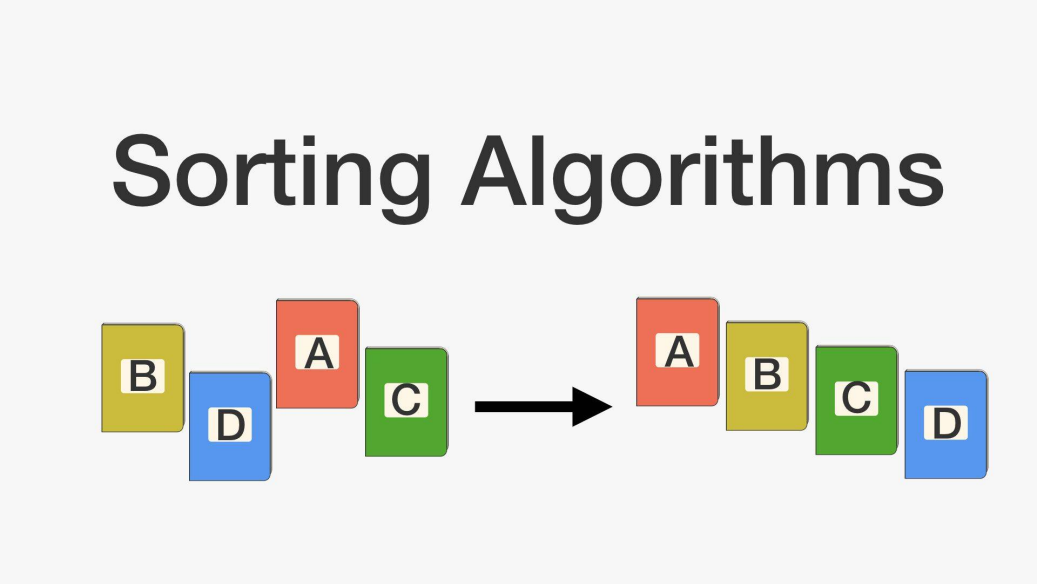
On Time-Complexity of various Sorting algorithms

Pulkit Malhota, Class XII 2018-19

RN Podar School, Mumbai



Preprint submitted to CBSE *September 17, 2019*

Contents

1. Acknowledgement

I would like to express my deepest regards to the Principal Mrs. Avnita Bir for providing us with the infrastructure and facilities for this project.

I would further like to extend my thanks to my family and friends for their constant support throughout the project.

I also wish to express my gratitude to the Computer Science Department of the school, especially Mr.Manish Agarwal and Ms. Shamila K., under whose guidance this project was performed and without whom this project would have never seen the light of day.

2. Introduction

The time complexity is the computational complexity that describes the amount of time it takes to run an algorithm. Time complexity is commonly estimated by counting the number of elementary operations performed by the algorithm, supposing that each elementary operation takes a fixed amount of time to perform. Thus, the amount of time taken and the number of elementary operations performed by the algorithm are taken to differ by at most a constant factor.

A Sorting Algorithm is used to rearrange a given array or list elements according to a comparison operator on the elements. The comparison operator is used to decide the new order of an element in the respective data structure. Because many programs use it as an intermediate step, sorting is a fundamental operation in computer science.

Since an algorithm's running time may vary among different inputs of the same size, one commonly considers the worst-case time complexity, which is the maximum amount of time required for inputs of a given size. Less common, and usually specified explicitly, is the average-case complexity, which is the average of the time taken on inputs of a given size.

In both cases, the time complexity is generally expressed as a function of the size of the input Since this function is generally difficult to compute exactly, and the running time for small inputs is usually not consequential, one commonly focuses on the behaviour of the complexity when the input size increasesthat is, the asymptotic behaviour of the complexity. Therefore, the time complexity is commonly expressed using big O notation.

In this project, we analysis the time taken by the INSERTION-SORT. The procedure depends on the input: sorting a thousand numbers takes longer than sorting three numbers. Moreover, INSERTION-SORT can take different amounts of time to sort two input sequences of the same size depending on how nearly sorted they already are.

