NAME	Shivam Kamble
UID:	2022301007
SUBJECT	DAA
EXPERIMENT NO.	01 B

AIM : Experiment to find the running time of an algorithm.

OBJECTIVE : To find out running time of 2 sorting algorithms - Selection sort and Insertion sort .

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

Characteristics of Insertion Sort:

1. This algorithm is one of the simplest algorithm with simple implementation

2. Basically, Insertion sort is efficient for small data values

3. Insertion sort is adaptive in nature, i.e. it is appropriate for data sets which are already partially sorted.

Selection sort - is a simple and efficient sorting algorithm that works by repeatedly selecting the smallest (or largest) element from the unsorted portion of the list and moving it to the sorted portion of the list. The algorithm repeatedly selects the smallest (or largest) element from the unsorted portion of the list and swaps it with the first element of the unsorted portion. This process is repeated for the remaining unsorted portion of the list until the entire list is sorted. One variation of selection sort is called “Bidirectional selection sort” that goes through the list of elements by alternating between the smallest and largest element, this way the algorithm can be faster in some cases.

The algorithm maintains two subarrays in a given array.

1. The subarray which already sorted.

2. The remaining subarray was unsorted.

In every iteration of the selection sort, the minimum element (considering ascending order) from the unsorted subarray is picked and moved to the beginning of unsorted subarray. After every iteration sorted subarray size increase by one and unsorted subarray size decrease by one.

ALGORITHM :

step 1: start

Step2: call rand_num() function

Step 2: create rand_num file and store the random numbers in it.

Step3: open rand_num file in reading mode

Step 4: Store all random numbers in an array

```
Step5: Traverse all elements using for loop take n as 100
Step6: Perform insertion and selection sort on each block of 100 numbers
Step7: Calculate time required to perform insertion and selection sort at each iteration
Step8: Increment n by 100
Step 9: If n reaches 1000 then end else go to step 6 rand_num() function:
step 1: start
step 2: crate the file pointer
step 3: open the file in writing mode
step 3: starts the loop from 0 to 100000
step 4: insert the 100000 random numbers in the file
step 5: close the file handle
step 6: end
```

<u>Insertion sort:</u>

```
Step 1: start
Step 2: start the loop from 1 to n
Step 3: initialize j with i-1
Step 4: current element is array(i)
Step 5: if array(key)>0 and j>=0
Repeat below steps 6,7
Step 6: j+1th element will jth element
 Step 7: decrement j
Step 8: array(j+1) = current.
Step 9: end.
```

<u>Selection sort:</u>

```
step 1: start
step 2: start the loop
step 3: initialize the min element
step 4: start the loop from i+1 to n
step 5: check the condition: if jth element less than min element then minimum element will be j.
```

step 6: if minimum element not equal to i, then initialize variable t with array(i) perform ith element = array of min array(min) = t

step 7: end.

PROGRAM: Program to generate one lakh random numbers from 1 to 100000 .

```
#include <stdio.h>
#include <stdlib.h>
int main() {
  int c, n;
  printf("one lakh random numbers in [1,100000]\n");
  for (c = 1; c <= 100000; c++) {
    n = rand() % 100000 + 1;
    printf("%d\n", n);
  }
  return 0;
}
void insertionSort(int array[], int n) {
int i, element, j;
 for (i = 1; i < n; i++) { element = array[i];j = i - 1; while (j >= 0 && array[j] > element) {
 array[j + 1] = array[j];
j = j - 1;
}
array[j + 1] = element;
}
}
void swap(int *xp, int *yp)
 {
int temp = *xp;
 *xp = *yp;
*yp = temp;
}
```

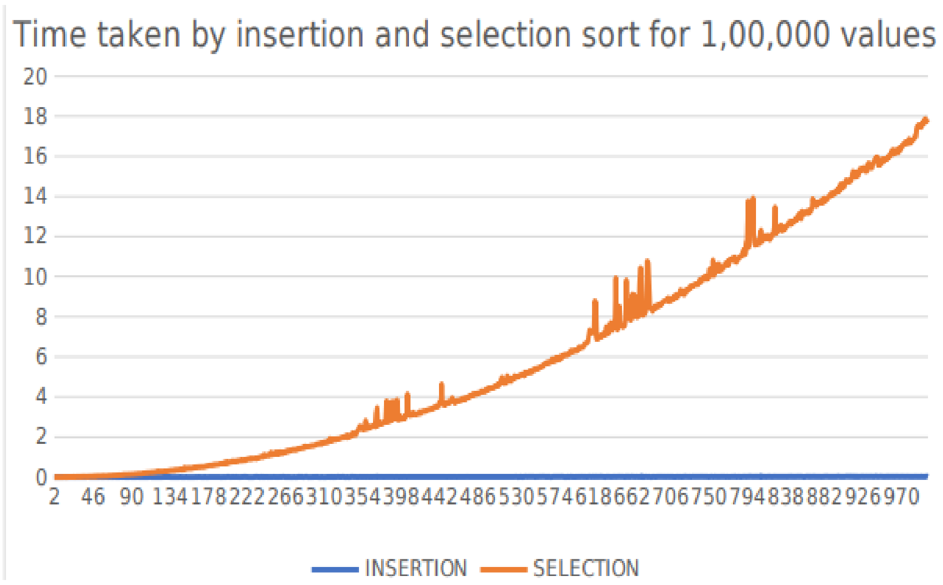
```
void selection Sort(int arr[], int n)
{
int i, j, min_idx;
for (i = 0; i < n-1; i++)
{
min_idx = i;
for (j = i+1; j < n; j++)
if (arr[j] < arr[min_idx])
min_idx = j;
if(min_idx != i)
swap(&arr[min_idx], &arr[i]);
}
}
int main()
{
int *x1;
int i,i1;
int arr[100000];
x1 = rand_num();
FILE *fp;
int ch;
fp = fopen("rand_num.txt","w");
for (i1 = 0; i1 < 100000; ++i1){
fprintf(fp,"%d\n",*(x1 + i1));
}
fp = fopen("rand_num.txt", "r");
for ( i = 0; i < 100000; i++)
{
fscanf(fp,"%d\n",&arr[i]);
}
FILE *file = fopen("output.txt","w");
```

```
int num = 100;

for ( i = 0; i < 1000; i++)
```

```
{
clock_t t1 = clock();
insertionSort(arr,num);
clock_t t2 = clock();
clock_t t3 = clock();
selectionSort(arr,num);
clock_t t4 = clock();
double insertion_time =
(double)(t2-t1)/(double)CLOCKS_PER_SEC;
double selection_time =
(double)(t4-t3)/(double)CLOCKS_PER_SEC;
fprintf(file,"%d\t",i+1);
fprintf(file,"%f\t",insertion_time);
fprintf(file,"%f\n",selection_time);
num += 100;
}
fclose(fp);
fclose(file);
return 0;
 }
```

RESULT :



Observation : As we can see from the above graph , Insertion sort always takes 0 to 1 ms whereas time taken to perform selection sort increases as we keep adding 100 numbers to it . Hence insertion sort is better than selection sort .

CONCLUSION: After performing the experiment , we find out that Insertion sort is a better sorting algorithm as compared to Selection sort .