

Name	Advait Ravi Sapkal
UID No.	2021700055
Experiment No.	1 B

AIM:	Experiment with finding the running time of an algorithm. Comparing selection and insertion sort algorithms.
Program 1	
PROBLEM STATEMENT :	<p>Details – The understanding of running time of algorithms is explored by implementing two basic sorting algorithms namely Insertion and Selection sorts. These algorithms work as follows.</p> <p>Insertion sort– It works similar to the sorting of playing cards in hands. It is assumed that the first card is already sorted in the card game, and then we select an unsorted card. If the selected unsorted card is greater than the first card, it will be placed at the right side; otherwise, it will be placed at the left side. Similarly, all unsorted cards are taken and put in their exact place.</p> <p>Selection sort– It first finds the smallest value among the unsorted elements of the array is selected in every pass and inserted to its appropriate position into the array. In this algorithm, the array is divided into two parts, first is sorted part, and another one is the unsorted part. Initially, the sorted part of the array is empty, and unsorted part is the given array. Sorted part is placed at the left, while the unsorted part is placed at the right. In selection sort, the first smallest element is selected from the unsorted array and placed at the first position. After that second smallest element is selected and placed in the second position. The process continues until the array is entirely sorted.</p> <hr/> <p>Problem Definition & Assumptions – For this experiment, you need to implement two sorting algorithms namely Insertion and Selection sort methods. Compare these algorithms based on time and space complexity. Time required to sorting algorithms can be performed using <code>high_resolution_clock::now()</code> under namespace <code>std::chrono</code>.</p> <p>You have to generate 1,00,000 integer numbers using C/C++ <code>Rand</code> function and save them in a text file. Both the sorting algorithms use these 1,00,000 integer numbers as input as follows. Each sorting algorithm sorts a block of 100 integers numbers with array indexes numbers <code>A[0..99]</code>, <code>A[0..199]</code>, <code>A[0..299]</code>, ..., <code>A[0..99999]</code>. You need to use <code>high_resolution_clock::now()</code> function to find the time required for 100, 200, 300.... 100000 integer numbers. Finally, compare two algorithms namely Insertion and Selection by plotting the time required to sort 100000 integers using LibreOffice Calc/MS Excel. The x-axis of 2-D plot represents the block no. of 1000 blocks. The y-axis of 2-D plot represents the running time to sort 1000 blocks of 100,200,300,...,100000 integer numbers.</p> <p>Note – You have to use C/C++ file processing functions for reading and writing randomly generated 100000 integer numbers.</p>
ALGORITHM:	<p>Insertion and selection sort are the 2 most fundamental sorting algorithms.</p> <p>Selection sort is a non-scalable, non-practical algorithm whereas insertion sort is practical for small inputs. Insertion sort is a stable sorting algorithm whereas Selection sort is unstable.</p> <p style="text-align: center;">Selection Sort:</p> <p>Invariant: All the elements to the left of the iterator are sorted and no element on the right is smaller than any of the elements on the left.</p>

Algorithm:

Algorithm 4 Selection Sort

```
1: for  $i = 1$  to  $n - 1$  do
2:    $min = i$ 
3:   for  $j = i + 1$  to  $n$  do
4:     // Find the index of the  $i^{th}$  smallest element
5:     if  $A[j] < A[min]$  then
6:        $min = j$ 
7:     end if
8:   end for
9:   Swap  $A[min]$  and  $A[i]$ 
10: end for
```

Trace:

i	min	0	1	2	3	4	5	6	7	8	9	10
		S	O	R	T	E	X	A	M	P	L	E
0	6	S	O	R	T	E	X	A	M	P	L	E
1	4	A	O	R	T	E	X	S	M	P	L	E
2	10	A	E	R	T	O	X	S	M	P	L	E
3	9	A	E	E	T	O	X	S	M	P	L	R
4	7	A	E	E	L	O	X	S	M	P	T	R
5	7	A	E	E	L	M	X	S	O	P	T	R
6	8	A	E	E	L	M	O	S	X	P	T	R
7	10	A	E	E	L	M	O	P	X	S	T	R
8	8	A	E	E	L	M	O	P	R	S	T	X
9	9	A	E	E	L	M	O	P	R	S	T	X
10	10	A	E	E	L	M	O	P	R	S	T	X
		A	E	E	L	M	O	P	R	S	T	X

