

Shubhan Singh

SE-Comps B/Batch C

2022300118

DAA Experiment 1

Aim – Experiment on finding the running time of an algorithm.

Details – The understanding of running time of algorithms is explored by implementing two basic sorting algorithms namely Insertion and Selection sorts.

Input – Each student have to generate random 100000 numbers using rand() function and use this input as 1000 blocks of 100 integer numbers to Insertion and Selection sorting algorithms.

Output –

- 1) Store the randomly generated 100000 integer numbers to a text file.
- 2) Draw two 2D plot of all functions such that 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.
- 3) Comment on Space complexity for two sorting algorithms

Source code(C language):

- I have printed the output in a csv file so that the data can be easily imported into excel.

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

#define SIZE 100000
FILE* file;

void runinsertionsort(int *arr, int len){
    int temp[len];
    for(int i=0;i<len;i++){
        temp[i]=arr[i];
    }
    clock_t time=clock();
    int key,j;
    for(int i=1;i<len;i++){
        key=temp[i];
        j=i;
        while(temp[j-1]>key && j>0){
            temp[j]=temp[j-1];
            j--;
        }
        temp[j]=key;
    }
}
```

```

    time=clock()-time;
    double tme=(double)time/CLOCKS_PER_SEC;
    fprintf(file,"%lf,",tme);
}

void runselectionsort(int *arr, int len){
    int temp[len];
    for(int i=0;i<len;i++){
        temp[i]=arr[i];
    }
    clock_t time=clock();
    int max,ind,tem;
    for(int i=len-1;i>=0;i--){
        max=INT_MIN;
        for(int j=0;j<=i;j++){
            if(temp[j]>max){
                max=temp[j];
                ind=j;
            }
        }
        tem=temp[i];
        temp[i]=temp[ind];
        temp[ind]=tem;
    }
    time=clock()-time;
    double tme=(double)time/CLOCKS_PER_SEC;
    fprintf(file,"%lf\n",tme);
}

```