In general, the time taken by an algorithm grows with the size of the input, so it is traditional to describe the running time of a program as a function of the size of its input. To do so, we need to define the terms running time and size of input more carefully

A constant amount of time is required to execute each line of our pseudocode. One line may take a different amount of time than another line, but we shall assume that each execution of the ith line takes time t, where t is a constant. The expression for the running time of INSERTION-SORT will

evolve from a messy formula that uses all the statement costs t to a much simpler notation that is more concise and more easily manipulated. This simpler notation will also make it easy to determine whether one algorithm is more efficient than another

We shall now make one more simplifying abstraction: it is the rate of growth, or order of growth, of the running time that really interests us. We therefore consider only the leading term of a formula (e.g., $a(n)^2$), since the lower-order terms are relatively insignificant for large values of n.

We also ignore the leading terms constant coefficient, since constant factors are less significant than the rate of growth in determining computational efficiency for large inputs. For insertion sort, when we ignore the lower-order terms and the leading terms constant coefficient, we are left with the factor of n^2 from the leading term. We write that insertion sort has a worst-case running time of n^2

We start by presenting the INSERTION-SORT procedure with the time cost of each statement and the number of times each statement is executed. We let t_j denote the number of times the while loop is executed. When a for or while loop exits in the usual way (i.e., due to the test in the loop header), the test is executed one time more than the loop body. We assume that comments are not executable statements, and so they take no time.

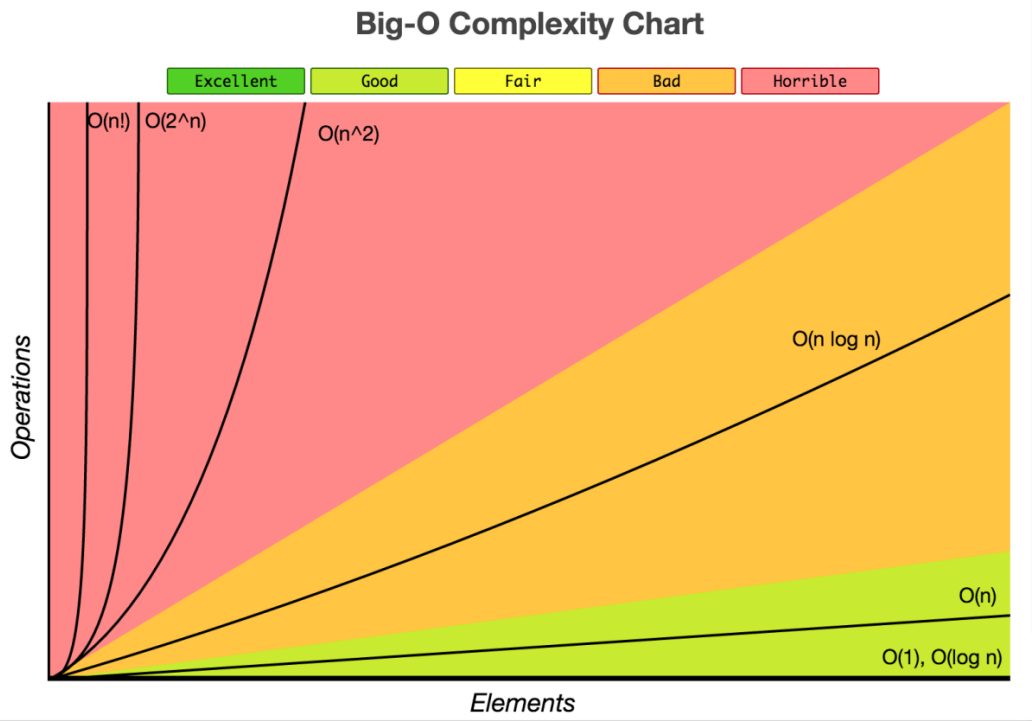
Analyzing an algorithm has come to mean predicting the resources that the algorithm requires. Occasionally, resources such as memory, communication bandwidth, or computer hardware are of primary concern, but most often it is computational time that we want to measure. Generally, by analyzing several candidate algorithms for a problem, we can identify a most efficient one. Such analysis may indicate more than one viable candidate, but we can often discard several inferior algorithms in the process

3. Big O Notation

Big O notation is a mathematical notation that describes the limiting behavior of a function when the argument tends towards a particular value or infinity.