Code:

```
void selection_sort(int* arr, int len)
{
    for (int i = 0; i < len; i++)
    {
        int mini = i;
        for (int j = i + 1; j < len; j++) if (arr[j] < arr[mini]) mini = j;
        swap(arr, i, mini);
    }
}
```

Analysis:

```
void selection_sort (int* arr, int len)
{
    for (int i = 0; i < len; i++)
    {
        int mini = i;
        for (int j = i + 1; j < len; j++)
            if (arr[j] < arr[mini]) mini = j;
        swap(arr, i, mini);
    }
}
```

Complexity analysis for selection sort:

- Outer loop: $N+1$
- Inner loop: N
- Comparison: $N(N-1)/2$
- Swap: $N(N-1)/2$
- Swap: N

Total cost: $4N + \frac{3N(N-1)}{2} + 1$ (ignoring cost coefficients)

$\approx 4N + \frac{3N^2}{2} - \frac{3N}{2} + 1$

$\approx \frac{3N^2}{2}$

$\approx \Theta(N^2)$, $O(N^2)$ in worst case, $\Omega(N^2)$ in best case

Insertion Sort:

Invariant: Elements to the left of the iterator are sorted, whereas elements to right are not seen yet. At every turn, each chosen element is put in its proper place on the left side of the iterator.

Algorithm:

```
INSERTION-SORT(A)
1 for j ← 2 to length[A]
2   do key ← A[j]
3   ▷ Insert A[j] into the sorted sequence A[1..j-1].
4   i ← j - 1
5   while i > 0 and A[i] > key
6     do A[i + 1] ← A[i]
7     i ← i - 1
8   A[i + 1] ← key
```

Trace:

i	j	0	1	2	3	4	5	6	7	8	9	10
		S	O	R	T	E	X	A	M	P	L	E
1	0	O	S	R	T	E	X	A	M	P	L	E
2	1	O	R	S	T	E	X	A	M	P	L	E
3	3	O	R	S	T	E	X	A	M	P	L	E
4	0	E	O	R	S	T	X	A	M	P	L	E
5	5	E	O	R	S	T	X	A	M	P	L	E
6	0	A	E	O	R	S	T	X	M	P	L	E
7	2	A	E	M	O	R	S	T	X	P	L	E
8	4	A	E	M	O	P	R	S	T	X	L	E
9	2	A	E	L	M	O	P	R	S	T	X	E
10	2	A	E	E	L	M	O	P	R	S	T	X

Code:

```
void insertion_sort(int *arr, int len)
{
    for (int i = 0; i < len; i++)
    {
        int curr = arr[i];
        for (int j = i-1; j >= 0; j--) if (arr[j] > curr) swap(arr, j, j+1);
        else break;
    }
}
```

Analysis:

void insertion_sort(int* arr, int len)

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

{ int curr = arr[i];

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

if (arr[j] > curr)

swap(arr, j, j+1);

else break;

}

}

total cost: $\leq \frac{3N(N-1)}{2} + 2N + 3 + \frac{N(N-1)}{2}$

$\leq 2N(N-1) + 2N + 3$

$\leq 2N^2 + 8$

$\leq \sim 2N^2$

$\leq \Theta(N^2)$

$\therefore \Theta(N^2)$, worst case $O(N^2)$, best case $\Omega(N)$

when ① & ② & ③ execute only once.

