding(arr, freq, size);

auto t1 = Time::now();

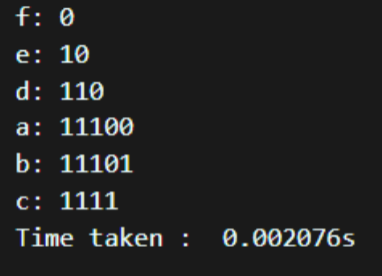
fsec fs = t1 - t0;

cout << "Time taken : " << fs.count() << "s " << endl;

return 0;

}

**OUTPUT :-**

****

**RESULT AND ANALYSIS:-**

If there are n nodes, extractMin() is called 2\*(n – 1) times and extractMin() takes O(log n) time as it calls minHeapify(). So, overall complexity is O (n log n) where n specifies the number of unique characters.

**EXPERIMENT-07**

**AIM:-** To implement Dijkstra’s algorithm and analyse its time complexity.

Dijkstra's algorithm solves the single-source shortest-paths problem on a weighted, directed graph G = (V, E) for the case in which all edge weights are nonnegative. In this section, therefore, we assume that w(u, v) >= 0 for each edge (u, v) belongs to E.

This algorithm maintains a set S of vertices whose final shortest-path weights from the source s have already been determined. That is, for all vertices v that belong to S, we have d[v] = delta(s, v).The algorithm repeatedly selects the vertex u belongs to V - S with the minimum shortest path estimate, inserts u into S, and relaxes all edges leaving u. In the following implementation, we maintain a priority queue Q that contains all the vertices in V - S, keyed by their d values. The implementation assumes that graph G is represented by adjacency lists.

**SOURCE CODE:-**

#include <bits/stdc++.h>

using namespace std;

#define V 6

int minDistance(int dist[], bool sptSet[]) {

int min = INT\_MAX, min\_index;

for (int v = 0; v < V; v++){

if (sptSet[v] == false && dist[v] <= min){

min = dist[v], min\_index = v;

}

}

return min\_index;

}

void printSolution(int dist[]) {

cout << "Vertex \t Distance from Source" << endl;

for (int i = 0; i < V; i++){

cout << i << " \t\t" << dist[i] << endl;

}

}

void dijkstra(int graph[V][V], int src) {

int dist[V];

bool sptSet[V];

for (int i = 0; i < V; i++){

dist[i] = INT\_MAX, sptSet[i] = false;

}

dist[src] = 0;

for (int count = 0; count < V - 1; count++) {

int u = minDistance(dist, sptSet);

sptSet[u] = true;

for (int v = 0; v < V; v++){

if (!sptSet[v] && graph[u][v] && dist[u] != INT\_MAX && dist[u] + graph[u][v] < dist[v]){

dist[v] = dist[u] + graph[u][v]; } printSolution(dist);

}

}

}

}

int main() {

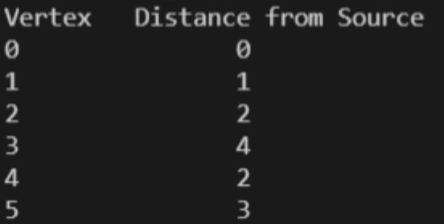
int graph[V][V] = { {0, 1, 2, 0, 0, 0}, {1, 0, 0, 5, 1, 0}, {2, 0, 0, 2, 3, 0}, {0, 5, 2, 0, 2, 2}, {0, 1, 3, 2, 0, 1}, {0, 0, 0, 2, 1, 0} };

dijkstra(graph, 0);

return 0;

}

**OUTPUT :-**

****

**RESULT AND ANALYSIS:-**

Every time the main loop executes, one vertex is extracted from the queue. Assuming that there are V vertices in the graph, the queue may contain O(V) vertices. Therefore the total run time is O (V lg V + E lg V), which is O(E lg V) because V is O(E) assuming a connected graph.

**EXPERIMENT-08**

**AIM:-** To implement Bellman Ford algorithm and analyse its time complexity.

Bellman Ford algorithm helps us find the shortest path from a vertex to all other vertices of a weighted graph.

It is similar to Dijkstra's algorithm but it can work with graphs in which edges can have negative weights.

Bellman Ford algorithm works by overestimating the length of the path from the starting vertex to all other vertices. Then it iteratively relaxes those estimates by finding new paths that are shorter than the previously overestimated paths.

By doing this repeatedly for all vertices, we can guarantee that the result is optimized.