In computer science, big O notation is used to classify algorithms according to how their running time or space requirements grow as the input size grows. In analytic number theory, big O notation is often used to express a bound on the difference between an arithmetical function and a better understood approximation; a famous example of such a difference is the remainder term in the prime number theorem.

Big O notation characterizes functions according to their growth rates: different functions with the same growth rate may be represented using the same O notation.



4. Sorting

4.1. Bubble Sort

Bubble sort, sometimes referred to as sinking sort, is a simple sorting algorithm that repeatedly steps through the list, compares adjacent pairs and swaps them if they are in the wrong order. The pass through the list is repeated until the list is sorted. The algorithm, which is a comparison sort, is named for the way smaller or larger elements "bubble" to the top of the list. Although the algorithm is simple, it is too slow and impractical for most problems even when compared to insertion sort. Bubble sort can be practical if the input is in mostly sorted order with some out-of-order elements nearly in position.

Listing 1: Bubble Sort

```
1  void sort::bubblesort(long arr[ ], long n)
2  {
3          for(long i=0; i<(n-1); i++)
4      {   for(long j=0; j<(n-i-1); j++)
5          {   if(arr[j]>arr[j+1])
6              {   long temp=arr[j];
7                  arr[j]=arr[j+1];
8                  arr[j+1]=temp;
9              }
10         }
11     }
12 }
```

4.2. Insertion Sort

Insertion sort iterates, consuming one input element each repetition, and growing a sorted output list. At each iteration, insertion sort removes one element from the input data, finds the location it belongs within the sorted list, and inserts it there. It repeats until no input elements remain.

Sorting is typically done in-place, by iterating up the array, growing the sorted list behind it. At each array-position, it checks the value there against the largest value in the sorted list (which happens to be next to it, in the previous array-position checked). If larger, it leaves the element in place and moves to the next. If smaller, it finds the correct position within the sorted list, shifts all the larger values up to make a space, and inserts into that correct position.

Listing 2: Insertion Sort

```
1  void sort::insertsort(long arr[], long n)
2  {
3      long i, j,temp;
4
5      for(i=1; i<n; i++)
6      {    temp=arr[i];
7           j=i-1;
8           while((temp<arr[j]) && (j>=0))
9           {    arr[j+1]=arr[j];
10               j=j-1;
11          }
12          arr[j+1]=temp;
13     }
14 }
```

4.3. Selection Sort

In computer science, selection sort is a sorting algorithm, specifically an in-place comparison sort. It has $O((n))^2)$ time complexity, making it inefficient on large lists, and generally performs worse than the similar insertion sort. Selection sort is noted for its simplicity, and it has performance advantages over more complicated algorithms in certain situations, particularly where auxiliary memory is limited.

The algorithm divides the input list into two parts: the sublist of items already sorted, which is built up from left to right at the front (left) of the list, and the sublist of items remaining to be sorted that occupy the rest of the list. Initially, the sorted sublist is empty and the unsorted sublist is the entire input list. The algorithm proceeds by finding the smallest (or largest, depending on sorting order) element in the unsorted sublist, exchanging (swapping) it with the leftmost unsorted element (putting it in sorted order), and moving the sublist boundaries one element to the right.

Listing 3: Selection Sort

```
1  void sort::selsort(long arr[], long n)
2  {
3      long min_idx;
4
5          for(long i = 0; i < n-1; i++)
6          {       min_idx = i;
7                  for(long j = i+1; j < n; j++)
8                  if(arr[j] < arr[min_idx])
9                          min_idx = j;
10
11                 long temp = arr[min_idx];
12         arr[min_idx] = arr[i];
13         arr[i] = temp;
14     }
15 }
```

5. Time Complexity

5.1. Constant Time

An algorithm is said to be constant time (also written as O(1) time) if the value of $T(n)$ is bounded by a value that does not depend on the size of the input. For example, accessing any single element in an array takes constant time as only one operation has to be performed to locate it. In a similar manner, finding the minimal value in an array sorted in ascending order; it is the first element. However, finding the minimal value in an unordered array is not a constant time operation as scanning over each element in the array is needed in order to determine the minimal value. Hence it is a linear time operation, taking $O(n)$ time. If the number of elements is known in advance and does not change, however, such an algorithm can still be said to run in constant time.