```

}

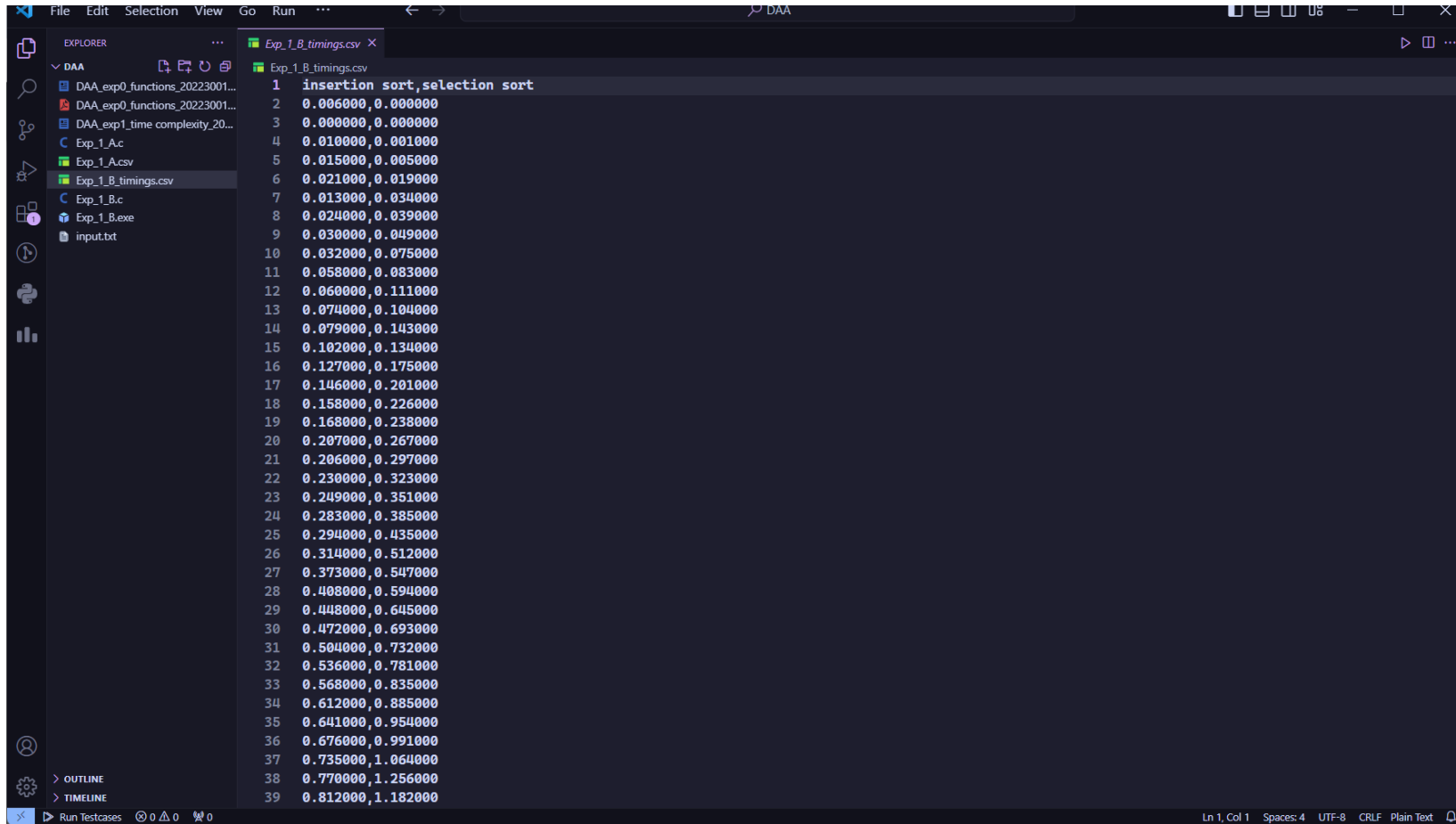
int main(){
    if(fopen("input.txt","r")==NULL){
        srand(time(NULL));
        file=fopen("input.txt","w");
        for(int i=0;i<SIZE;i++){
            fprintf(file,"%d ",rand());
        }
        fclose(file);
    }
    file=fopen("input.txt","r");
    int arr[SIZE];
    for(int i=0;i<SIZE;i++){
        fscanf(file,"%d",&arr[i]);
    }
    fclose(file);
    file=fopen("Exp_1_B_timings.csv","w");
    fprintf(file,"insertion sort,selection sort\n");
    for(int i=1000;i<=SIZE;i+=1000){
        runinsertionsort(arr,i);
        runselectionsort(arr,i);
    }
    fclose(file);

