**Problem 1.1:**

**F = Ω(g)**

**F = O(g)**

**F = O(g)**

**Problem 1.2 :**

Selection sort program in C++ :

|  |
| --- |
| int min\_finder(int \*arr, int firstidx, int size){  int min = arr[firstidx];  int idx = firstidx;  for (int i = firstidx; i < size; i++){  if ( arr[i] < min ){  min = arr[i];  idx = i;  }  }  return idx; } |

|  |  |
| --- | --- |
| void selection\_sort(int \*arr, int size){  int smallestidx;  for (int lim = 0; lim < size; lim++){  smallestidx = min\_finder(arr, lim, size);  swap(arr[lim], arr[smallestidx]);  } }  **Proof that algorithm works:**  In order for me to show that the algorithm works, I decided to feed an unsorted array and added a function that shows the changes after every iteration, I also implemented a little check to make sure that our current element is smaller than the first one.  void selection\_sort(int \*arr, int size){  int smallestidx;  for (int lim = 0; lim < size; lim++){  smallestidx = min\_finder(arr, lim, size);  swap(arr[lim], arr[smallestidx]);  // Checks that function is working  if ( arr[0] > arr[lim] ) break;  cout << "[ ";  for (int i = 0; i < size; i++){  cout << arr[i] << " ";  }  cout << "]" << endl;  }   |  | | --- | | int main () {  // Test program  int size = rand()%10;  int array[size];  for (int i = 0; i < size; i++){  array[i] = rand()%10;  }  selection\_sort(array, size);  return 0; } |   **OUTPUT :**  [ 0 3 8 9 2 4 8 ]  [ 0 2 8 9 3 4 8 ]  [ 0 2 3 9 8 4 8 ]  [ 0 2 3 4 8 9 8 ]  [ 0 2 3 4 8 9 8 ]  [ 0 2 3 4 8 8 9 ]  [ 0 2 3 4 8 8 9 ]  Program ended with exit code: 0 |
| In this example we can see how the swapping works and that the array gets more arranged after every step. |

**Random sequence generator:**

The sequence generator I built is in c++, using the cstdlib library and the rand() function.

First I generate a random n arrays of n (0 to 9) elements and I feed it random values. It repeats this process 10 times and then increases the range of the random numbers (from 10 to 100) it repeats the same process 100 times and then increases the range on last time from 100 to 1000 elements and repeats the process 1000 times. A clock is also reset every time an array is sorted in order to calculate the runtime. It then saves the array size and runtime in a text file generating a total of 1110 cases.

int main(int argc, const char \* argv[]) {

ofstream outfile ("/Users/otmanesabir/Desktop/randomcases.txt");  
 int mult = 1;  
 for (int j = 0; j < 3; j++){  
 mult = mult\*10;  
 for (int i = 0; i < mult; i++){  
 int size = rand()%mult;  
 int array[size][size];  
 for (int j = 0; j < size; j++){  
 for (int i = 0; i < size; i++){  
 array[j][i] = rand()%mult;  
 }  
 }  
 clock\_t tStart = clock();  
 for (int j = 0; j < size; j++){  
 selection\_sort(array[j], size);  
 }  
 outfile << size << "," << (double)(clock() - tStart)/CLOCKS\_PER\_SEC << endl;  
 }  
 }  
 outfile.close();  
 return 0;  
}

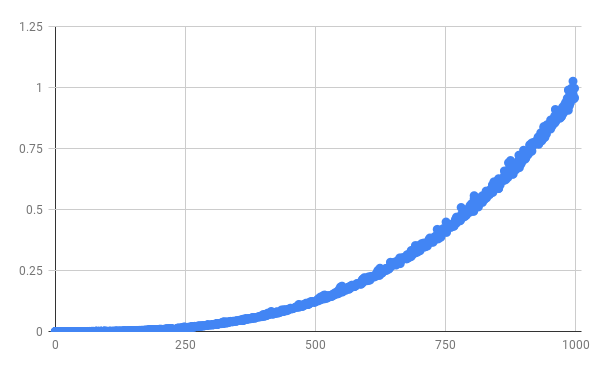
In order to find the best case, I added a function that iterates through all the arrays and sorts them before starting the clock. It then does the same thing as the first function and returns the array sizes and runtime in a text file.