**SOURCE CODE:-**

#include <bits/stdc++.h>

struct Edge {

int u;

int v;

int w;

};

struct Graph {

int V;

int E;

struct Edge\* edge;

};

struct Graph\* createGraph(int V, int E) {

struct Graph\* graph = new Graph;

graph->V = V;

graph->E = E;

graph->edge = new Edge[E];

return graph;

}

printArr(int arr[], int size) {

int i;

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

printf("%d ", arr[i]);

}

printf("\n");

}

void BellmanFord(struct Graph\* graph, int u) {

int V = graph->V;

int E = graph->E;

int dist[V];

for (int i = 0; i < V; i++){

dist[i] = INT\_MAX;

}

dist[u] = 0;

for (int i = 1; i <= V - 1; i++) {

for (int j = 0; j < E; j++) {

int u = graph->edge[j].u;

int v = graph->edge[j].v;

int w = graph->edge[j].w;

if (dist[u] != INT\_MAX && dist[u] + w < dist[v]){

dist[v] = dist[u] + w;

}

}

}

for (int i = 0; i < E; i++) {

int u = graph->edge[i].u;

int v = graph->edge[i].v;

int w = graph->edge[i].w;

if (dist[u] != INT\_MAX && dist[u] + w < dist[v]) {

printf("Graph contains negative w cycle");

return;

}

}

printArr(dist, V);

return;

}

int main() {

int V = 5;

int E = 8;

struct Graph\* graph = createGraph(V, E);

graph->edge[0].u = 0;

graph->edge[0].v = 1;

graph->edge[0].w = 5;

graph->edge[1].u = 0;

graph->edge[1].v = 2;

graph->edge[1].w = 4;

graph->edge[2].u = 1;

graph->edge[2].v = 3;

graph->edge[2].w = 3;

graph->edge[3].u = 2;

graph->edge[3].v = 1;

graph->edge[3].w = 6;

graph->edge[4].u = 3;

graph->edge[4].v = 2;

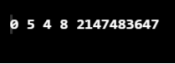
graph->edge[4].w = 2;

BellmanFord(graph, 0);

return 0;

}

**OUTPUT :-**

****

**RESULT AND ANALYSIS:-**

The Bellman-Ford algorithm is an algorithm that calculates the shortest paths in a weighted digraph from one source vertex to all other vertices. Bellman-Ford is also simpler than Dijkstra and suites well for distributed systems. But the time complexity of Bellman-Ford is O(VE), which is more than Dijkstra.

where V is a number of vertices and E is a number of edges. For a complete graph with n vertices, V = n, E = O(n^2). So overall time complexity becomes O(n^3).

**EXPERIMENT-09**

**AIM:-** To implement following algorithms and analyse its time complexity :-

a. Naive String Matching algorithm

b. Rabin Karp algorithm

c. Knuth Morris Pratt algorithm

1. **Naive String Matching Algorithm**

The naive string-matching procedure can be interpreted graphically as sliding a "template" containing the pattern over the text, noting for which shifts all of the characters on the template equal the corresponding characters in the text. This algorithm finds all valid shifts using a loop that checks the condition P[1 . . m] = T[s + 1 . . s + m] for each of the n - m + 1 possible values of s, where P is the pattern to be found in Text T.

**SOURCE CODE:-**

#include <bits/stdc++.h>

using namespace std;

void search(char \*pat, char \*txt) {

int M = strlen(pat);

int N = strlen(txt);

for (int i = 0; i <= N - M; i++) {

int j;

for (j = 0; j < M; j++) {

if (txt[i + j] != pat[j]){

break;

}

}

if (j == M) {

cout << "Match found at index : " << i << endl;

}

}

}

int main() {

char txt[] = "AABCAACAADAABCAAADABCAA";

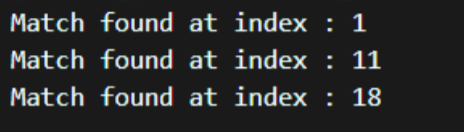
char pat[] = "ABC";

search(pat, txt);

return 0;

}

**OUTPUT :-**

****

**RESULT AND ANALYSIS:-**

This Procedure takes time ((n - m + 1) m) in the worst case. For each of the n - m + 1 possible values of the shift s, the implicit loop on line 4 to compare corresponding characters must execute m times to validate the shift. The worst-case running time is thus ((n - m + 1) m), which is (n2) if m = n/2 .

1. **Rabin Karp Algorithm**

Given a pattern P [1 . . m], we let p denote its corresponding decimal value. In a similar manner, given a text T[1 . . n], we let ts denote the decimal value of the length substring T[s + 1 . . s + m], for s = 0, 1, . . . , n - m. Certainly, ts = p if and only if T[s + 1 . . s + m] = P[1 . . m]; thus, s is a valid shift if and only if ts = p. If we could compute p in time O(m) and all of the ti values in a total of O(n) time, then we could determine all valid shifts in time O(n) by comparing p with each of the ts's.

* We can compute p in time O(m) using Horner's rule :
* p = P[m] + 10 (P[m - 1] + 10(P[m - 2] + . . . + 10(P[2] + 10P[1]) . . . )).
* The value t0 can be similarly computed from T[1 . . m] in time O(m).
* To compute the remaining values t1, t2, . . . , tn-m in time O(n - m), it suffices to observe that ts + 1 can be computed from ts in constant time, since.
* ts + 1 = 10(ts - 10m - 1T[s + 1]) + T[s + m + 1]. …….(1)

For example, if m= 5 and ts = 31415, then we wish to remove the high-order digit T[s + 1] = 3 and bring in the new low-order digit (suppose it is T[s + 5 + 1] = 2) to obtain ts+1 = 10(31415 - 10000.3) + 2= 14152 .