PROGRAM:

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include <limits.h>
#include <time.h>

void swap(int *arr, int i, int j)
{
    int temp = arr[i];
    arr[i] = arr[j];
    arr[j] = temp;
}

void selection_sort(int* arr,int len)
{
    for (int i = 0; i < len; i++)
    {
        int mini = i;
        for (int j = i + 1; j < len; j++)if (arr[j] < arr[mini])mini = j;
        swap(arr, i, mini);
    }
}

void insertion_sort(int *arr, int len)
{
    for (int i = 0; i < len; i++)
    {
        int curr = arr[i];
        for (int j = i-1; j >= 0; j--)if (arr[j] > curr)swap(arr, j, j+1);
    else break;
    }
}

int main(){

    //accessing the required file
    FILE *fptr;
    fptr=fopen("input.txt","w");
    if(fptr!=NULL)printf("file accessed.\n");
    else {printf("error in accessing file.\n");return -1;}

    //setting capacity and range to fill the file
```

```

int capacity=100000;
int range=100;

for(int i=0;i<capacity;i++){
    putw(rand()%(range+1),fptr);
}

//initializing reciever array and clock variables
int arr[capacity];
    int arr2[capacity];
clock_t start_t;
clock_t end_t;

for(int i=100;i<=capacity;i+=100){

    //filling for selection sort
    for(int j=0;j<i;j++){
        arr[j]=getw(fptr);
    }
    start_t=clock();
    selection_sort(arr,i);//selection sort
    end_t=clock();
    printf("%7d| %10.3f|",i,(double)(end_t-start_t));

    /*rewind(fp); //or can use fseek(fp, 0, SEEK_SET); to reset
pointer to start of file
as we want same numbers for insertion sort too*/
    fseek(fp, 0, SEEK_SET);

    //filling for insertion sort
    for(int k=0;k<i;k++){
        arr2[k]=getw(fptr);
    }
    start_t=clock();
    insertion_sort(arr2,i);//insertion sort
    end_t=clock();
    printf("%10.3f\n",(double)(end_t-start_t) );
}