Despite the name "constant time", the running time does not have to be independent of the problem size, but an upper bound for the running time has to be bounded independently of the problem size.

5.2. Logarithmic time

An algorithm is said to take logarithmic time when $T(n) = O(log(n))$. Since $log_a\ n$ and $log_b\ n$ are related by a constant multiplier, and such a multiplier is irrelevant to big-O classification, the standard usage for logarithmic-time algorithms is $O(log(n))$ regardless of the base of the logarithm appearing in the expression of T.

Algorithms taking logarithmic time are commonly found in operations on binary trees or when using binary search.

An $O(log(n))$ algorithm is considered highly efficient, as the ratio of the number of operations to the size of the input decreases and tends to zero when n increases. An algorithm that must access all elements of its input cannot take logarithmic time, as the time taken for reading an input of size n is of the order of n.

5.3. Polylogarithmic time

An algorithm is said to run in polylogarithmic time if $T(n) = O((log(n))^k)$, for some constant k. For example, matrix chain ordering can be solved in polylogarithmic time on a parallel random-access machine.

5.4. Linear Time

An algorithm is said to take linear time, or $O(n)$ time, if its time complexity is $O(n)$. Informally, this means that the running time increases at most linearly with the size of the input. More precisely, this means that there is a constant c such that the running time is at most cn for every input of size n. For example, a procedure that adds up all elements of a list requires time proportional to the length of the list, if the adding time is constant, or, at least, bounded by a constant.

6. Theory

INSERTION-SORT(A)	*cost*	*times*
1 **for** $j = 2$ **to** $A.length$	c_1	n
2 $key = A[j]$	c_2	$n-1$
3 // Insert $A[j]$ into the sorted sequence $A[1..j-1]$.	0	$n-1$
4 $i = j-1$	c_4	$n-1$
5 **while** $i > 0$ and $A[i] > key$	c_5	$\sum_{j=2}^{n} t_j$
6 $A[i+1] = A[i]$	c_6	$\sum_{j=2}^{n}(t_j-1)$
7 $i = i-1$	c_7	$\sum_{j=2}^{n}(t_j-1)$
8 $A[i+1] = key$	c_8	$n-1$

The running time of the algorithm is the sum of running times for each statement executed; a statement that takes c_i steps to execute and executes n times will contribute can to the total running time. To compute $T(n)$, the running time of INSERTION-SORT on an input of n values, we sum the products of the cost and times columns, obtaining

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^{n} t_j + c_6 \sum_{j=2}^{n}(t_j - 1) + c_7 \sum_{j=2}^{n}(t_j - 1) + c_8(n-1) \quad (1)$$

Even for inputs of a given size, an algorithm's running time may depend on which input of that size is given. For example, in INSERTION-SORT, the best case occurs if the any is already sorted. For each $j = 2, 3, ...n$ we then find that $A[i] \leq key$ in line 5 when i has its initial value of j - 1. Thus $t_j = 1$ for $j = 2, 3, ...n$ and the best-case running time is

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5(n-1) + c_8(n-1)$$
$$T(n) = (c_1 + c_2 + c_4 + c_5 + c_8)n - (c_2 + c_4 + c_5 + c_8)$$

We can express this running time as $a(n) + b$ for constants a and b that depend on the statement costs 0 : it is thus a linear function of n. If the array is in reverse sorted orderthat is, in decreasing orderthe worst case results. We must compare each element $A[j]$ with each element in the entire soiled sub-array $A[1....j-1]$, and so $t_j = j$ for $j = 2, 3, ...n$. Noting that

$$\sum_{j=2}^{n} j = (\frac{n(n+1)}{2} - 1)$$

and

$$\sum_{j=2}^{n} (j-1) = (\frac{n(n+1)}{2} - 1)$$

We find that in the worst case, the running time of INSERTION-SORT is

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5(\frac{n(n+1)}{2} - 1) + c_6(\frac{n(n+1)}{2} - 1) + c_7(\frac{n(n+1)}{2} - 1) + c_8(n-1) \quad (2)$$

$$T(n) = (\frac{c_5}{2} + \frac{c_6}{2} + \frac{c_7}{2})n^2 - (c_1 + c_2 + c_4 + \frac{c_5}{2} - \frac{c_6}{2} - \frac{c_7}{2} + c_8)n - (c_1 + c_2 + c_4 + c_8) \quad (3)$$

We can express this worst-case running time as $a(n)^2 + b(n) + c$ for constants a, b. and c that again depend on the statement costs c_j; it is thus a quadratic function of n. Typically, as in insertion sort, the running time of an algorithm is fixed for a given input, although in later chapters we shall see some interesting "randomized" algorithms whose behavior can vary even for a fixed input.

