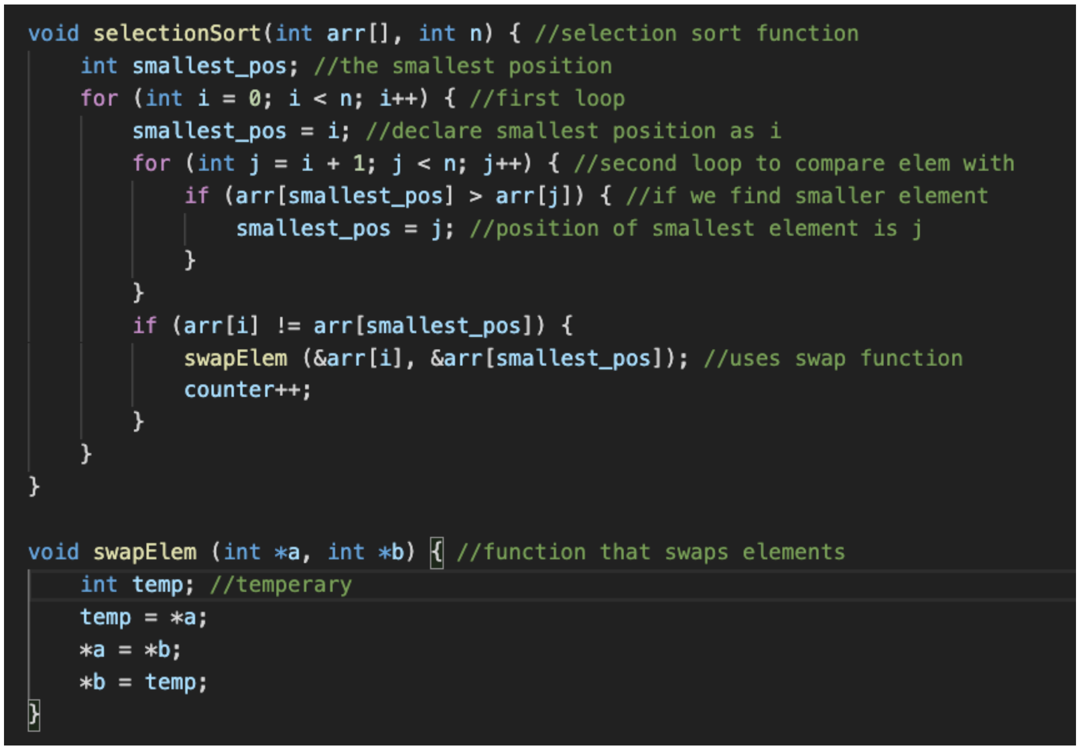
Assignment 3.2

Nayan Man Singh Pradhan

a.

Selection sort has been implemented in the c++ file named "selectionSort.cpp". The c++ program contains a function called selectionSort that has the array and the number of terms in the array as two parameters. A swapElem function is created that swaps the two elements passed by reference (using pointers) in the original array.



```
void selectionSort(int arr[], int n) { //selection sort function
    int smallest_pos; //the smallest position
    for (int i = 0; i < n; i++) { //first loop
        smallest_pos = i; //declare smallest position as i
        for (int j = i + 1; j < n; j++) { //second loop to compare elem with
            if (arr[smallest_pos] > arr[j]) { //if we find smaller element
                smallest_pos = j; //position of smallest element is j
            }
        }
        if (arr[i] != arr[smallest_pos]) {
            swapElem (&arr[i], &arr[smallest_pos]); //uses swap function
            counter++;
        }
    }
}

void swapElem (int *a, int *b) { //function that swaps elements
    int temp; //temperary
    temp = *a;
    *a = *b;
    *b = temp;
}
```

b.

Running the program "./selectionSort" shows that the implementation of selection sort is correct (a makefile is made). There are two loops in our in our "selectionSort" function. The first for loop is a cursor that points to the element that needs to be sorted and compared. The second for loop is a cursor that points to the element that needs to be compared to the first cursor. If the element in the second cursor is smaller than the element in the first cursor, it's position is stored, and if no other elements are smaller, a swap is performed. After the second cursor moves throughout the array (checks the array for smallest term), the first cursor moves a position forward and the process continues until the whole array is sorted. Considering the loop invariant for the outer loop,

- Initialization: All elements in the right of the first cursor is always unsorted.
- Maintenance: All elements to the left of the first cursor is sorted and to the right of the first cursor is unsorted.
- Termination: All elements are sorted.