    return 0;
}

```

Output:

File created



The screenshot shows a code editor with a file explorer on the left and a code editor on the right. The file explorer shows a project named 'DAA' with several files and folders. The code editor is open to a file named 'Exp_1_B_timings.csv'. The file contains a list of 39 lines of data, each representing a comparison between insertion sort and selection sort. The first line is a header: '1 insertion sort,selection sort'. The subsequent lines are numbered 2 through 39 and contain two decimal values separated by a comma, representing the execution times for each sort algorithm.

Line	insertion sort	selection sort
1	insertion sort	selection sort
2	0.006000	0.000000
3	0.000000	0.000000
4	0.010000	0.001000
5	0.015000	0.005000
6	0.021000	0.019000
7	0.013000	0.034000
8	0.024000	0.039000
9	0.030000	0.049000
10	0.032000	0.075000
11	0.058000	0.083000
12	0.060000	0.111000
13	0.074000	0.104000
14	0.079000	0.143000
15	0.102000	0.134000
16	0.127000	0.175000
17	0.146000	0.201000
18	0.158000	0.226000
19	0.168000	0.238000
20	0.207000	0.267000
21	0.206000	0.297000
22	0.230000	0.323000
23	0.249000	0.351000
24	0.283000	0.385000
25	0.294000	0.435000
26	0.314000	0.512000
27	0.373000	0.547000
28	0.408000	0.594000
29	0.448000	0.645000
30	0.472000	0.693000
31	0.504000	0.732000
32	0.536000	0.781000
33	0.568000	0.835000
34	0.612000	0.885000
35	0.641000	0.954000
36	0.676000	0.991000
37	0.735000	1.064000
38	0.770000	1.256000
39	0.812000	1.182000

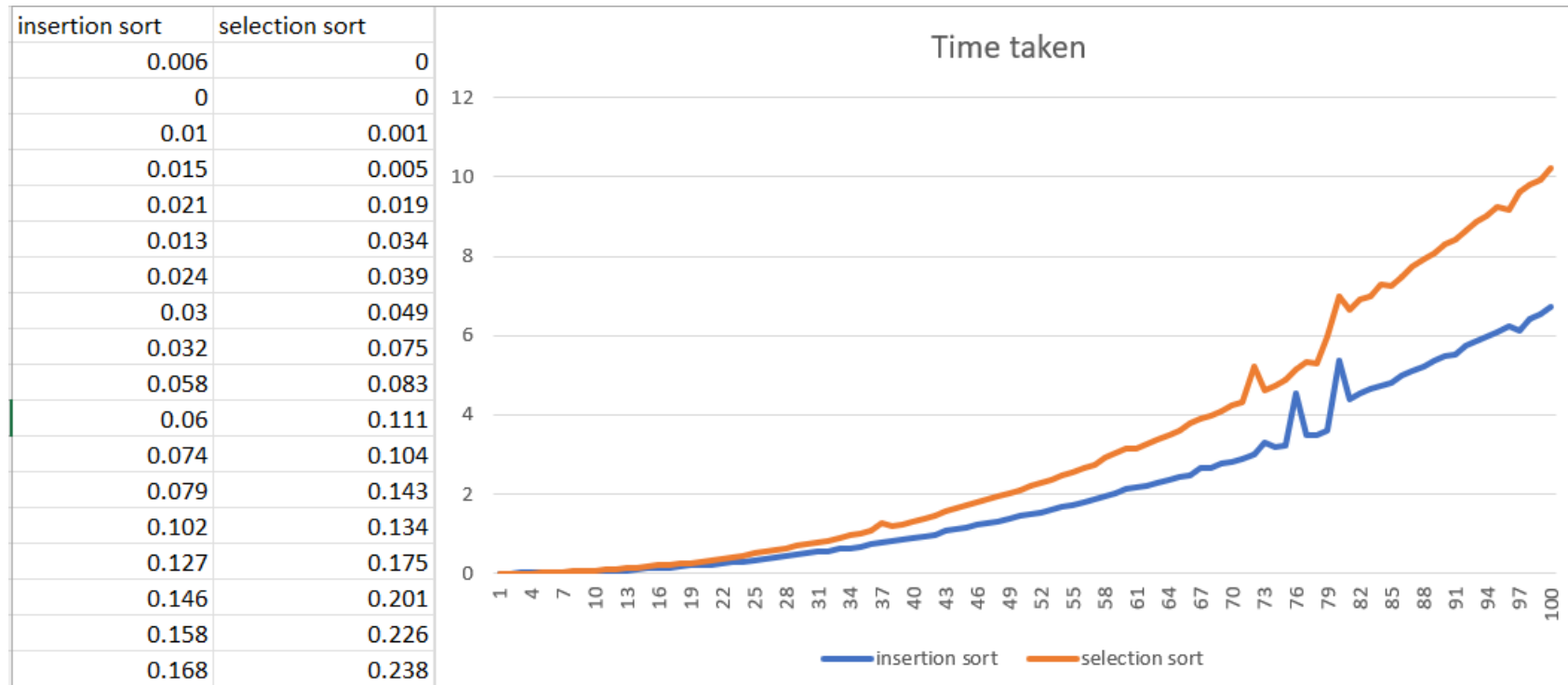
Microsoft Excel interface showing a spreadsheet titled "Exp_1_B_timings". The ribbon includes File, Home, Insert, Page Layout, Formulas, Data, Review, View, Automate, and Help. The Home ribbon is active, displaying options for Clipboard, Font, Alignment, Number, Styles, Cells, Editing, and Add-ins.

The spreadsheet data is as follows:

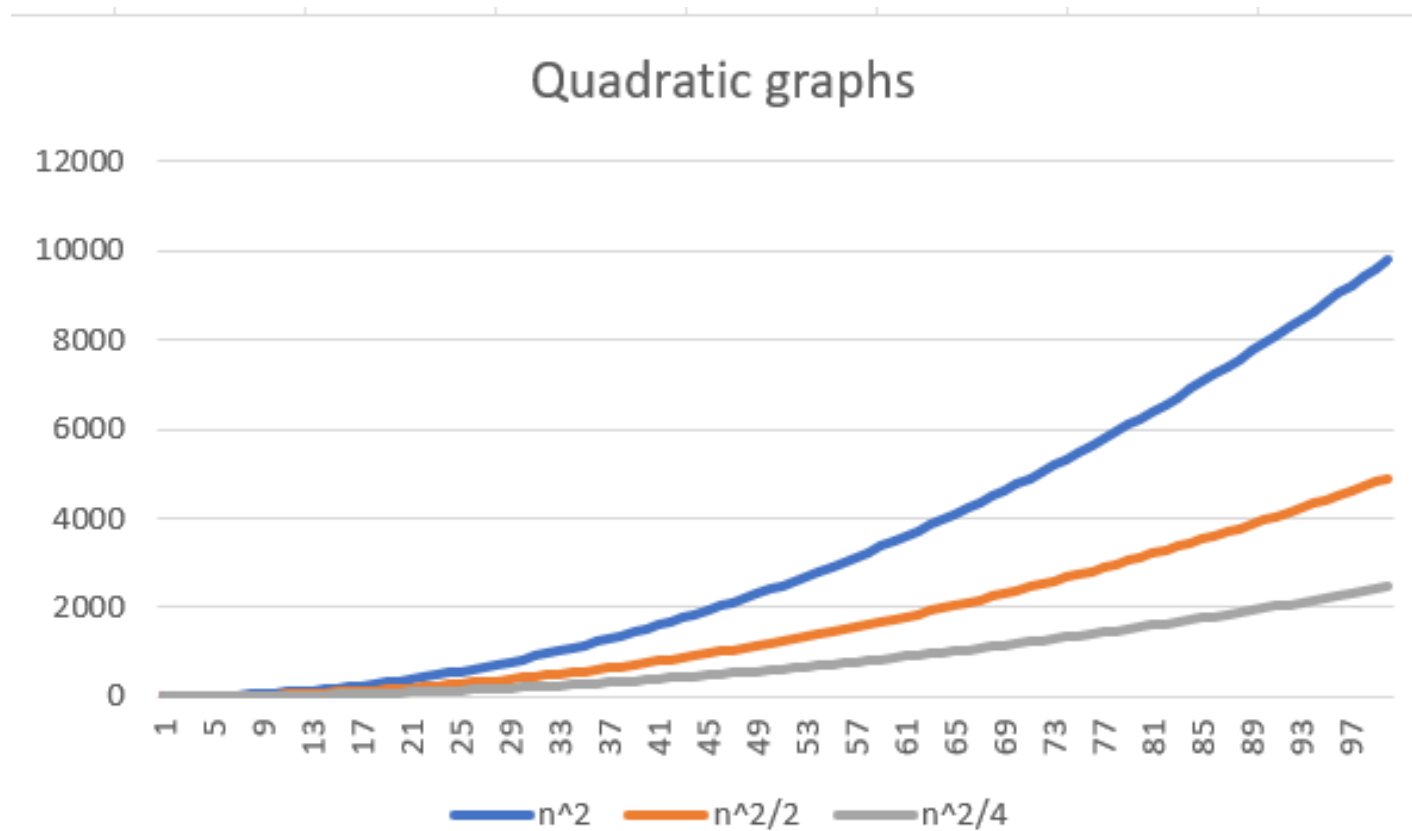