7. Analysis

7.1. Worst-case and average-case analysis

In our analysis of insertion sort, we looked at both the best case, in which the input array was already sorted, and the worst case, in which the input array was reverse sorted. We shall usually concentrate on finding only the worst-case running time, that is, the longest running time for any input of size n. We give three reasons for this orientation.

- The worst-case running time of an algorithm gives us an upper bound on the running time for any input. Knowing it provides a guarantee that the algorithm will never take any longer. We need not make some educated guess about the running time and hope that it never gets much worse.

- For some algorithms, the worst case occurs fairly often. For example, in searching a database for a particular piece of information, the searching algorithms worst case will often occur when the information is not present in the database. In some applications, searches for absent information may be frequent.

- The average case is often roughly as bad as the worst case. Suppose that we randomly choose n numbers and apply insertion sort. How long does it take to determine where in sub-array $A[1..j1]$ to insert element $A[j]$ On average, half the elements in $A[1..j-1]$ are less than $A[j]$, and half the elements are greater. On average, therefore, we check half of the sub-array $A[1..j-1]$, and so t_j is about $\frac{j}{2}$. The resulting average-case running time turns out to be a quadratic function of the input size, just like the worst-case running time.

7.2. Best case Analysis

The term best-case performance is used in computer science to describe an algorithm's behavior under optimal conditions. For example, the best case for a simple linear search on a list occurs when the desired element is the first element of the list. Development and choice of algorithms is rarely based on best-case performance though and usually the improvements made to the code revolve around the worst case and average case scenarios.