Considering the loop invariant for the inner loop,

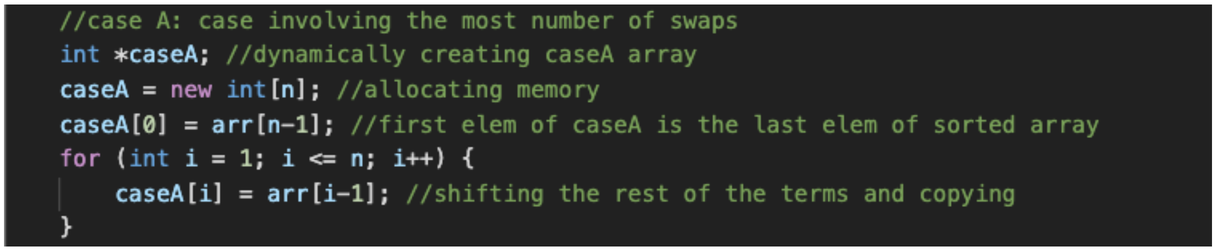
- Initialization: All elements to the left is larger than or equal to the arr[smallest_pos].
- Maintenance: All elements to the left is larger than or equal to the arr[smallest_pos] and everything to the right needs to be checked.

- Termination: No more elements need to be checked and the smallest element has been found.

c.

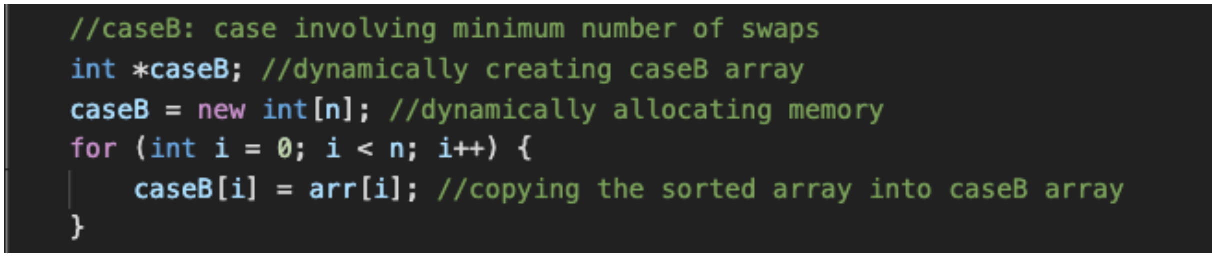
In the program “./selectionSort”, we have three arrays. The first one is the original array. This array consists of “n” number of elements, where “n” needs to be entered by the user. Then, an array of “n” random elements is created! A global variable “counter” is created to monitor the number of swaps for an array. Two other arrays: “Case A” and “Case B” are also created with “n” elements. Case A represents the case with most swaps. The case with most swaps is the case where the array contains the largest element in the first position, and all the other array elements in ascending order from the second position. This is because the selection sort algorithm has to compare and swap n-1 elements. Case B represents the case with the least number of swaps. This is when the array is already arranged and 0 swaps are required.

The array for caseA is made by copying the last element of the sorted array into the first element of the caseA array, then shifting all the elements of the array.