	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T
1	insertion sort	selection sort																		
2	0.006	0																		
3	0	0																		
4	0.01	0.001																		
5	0.015	0.005																		
6	0.021	0.019																		
7	0.013	0.034																		
8	0.024	0.039																		
9	0.03	0.049																		
10	0.032	0.075																		
11	0.058	0.083																		
12	0.06	0.111																		
13	0.074	0.104																		
14	0.079	0.143																		
15	0.102	0.134																		
16	0.127	0.175																		
17	0.146	0.201																		
18	0.158	0.226																		
19	0.168	0.238																		
20	0.207	0.267																		
21	0.206	0.297																		
22	0.23	0.323																		
23	0.249	0.351																		
24	0.283	0.385																		
25	0.324	0.425																		

CSV file

PLOTS:



Graph of time taken by both algorithms to sort 100 blocks of 1000,2000...100000 elements



Graphs of quadratic functions

Conclusion:

- Since the graphs of the running times of the sorting algorithms with respect to the input size are similar to graphs of quadratic functions, it can be said that insertion and selection sort have $O(n^2)$ time complexity.

Algorithm:

1. runinsertionsort function

- Create a copy of the input array arr into temp.
- Start a timer.
- For each element in temp starting from the second element:
 - Set key to the current element and j to the current index.
 - While the previous element is greater than key and j is not at the start of the array:
 - Move the previous element to the current position.
 - Decrement j.
 - Place key at the position j.
- Stop the timer and calculate the elapsed time.
- Write the elapsed time to the file.

2. runselectionsort function

- Create a copy of the input array arr into temp.
- Start a timer.

- For each element in temp starting from the last element:
 - Set max to the smallest possible integer.
 - For each element up to the current index:
 - If the current element is greater than max:
 - update max and remember the index of the maximum element, then swap the maximum element with the element at the current index
- stop the timer, calculate the elapsed time, and write the elapsed time to the file.

3. Main()

1. Check if the file "input.txt" exists.
 - If it does not exist:
 - Seed the random number generator with the current time.
 - Open "input.txt" for writing.
 - Generate SIZE random numbers and write them to the file.
 - Close the file.
2. Open "input.txt" for reading.
3. Read SIZE integers from the file into the array arr.
4. Close the file.
5. Open "Exp_1_B_timings.csv" for writing.
6. Write the header line "insertion sort,selection sort" to the file.

7. For each i from 1000 to SIZE in steps of 1000:
 - Call `runinsertionsort` with `arr` and i as arguments.
 - Call `runselectionsort` with `arr` and i as arguments.
8. Close the file.
9. Return 0 to indicate successful execution of the program.

Theory:

Insertion Sort:

Insertion sort is a simple sorting algorithm that builds the final sorted array one item at a time. It is much less efficient on large lists than more advanced algorithms such as quicksort, heapsort, or merge sort. However, insertion sort provides several advantages:

- It is simple to implement.
- It is efficient for small data sets, much like other quadratic sorting algorithms
- It is more efficient in practice than most other simple quadratic (i.e., $O(n^2)$) algorithms such as selection sort or bubble sort.
- It is stable i.e., it does not change the relative order of elements with equal keys.
- It is online i.e., it can sort a list as it receives it.

Time Complexity:

- Best Case: $O(n)$ when input is already sorted
- Average and Worst Case: $O(n^2)$ when input is reverse sorted or random