8. Source Code

Listing 4: Source Sort

```
1  //---------- SOURCE CODE ----------
2
3  //---------- Header Section ----------
4  #include <iostream>
5  #include <ctime>
6  #include <cmath>
7  #include<fstream>
8  #include <cstdlib>
9  #include <chrono> //Only in C++ 11
10
11 using namespace std;
12
13 // This section consists solely of declarations
14 // Put all class structure related things here.
15 // Put all function prototypes here.
16 // Only put prototypes and not the definition.
17
18 class sort
19 {
20 public:
21     void insertsort(long _arr[ ], long n);
22     //Insertion Sort declared
23     void selsort(long _arr[ ], long n);
24     //Selection Sort declared
25     void bubblesort(long _arr[ ], long n);
26     //Bubble Sort declared
27 };
28 //Insertion Sort defined
29 void sort::insertsort(long _arr[ ], long n)
30 {
31     long arr[n];
32         for (long i = 0; i < n; i++)
33         {
34             arr[i]=_arr[i];
35         }
```

```
36
37      long i, j,temp;
38
39      for(i=1; i<n; i++)
40      {    temp=arr[i];
41           j=i-1;
42           while((temp<arr[j]) && (j>=0))
43           {    arr[j+1]=arr[j];
44                j=j-1;
45           }
46           arr[j+1]=temp;
47      }
48      cout<<"Insertion Sort complete";
49  }
50
51  //Selection Sort defined
52  void sort::selsort(long _arr[ ], long n)
53  {
54  long arr[n];
55          for (long i = 0; i < n; i++)
56          {
57               arr[i]=_arr[i];
58          }
59
60      long min_idx;
61
62          for(long i = 0; i < n-1; i++)
63          {        min_idx = i;
64                   for(long j = i+1; j < n; j++)
65                   if(arr[j] < arr[min_idx])
66                            min_idx = j;
67
68                   long temp = arr[min_idx];
69          arr[min_idx] = arr[i];
70          arr[i] = temp;
71      }
72      cout<<"Selection Sort complete";
73
```

```
74 }
75
76 //Bubble Sort defined
77 void sort::bubblesort(long _arr[ ], long n)
78 {   long arr[n];
79         for (long i = 0; i < n; i++)
80         {
81             arr[i]=_arr[i];
82         }
83
84         for(long i=0; i<(n-1); i++)
85     {   for(long j=0; j<(n-i-1); j++)
86         {   if(arr[j]>arr[j+1])
87             {   long temp=arr[j];
88                 arr[j]=arr[j+1];
89                 arr[j+1]=temp;
90             }
91         }
92     }
93
94     cout<<"Bubble Sort complete";
95 }
96 //---------- Final main Function ----------
97 int main()
98 {
99     const long count= 1000;
100    //constant defined
101    long a[count];
102    clock_t time_req,time_req1,time_req2;
103    //time variables defined
104    sort obj;
105    //object of class sort declared
106
107
108        for (long i = 0; i < count; i++)
109         {
110            a[i]= random();
111         }
```

```
112         //Taking random input in the array
113
114             time_req = clock();
115             //measuring current time of system
116             obj.bubblesort(a, count);
117             //Bubble sorting
118             time_req = clock() - time_req;
119             //measuring time after sorting
120             cout << " Using Bubble Sort, it took "
121             << (double)time_req/CLOCKS_PER_SEC
122             << " seconds" << endl;
123             //getting the difference to get time taken
124
125
126             time_req1 = clock();
127             obj.selsort(a, count);//Selection Sort
128             time_req1 = clock() - time_req1;
129             cout << " Using Selection Sort, it took "
130             << (double)time_req1/CLOCKS_PER_SEC
131             << " seconds" << endl;
132
133             time_req2 = clock();
134             obj.insertsort(a, count); //Insertion sort
135             time_req2 = clock() - time_req2;
136             cout << " Using Insertion Sort, it took "
137             << (double)time_req2/CLOCKS_PER_SEC
138             << " seconds" << endl;
139
140             ofstream fout;
141                fout.open("PARA1.TXT");
142               if(!fout.eof())
143               {
144                 fout<<time_req<<time_req1<<time_req2;
145               }
146               //Data is stored in a text file
147              fout.close();
148 return 0; }
```

9. Hardware Requirements

This program can be run on the following hardware:

- 1 GB RAM and above
- Intel Pentium IV processor and above
- CPU Architecture 32 bit

10. Software Requirements

This program can be run on the following software

- Windows XP/Vista and above

11. Developed on

This project was made with the following software:

- Development and Debugging with g++ v7.1.1 and gdb v8.0
- Executable created in Bloodhound Dev C++ with MinGW-

12. Output

The output comes in the form:

```
Number of elements in the array: 5000

Elements sorted successfully by Bubble Sort
 Using Bubble Sort, it took 0.046797 seconds
Elements sorted successfully by Selection Sort
 Using Selection Sort, it took 0.009778 seconds
Elements sorted successfully by Insertion Sort
 Using Insertion Sort, it took 0.006872 seconds

Exit code: 0 (normal program termination)
```

Figure 1: Compiled Output

For different inputs, the three sorts take the following time:

No. of elements in the array	Insertion sort (in seconds)	Selection sort (in seconds)	Bubble sort (in seconds)
500	0.000077	0.000138	0.000429
1000	0.000286	0.000456	0.001883
1500	0.001163	0.001783	0.007332
2000	0.001330	0.001967	0.012939
2500	0.001691	0.002489	0.013500
3000	0.002477	0.003699	0.017317
3500	0.003379	0.004839	0.029584
4000	0.004377	0.005984	0.027859
4500	0.005531	0.007818	0.042182
5000	0.006872	0.009778	0.046797
5500	0.008239	0.011609	0.055783

Figure 2: Input-Time Table

13. Results

From the data that we have procured, we can safely conclude that Insertion Sort is the most efficient sorting algorithm, followed by Selection Sort, which in turn is followed by Bubble Sort. The plot of average time taken by each takes a parabolic form and is directly proportional to the number of elements in the array and the size of each element.