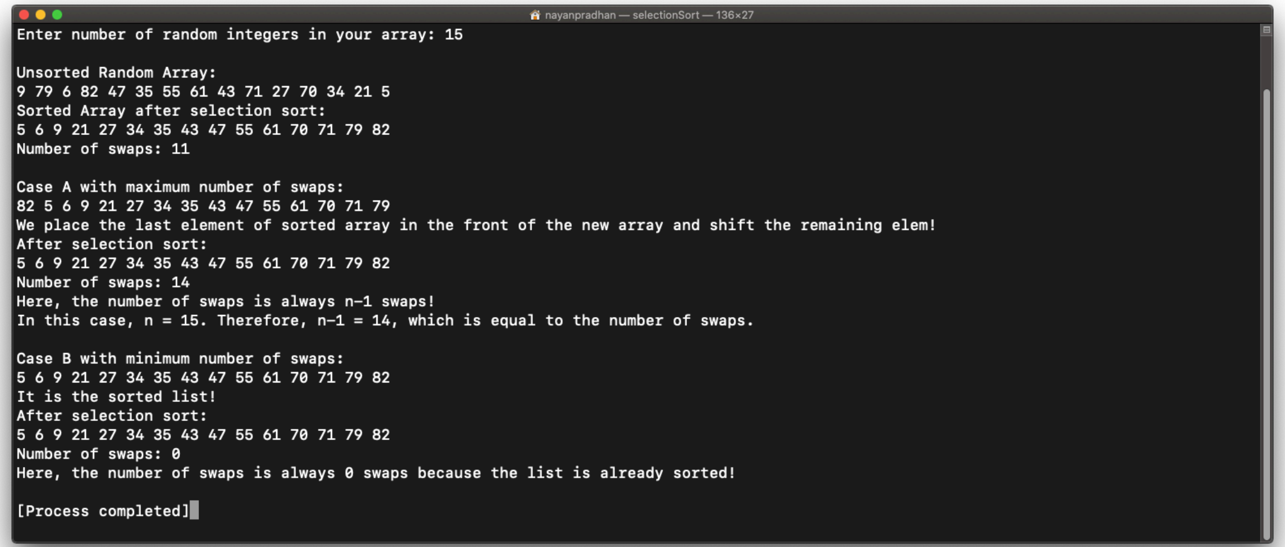
```
//case A: case involving the most number of swaps
int *caseA; //dynamically creating caseA array
caseA = new int[n]; //allocating memory
caseA[0] = arr[n-1]; //first elem of caseA is the last elem of sorted array
for (int i = 1; i <= n; i++) {
    caseA[i] = arr[i-1]; //shifting the rest of the terms and copying
}
```

The array for caseB is simply just a copy of the sorted array.



```
//caseB: case involving minimum number of swaps
int *caseB; //dynamically creating caseB array
caseB = new int[n]; //dynamically allocating memory
for (int i = 0; i < n; i++) {
    caseB[i] = arr[i]; //copying the sorted array into caseB array
}
```