int main(int argc, const char \* argv[]) {  
 ofstream outfile ("/Users/otmanesabir/Desktop/bestcases.txt");  
 int mult = 1;  
 for (int j = 0; j < 3; j++){  
 mult = mult\*10;  
 for (int i = 0; i < mult; i++){  
 int size = rand()%mult;  
 int array[size][size];  
 for (int j = 0; j < size; j++){  
 for (int i = 0; i < size; i++){  
 array[j][i] = rand()%mult;  
 }  
 }  
 for (int i = 0; i < size; i++){  
 selection\_sort(array[i], size);  
 }  
 clock\_t tStart = clock();  
 for (int j = 0; j < size; j++){  
 selection\_sort(array[j], size);  
 }  
 outfile << size << "," << (double)(clock() - tStart)/CLOCKS\_PER\_SEC << endl;  
 }  
 }  
 outfile.close();  
 return 0;  
}

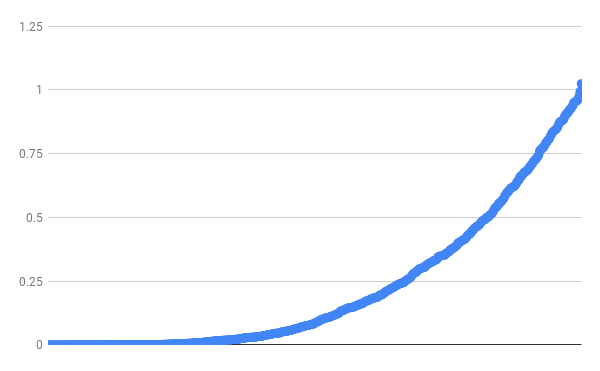
For the worst case, I made a function that inverts the sorted array by swapping all elements of the array and by running the selection sort algorithm and exporting the results to a file.

|  |
| --- |
| int main(int argc, const char \* argv[]) {  ofstream outfile ("/Users/otmanesabir/Desktop/worstcases.txt");  int mult = 1;  for (int j = 0; j < 3; j++){  mult = mult\*10;  for (int i = 0; i < mult; i++){  int size = rand()%mult;  int array[size][size];  for (int j = 0; j < size; j++){  for (int i = 0; i < size; i++){  array[j][i] = rand()%mult;  }  }  for (int i = 0; i < size; i++){  worstcase(array[i], size);  }  clock\_t tStart = clock();  for (int j = 0; j < size; j++){  selection\_sort(array[j], size);  }  outfile << size << "," << (double)(clock() - tStart)/CLOCKS\_PER\_SEC << endl;  }  }  outfile.close();  return 0; }  After running these functions, I was able to gather 3330 cases (1110 for average, best & worst case which you can find the test cases in the xlsx sheet) and draw 3 curves for each of the test sets.  ***Average cases graph :*** |

***Best cases graph :***

******

***Worst cases :***

******

**In our selection sort algorithm, we use nested loops to iterate through the array, find the minimum and swap it.**

The total running time for selection sort has three parts:

1. The running time for all the calls to min\_finder.
2. The running time for all the calls to swap.
3. The running time for the rest of the loop in the selection\_sort function.

For the first part we’re using loops and we know that each iteration of a loop takes constant time, so the number of iterations in this first call is n times then n-1, n-2 until we reach 1. This sum is an arithmetic series that we can turn to (n+1)/(n/2). We can ignore the ½ factor and also the “n” which leaves us with an n^2. So the results of running time in big-o-notation are O(n^2). This for min\_finder function.

For the second part, we use the swap function and there are n swaps and each call takes constant time. Therefore using our asymptotic notation the time for the swap function is O(n). The selection sort is only used for testing and incrementing the loop, therefore it also takes O(n).

Adding up all three parts together we get O(n^2 + 2n) we can ignore the 2n again which leaves us with O(n^2).

Let's test the runtime and start with let's say n = 100. Then the running time of selection sort is about 100^2/10^6 = 1/100 seconds. That seems pretty fast. But what if n = 1000 then selection sort takes about 1000^2/10^6 = 1second. The array grew by a factor of 10, but the running time increased 100 times. What if n = 1,000,000 ? Then selection sort takes 1,000,000^2/10^6 = 1,000,000 seconds, which is a little more than 11.5 days. Which shows how ineffective the algorithm is in big data sets.