Algorithm	Time complexity: Best	Time complexity: Average	Time complexity: Worst
Bubble sort	$O(n)$	$O(n^2)$	$O(n^2)$
Insertion sort	$O(n)$	$O(n^2)$	$O(n^2)$
Selection sort	$O(n^2)$	$O(n^2)$	$O(n^2)$

Figure 3: Big O notation for the three sorts

14. Bibliography

1. Computer Science textbook for Class XII by Sumita Arora
2. Sipser, Michael (2006). Introduction to the Theory of Computation. Course Technology
3. Sedgewick, Robert (1 September 1998). Algorithms In C: Fundamentals, Data Structures, Sorting, Searching, Parts 1-4 (3 ed.). Pearson Education.
4. Sedgewick, R. and Wayne K (2011). Algorithms, 4th Ed. p. 186. Pearson Education, Inc.
5. Cormen, Thomas H.; Leiserson, Charles E.; Rivest, Ronald L.; Stein, Clifford (2001) [1990]. Introduction to Algorithms (2nd ed.).

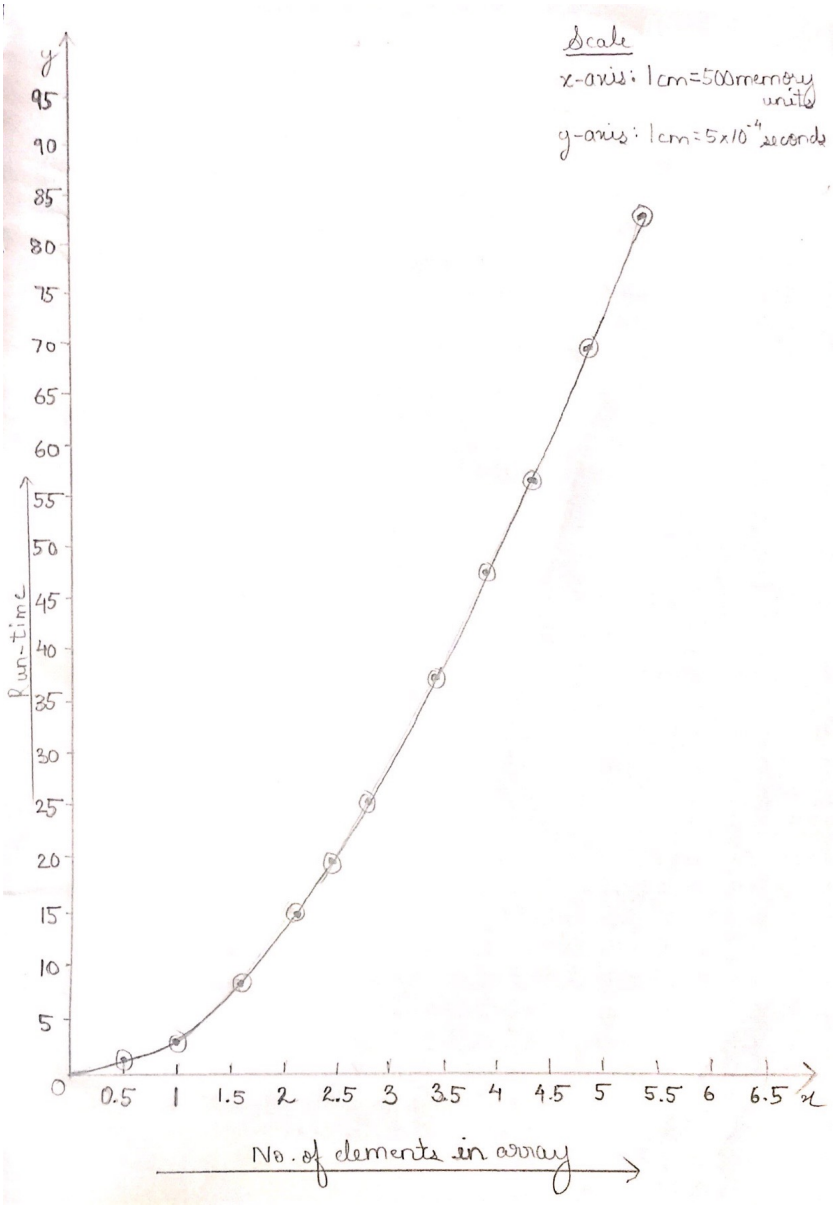


Figure 4: Time vs No. of input graph