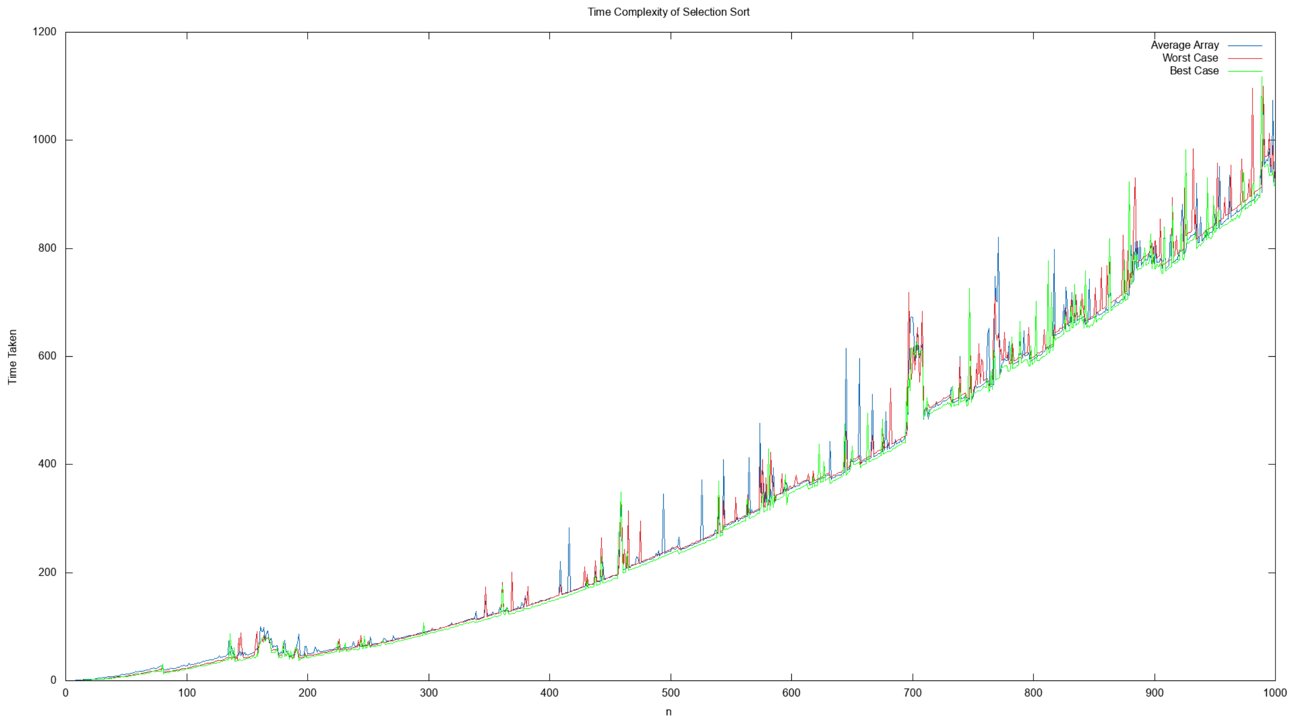
The output of a random array with n = 15 is illustrated below.



```
Enter number of random integers in your array: 15

Unsorted Random Array:
9 79 6 82 47 35 55 61 43 71 27 70 34 21 5
Sorted Array after selection sort:
5 6 9 21 27 34 35 43 47 55 61 70 71 79 82
Number of swaps: 11

Case A with maximum number of swaps:
82 5 6 9 21 27 34 35 43 47 55 61 70 71 79
We place the last element of sorted array in the front of the new array and shift the remaining elem!
After selection sort:
5 6 9 21 27 34 35 43 47 55 61 70 71 79 82
Number of swaps: 14
Here, the number of swaps is always n-1 swaps!
In this case, n = 15. Therefore, n-1 = 14, which is equal to the number of swaps.

Case B with minimum number of swaps:
5 6 9 21 27 34 35 43 47 55 61 70 71 79 82
It is the sorted list!
After selection sort:
5 6 9 21 27 34 35 43 47 55 61 70 71 79 82
Number of swaps: 0
Here, the number of swaps is always 0 swaps because the list is already sorted!

[Process completed]
```

d.

A new c++ file named “3_2d.cpp” is made that is an edited version of the “selectionSort.cpp” file. In this file, the user inputs the maximum size of the array he/she wants to compute as well as the interval size. Then the algorithm generates “n” random arrays with size of array from 1 to n with interval (n = n + interval). This program takes into account the time taken for the selection sort using the chrono standard library. We print the serial number, time taken to sort arr (average case), time taken to sort caseA (worst case), and time taken to sort caseB (best case) into a “data.txt” file. Then, we use Gnuplot to plot the data in the “data.txt” file into a graph. The file “plot.plt” handles the plotting part of the assignment, making a png file “plot.png”. I used n = 1000 with interval = 1 while running the test case. Using higher values of n with small intervals would lead to large processing time.



Plot with n = 1000 and interval = 1. *Note that you have to type “make plot”(I have it in my makefile) in terminal to generate a new plot after you run ./3_2d and enter your n and interval.

e.

Above is an example of a graph plotted when the maximum n value = 1000 with an interval = 1. We notice that after a certain point (n_o), the blue line (average case) is tightly bounded by the red line (worst case A) and the green line (best case B). (Also notice that we see some cases when the average case takes longer to compute than the worst case!) Therefore, we see that our average case line (let us call it $f(n)$) is asymptotically tightly bounded by the worst case line (let us call it $c_1 . g(n)$) and the best case line (let us call it $c_2 . g(n)$). Through further investigation and playing with graphs, we notice that $g(n)$ behaves like a x^2 graph.
Therefore, we have a **Θ-Notation** such that, $f(n) = \Theta . (g(n))$.ß
Hence, $f(n) = \Theta . (n^2)$.