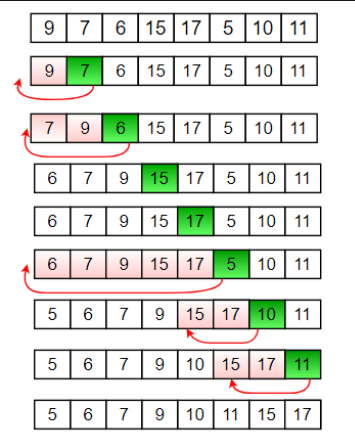
**Experiment No: 1**

**Aim**

Write a Program to demonstrate Insertion Sort and analyse the Time Complexity with different range of inputs

**Introduction**

Insertion Sort is one of the simplest Sorting Technique. 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**

It is a **STABLE** ( i.e , it maintains the relative order of the items with equal sort keys ) and **IN-PLACE ALGORITHM** (does not need an extra space and produces an output in the same memory that contains the data ) .

Steps to sort the array of size N in Asceding Order :

* Iterate from **arr[1] to arr[N-1]** over the array.
* Compare the current element (key) to its predecessor.
* 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.

**Program**

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

#include <stdbool.h>

**#define N 10000**

**int i, j, key;**

bool **isSorted** ( int \*arr )

{

for (i = 0; i < N - 1; i++)

{

if ( arr[i + 1] < arr[i] )

{

return false;

}

return true;

}

}

void **insertion\_sort** ( int \*arr )

{

long long int Comparisons = 0;

for (i = 1; i < N; i++)

{

key = arr[i];

for (j = i - 1; j >= 0; j--)

{

Comparisons++;

if (key < arr[j])

{

arr[j + 1] = arr[j];

}

else

{

break;

}

}

arr[j + 1] = key;

}

printf("\nNo of Comparisons is %llu\n", Comparisons);

}

void **calculateTime** ( int \*arr )

{

clock\_t time;

time = clock();

insertion\_sort(arr);

time = clock() - time;

double time\_taken = ((double)time) / CLOCKS\_PER\_SEC;

printf("The Time taken is equal to %.12lf seconds\n", time\_taken);

}

**int main ( )**

{

int arr[N];

// using random() function to assign random values to the array elements

for (i = 0; i < N; i++)

{

arr[i] = rand();

}

**// \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* A V E R A G E C A S E \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

**calculateTime(arr);**

**isSorted(arr)** ? printf("\nThe Array is Sorted , All is Well\n ") : printf("The Array is not Sorted :( ");

**// \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* B E S T C A S E \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

**calculateTime(arr);**

**// \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* W O R S T C A S E \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

**// reversing the array ( Worst Case ) ( By Swapping )**

for (i = 0; i < N / 2; i++)

{

arr[i] = arr[i] + arr[N - i - 1];

arr[N - i - 1] = arr[i] - arr[N - i - 1];

arr[i] = arr[i] - arr[N - i - 1];

}

**reverse(arr);**

**calculateTime(arr);**

}

**Output**

***Comparisons”* refers the total number of checks of a[i-1] > a[i] for all i >= 1 to i < n .**

|  |  |  |  |
| --- | --- | --- | --- |
| **INPUT SIZE**  **( N )** | **AVERAGE CASE**  **TIME COMPLEXITY**  **(in ms)** | **BEST CASE**  **TIME COMPLEXITY**  **(in ms)** | **WORST CASE TIME COMPLEXITY**  **(in ms)** |
|  | **RunTime , Comparisons** | **RunTime , Comparisons** | **RunTime , Comparisons** |
| 10 | 0 ms  22 | **0 ms**  **9** | **0 ms**  **45** |
| 100 | **0 ms**  **2459** | **0 ms**  **99** | **0 ms**  **4950** |
| 1000 | 2 ms  249143 | 0 ms  999 | 10 ms  499500 |
| 10000 | 80ms  24962883 | 0 ms  9999 | 156 ms  49994850 |
| 100000 | 7530 ms , 2506144504 | 6 ms  99999 | 15318 ms  4999866158 |

**Analysis**

**Best Case ( Sorted Array )** takes the least time since it undergoes only ( N-1 ) Comparisons. ( Time Complexity ≈ O(N) )

**Average Case ( Random Input Array )** takes on an average of the time of Best and Worst Cases. ( Time Complexity ≈ O( N2 / 4 ) ≈ O(N2) )

**Worst Case ( Reversed Array )** undergoes the most number of Comparsions and hence it takes the most time of all of cases.

No.of Comparisons = ( n-1 ) + ( n-2 ) + ( n-3 ) + …. 3 + 2 + 1

= **( n2 – n ) / 2 ;**

∴ Time Complexity ≈ O( N2 / 2 ) ≈ O(N2)

|  |  |  |  |
| --- | --- | --- | --- |
| **CASES** | **AVERAGE** | **BEST** | **WORST** |
| **TIME COMPLEXITY** | O(N2) | O(N) | O(N2) |

**Applications**

**1 ) Sorting a small list of numbers:**

If you have a small list of numbers that needs to be sorted, insertion sort can be an efficient option. It's simple to implement and its time complexity is O(n^2) which make it good for small lists.

**2 ) Sorting Linked List :**

Insertion sort is also good for sorting linked list, when compared to other sorting algorithms like merge sort which needs extra space for merge operations

**References**

1 ) <https://www.geeksforgeeks.org/insertion-sort/>

( Press Ctrl + Click to open )

2) Image

https://www.geeksforgeeks.org/recursive-insertion-sort/