Space Complexity: $O(1)$ as it is an in-place sorting algorithm.

Selection Sort:

Selection sort is a simple in-place comparison sorting algorithm. 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 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: a sorted sublist of items which is built up from left to right at the front (left) of the list and a sublist of the remaining unsorted items 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.

Time Complexity:

- Best, Average and Worst Case: $O(n^2)$ irrespective of the nature of the input.

Space Complexity: $O(1)$ as it is an in-place sorting algorithm.

Library functions used in this program:

1. **time**: This function is part of the standard C library and is used to get the current calendar time. When called with NULL as its argument, it returns the current calendar time (the number of seconds since the Epoch, 1970-01-01 00:00:00 +0000, UTC). This value is often used to seed the random number generator (as seen in the srand function).
2. **clock**: This function is also part of the standard C library and is used to calculate the processor time used by the program. The clock function returns the amount of CPU time used since the program started. To convert this to seconds, you divide by CLOCKS_PER_SEC.
3. **CLOCKS_PER_SEC**: This is a constant defined in time.h and it represents the number of processor clock ticks per second. It is used to convert the value returned by clock into seconds by dividing the clock ticks by CLOCKS_PER_SEC.

4. **srand**: This function is used to seed the random number generator in C. It should only be called once at the beginning of the program. If srand is not called, the rand function will use a default seed value. In this code, srand is called with the current time (time(NULL)) as the seed. This means that the random numbers generated by rand will be different each time the program is run.