//closing file
fclose(fp);
}

```

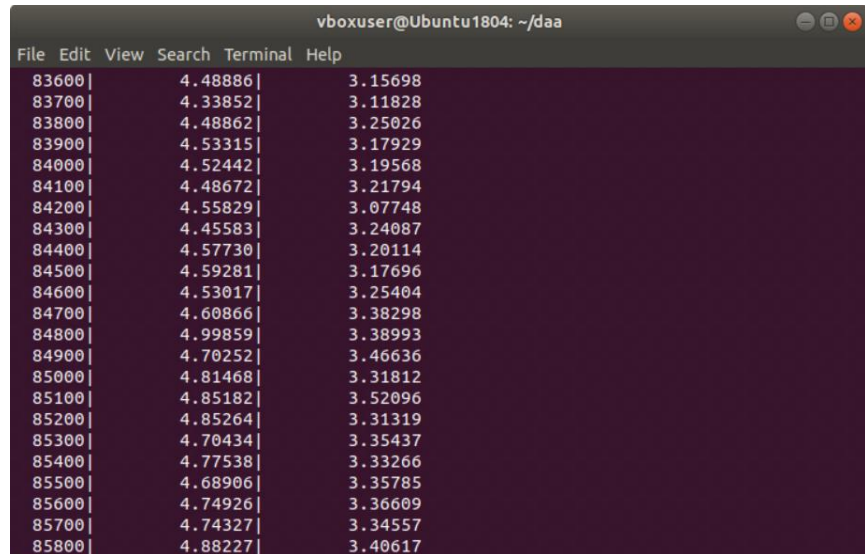
Hence a text file is created with 1 lakh random integers and the experiment is carried upon

this file.

RESULT:

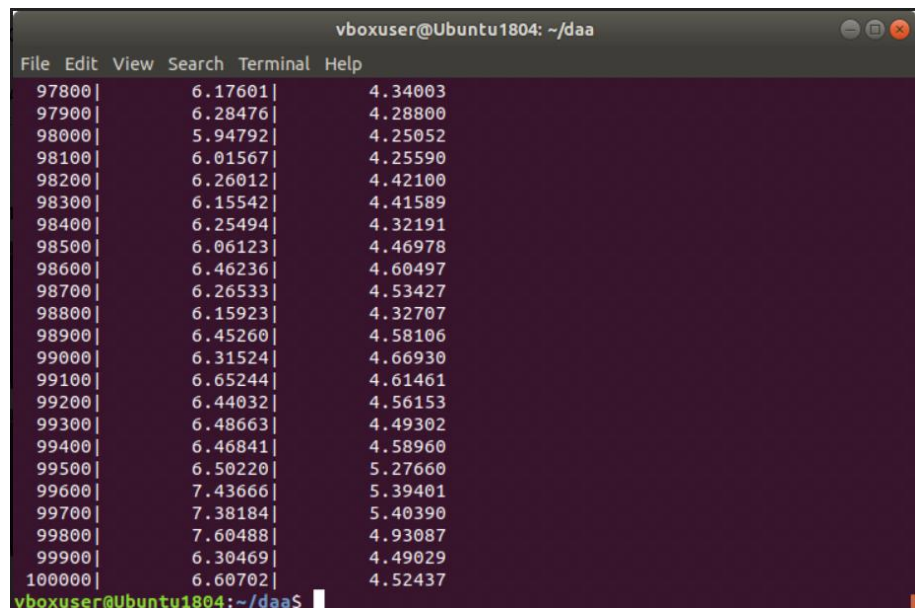
Outputs:

Column 1 is input size, column 2 is selection sort running time, and column 3 is insertion sort running time. All running times are in terms of seconds. Input size was varied from 100 to 100000.



A terminal window titled 'vboxuser@Ubuntu1804: ~/daa' displaying a table of running times for selection and insertion sorts. The table has three columns: input size, selection sort time, and insertion sort time. The data is as follows:

Input Size	Selection Sort Time (s)	Insertion Sort Time (s)
83600	4.48886	3.15698
83700	4.33852	3.11828
83800	4.48862	3.25026
83900	4.53315	3.17929
84000	4.52442	3.19568
84100	4.48672	3.21794
84200	4.55829	3.07748
84300	4.45583	3.24087
84400	4.57730	3.20114
84500	4.59281	3.17696
84600	4.53017	3.25404
84700	4.60866	3.38298
84800	4.99859	3.38993
84900	4.70252	3.46636
85000	4.81468	3.31812
85100	4.85182	3.52096
85200	4.85264	3.31319
85300	4.70434	3.35437
85400	4.77538	3.33266
85500	4.68906	3.35785
85600	4.74926	3.36609
85700	4.74327	3.34557
85800	4.88227	3.40617

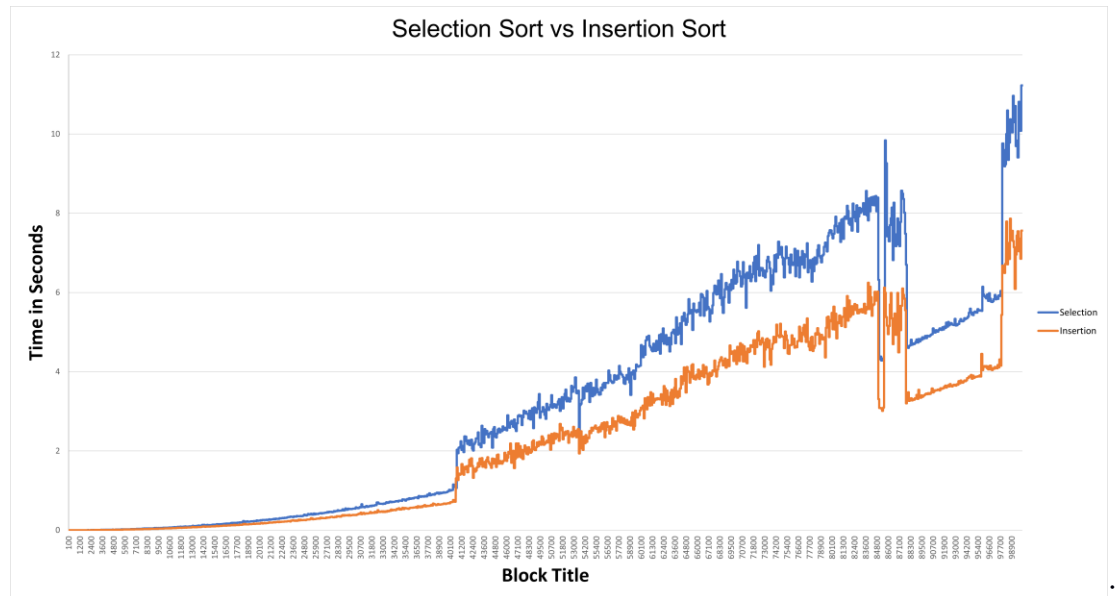