Subtracting 10m-1 T[s+1] removes the high-order digit from ts, multiplying the result by 10 shifts the number left one position, and adding T[s + m + 1] brings in the appropriate low-order digit. If the constant 10m-1 is pre computed (which can be done in time O(1g m) , although for this application a straightforward O(m) method is quite adequate), then each execution of equation(1) takes a constant number of arithmetic operations. Thus, p and t0, t1, . . , tn-m can all be computed in time O(n + m), and we can find all occurrences of the pattern P[1 . . m] in the text T[1 . . n] in time O(n + m). The only difficulty with this procedure is that p and ts may be too large to work with conveniently. If P contains m characters, then assuming that each arithmetic operation on p (which is m digits long) takes "constant time" is unreasonable. So compute p and the ts's modulo a suitable modulus q. Since the computation of p, t0, and the recurrence (1) can all be performed modulo q, we see that p and all the ts's can be computed modulo q in time O(n + m). The modulus q is typically chosen as a prime such that 10q just fits within one computer word, which allows all of the necessary computations to be performed with single-precision arithmetic.

**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;

int t = 0;

int h = 1;

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

h = (h \* d) % q;

}

for (i = 0; i < M; i++) {

p = (d \* p + pat[i]) % q;

t = (d \* t + txt[i]) % q;

}

for (i = 0; i <= N - M; i++) {

if (p == t) {

bool flag = true;

for (j = 0; j < M; j++) {

if (txt[i + j] != pat[j]) {

flag = false;

break;

}

}

if (j == M){

cout << "Match found at index " << i << endl;

}

}

if (i < N - M) {

t = (d \* (t - txt[i] \* h) + txt[i + M]) % q;

if (t < 0) t = (t + q);

}

}

int main() {

char txt[] = "PREM AGARWAL";

char pat[] = "AGARWAL";

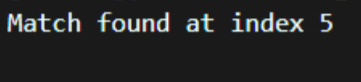
int q = 101;

search(pat, txt, q);

return 0;

}

**OUTPUT :-**

****

**RESULT AND ANALYSIS:-**

The running time of this algorithm is ((n - m + 1)m) in the worst case, since (like the naive string-matching algorithm) the Rabin-Karp algorithm explicitly verifies every valid shift. If P = am and T = an , then the verifications take time ((n - m + 1)m), since each of the n - m + 1 possible shifts is valid. (Note also that the computation of d m-1 mod q on line 3 and the loop on lines 6-8 take time O(m) = O((n - m + 1 )m).

1. **Knuth Morris Pratt Algorithm**

At a high level, the KMP algorithm is similar to the naive algorithm: it considers shifts in order from 1 to n−m, and determines if the pattern matches at that shift. The difference is that the KMP algorithm uses information gleaned from partial matches of the pattern and text to skip over shifts that are guaranteed not to result in a match. Suppose that, starting with the pattern aligned underneath the text at the leftmost end, we repeatedly “slide” the pattern to the right and attempt to match it with the text. Let’s look at some examples of how sliding can be done.

**SOURCE CODE:-**

#include <bits/stdc++.h>

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];

computeLPSArray(pat, M, lps);

int i = 0;

int j = 0;

while (i < N) {

if (pat[j] == txt[i]) {

j++;

i++;

}

if (j == M) {

printf("Match pattern at index %d \n", i - j);

j = lps[j - 1];

}

else if (i < N && pat[j] != txt[i]) {

if (j != 0){

j = lps[j - 1];

}

else{

i = i + 1;

}

}

}

}

void computeLPSArray(char \*pat, int M, int \*lps) {

int len = 0;

lps[0] = 0;

int i = 1;

while (i < M) {

if (pat[i] == pat[len]) {

len++;

lps[i] = len;

i++;

}

else {

if (len != 0) {

len = lps[len - 1];

}

else {

lps[i] = 0; i++;

}

}

}

}

int main() {

char txt[] = "AABCAACAADAABCAAADABCAA";

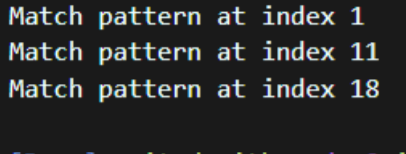
char pat[] = "ABC";

KMPSearch(pat, txt);

return 0;

}

**OUTPUT :-**

****

**RESULT AND ANALYSIS:-**

Each time through the loop, either we increase i or we slide the pattern right. Both of these events can occur at most n times, and so the repeat loop is executed at most 2n times. The cost of each iteration of the repeat loop is O(1). Therefore, the running time is O(n), assuming that the values π(q) are already computed.