5. **rand**: This function returns a pseudo-random number in the range of 0 to RAND_MAX. The sequence of numbers generated by rand is determined by the seed given to srand. If rand is called before any calls to srand have been made, the same sequence will be generated each time the program is run

Time and space complexities of insertion and selection sort:

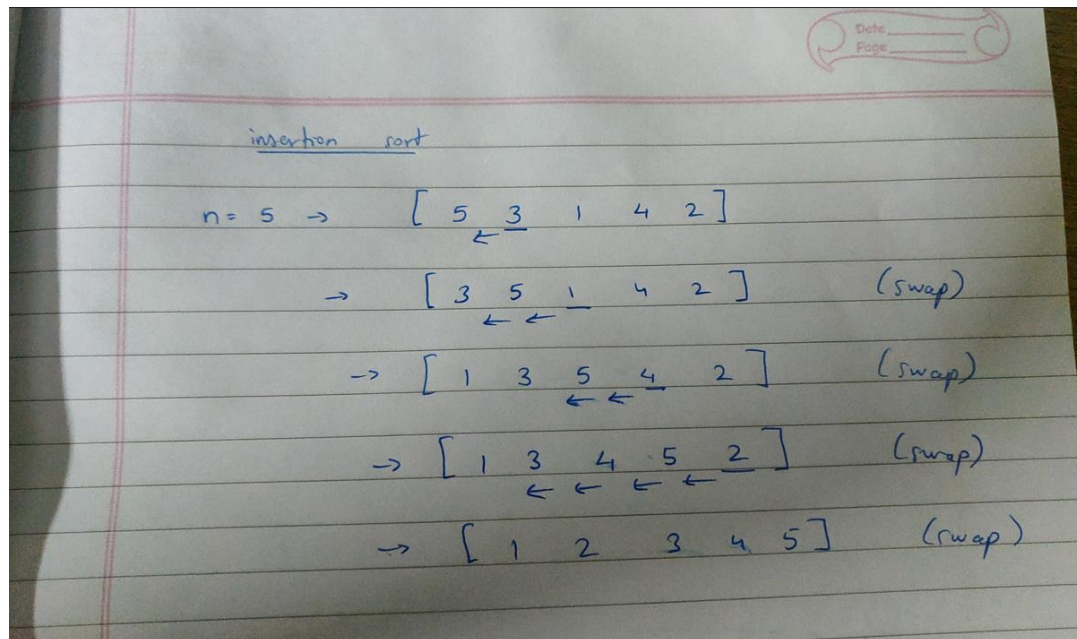
Insertion Sort:

Time Complexity:

- Best Case: $O(n)$ when input is already sorted
- Average and Worst Case: $O(n^2)$ when input is reverse sorted or random

Space Complexity: $O(1)$ as it is an in-place sorting algorithm.

Here's a simple visualization of the time complexity of Insertion Sort:



It can be seen that no auxiliary space is needed for the execution of insertion sort, so the space complexity of insertion sort is $O(1)$

Also, in the worst case, each while loop runs for $i-1$ times for i going from 1 to n , so the time complexity of insertion sort is $O(n^2)$ in the worst case. It is the same for average case too, as the number of times the inner while loop runs increases proportionally to n .

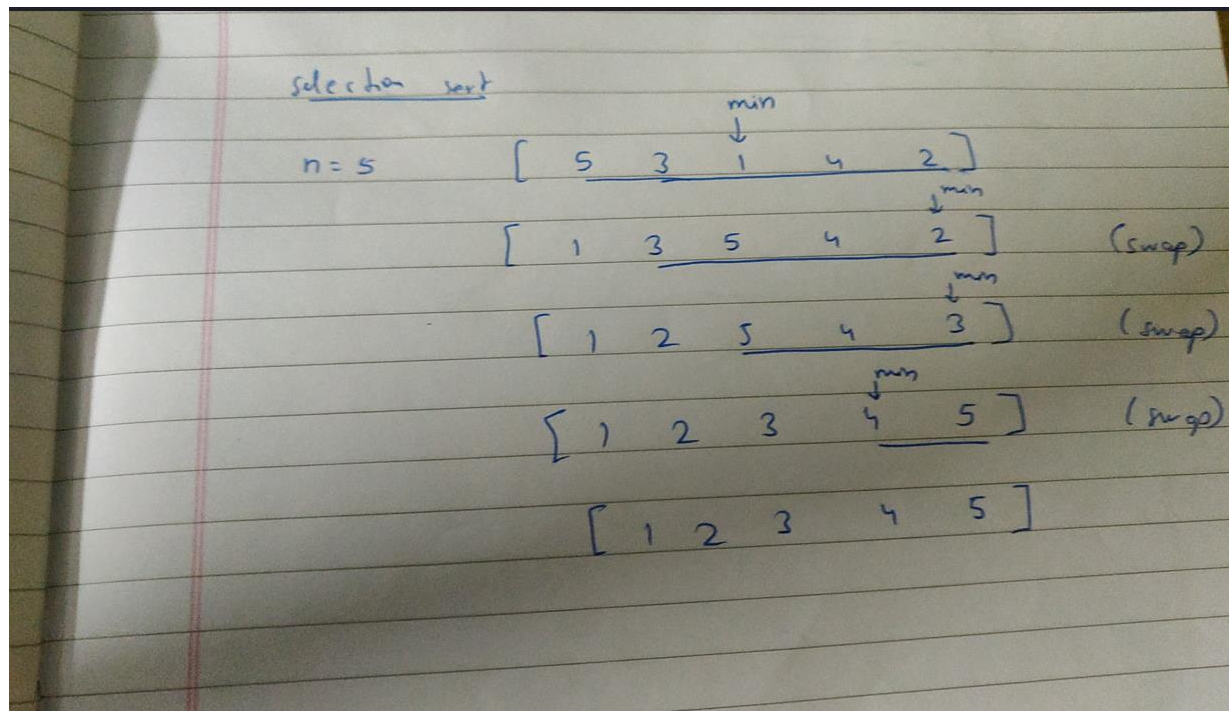
Selection Sort:

Time Complexity:

- Best, Average and Worst Case: $O(n^2)$ irrespective of the nature of the input.

Space Complexity: $O(1)$ as it is an in-place sorting algorithm.

Here's a simple visualization of the time complexity of Selection Sort:



Even though the number of swaps in Selection Sort is less than in Insertion Sort, the time complexity is still $O(n^2)$ because it always goes through the entire list to find the minimum element.

The space complexity of selection sort too is $O(1)$ as it does not need auxiliary space. The time complexity for best, worst and average case in this algorithm is $O(n^2)$ because we always must traverse the entirety of the remaining $n-i$ elements to find the minimum element.