A terminal window titled 'vboxuser@Ubuntu1804: ~/daa' displaying a table of running times for selection and insertion sorts. The table has three columns: input size, selection sort time, and insertion sort time. The data is as follows:

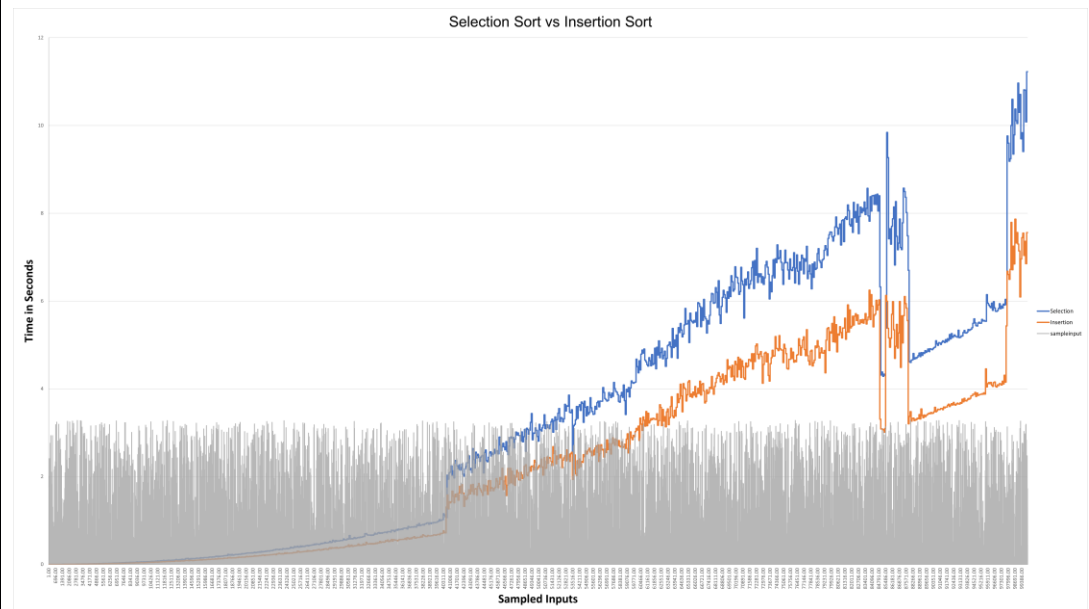
Input Size	Selection Sort Time (s)	Insertion Sort Time (s)
97800	6.17601	4.34003
97900	6.28476	4.28800
98000	5.94792	4.25052
98100	6.01567	4.25590
98200	6.26012	4.42100
98300	6.15542	4.41589
98400	6.25494	4.32191
98500	6.06123	4.46978
98600	6.46236	4.60497
98700	6.26533	4.53427
98800	6.15923	4.32707
98900	6.45260	4.58106
99000	6.31524	4.66930
99100	6.65244	4.61461
99200	6.44032	4.56153
99300	6.48663	4.49302
99400	6.46841	4.58960
99500	6.50220	5.27660
99600	7.43666	5.39401
99700	7.38184	5.40390
99800	7.60488	4.93087
99900	6.30469	4.49029
100000	6.60702	4.52437

Plots:

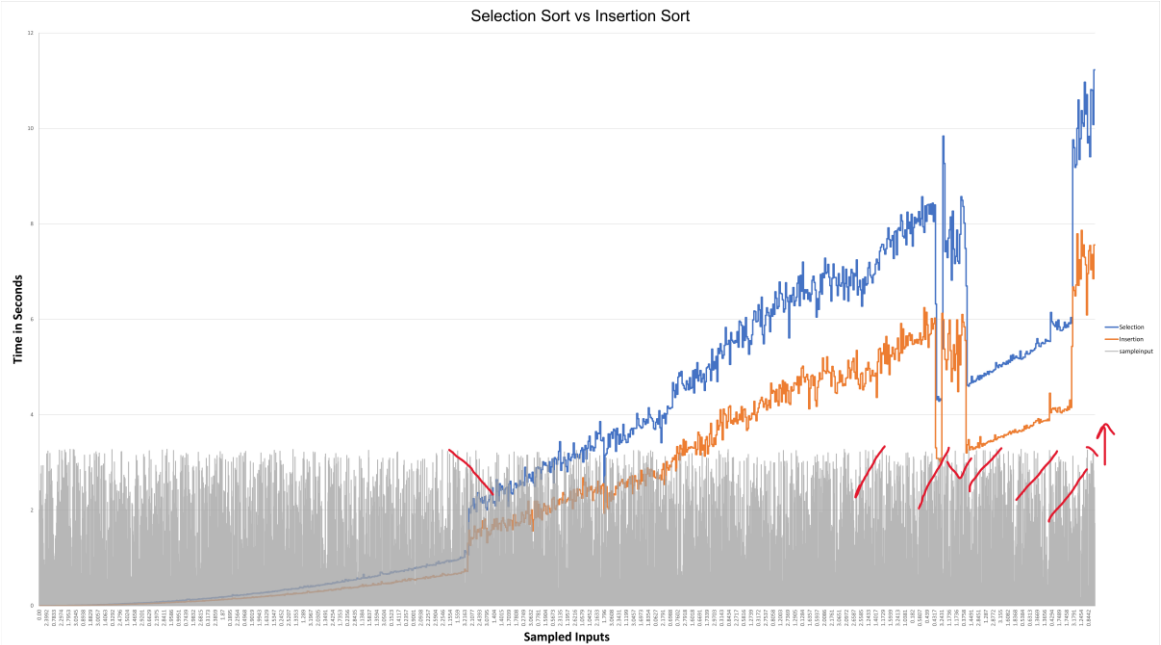
The above output was exported to Excel via a text file and the line graph was plotted.



Plots with the input values to examine spikes,drops etc:



The trends in the input were analysed and observed:



Conclusions:

1. It is clearly seen that insertion sort is much better than selection sort for any input size given.
2. Selection sort is undergoing a quadratic increase in running time with respect to input size and becoming unscalable. This is in line with its theoretical classification of $\Theta(N^2)$.
3. Insertion sort is also quadratic with less rate of increase. It is supposed to be $\sim(N^2)$ on average and $\Theta(N)$ only in the best case.
4. At some places in the graph, it is undergoing a sharp increase or sharp decrease in running time. This could be because the new input block that was included was either already partially sorted in ascending order(leading to sharp decreases) or the block was almost in the reverse, i.e. descending order(leading to sharp increase in running time).
5. At a certain range of inputs, the graphs have become linear. But this might be because some parts of the randomly generated inputs are partially sorted, and hence the only sorting required is for the out-of-order pairs(inversions) in which case it is linearly dependent on the number of inversions.
6. Both algorithms store inputs in arrays and are in-place sorts. We have declared a constant 1 lakh size int array for both algorithms. Other than that, selection sort takes 7 int declarations and 2 pointers. Insertion sort takes 6 int declarations and 2 pointers. All these considerations are irrespective of input size. Hence both algorithms have constant space complexity $\Theta(1)$.