Assignment 2

Batuhan Faik Derinbay derinbay18@itu.edu.tr
150180705

Part 2

Question 1)

The simulation starts by generating a random real value from a uniform distribution $[0, 1]$. It is done within templates defined in `main.cpp` and uses built-in libraries of the C++ that are publicly available. If the returned real value is less than p (it means that the simulation has a probability of executing this statement p or in other words $p \times 100\%$) a random index to update the taxi within the heap is selected similarly. Then a random taxi's distance is decreased by 0.01 (taxi updated). On the other hand, if the returned real value is greater than p, then a coordinates of a new taxi is read from the file and a taxi object is added to the queue. This continues for m number of operations and each operation that is a multiplicate of 100 calls a taxi, which removes the closest taxi from the queue. The code snippet below depicts the implementation of the simulation in `main.cpp`.

Please note that for large values of p, since the taxi queue starts without any taxis inside, the simulation skips the update and counts it as an operation. This behaviour is clearly visible for values of $p > 0.95$ and when the taxi queue is empty. At the end of the simulation, the number of skipped updates is printed for the user to see. Therefore, the user can verify the number of operations (m), with the printed number of taxi additions (a), distance updates (d) and skipped updates (s) as follows: $m = a + d + s$

Implementation of the Simulation in main.cpp

```
1  // Start simulation
2  for (int i = 0; i < m; i++) {
3      rand_real = get_random_real(double(0), double(1));
4      if (rand_real <= p){    // Update taxi (decrease the distance)
5          if (!taxi_heap.get_size()) {    // If the heap is empty don't ↩
               update taxis
6              // The reverse logic here is intentional due to compiler ↩
                   optimizations
7              cout << "There are no taxis available!" << endl;
8              skipped_update++;
9          } else {
10             int rand_index = get_random_int(0, taxi_heap.get_size() - ↩
                   1);
11             taxi_heap.update_random_taxi(rand_index, 0.01);
12             distance_updates++;
13         }
```

```
14      } else {    // Read a new taxi object (add to heap)
15          file >> taxi_long; // longitude of the taxi (float)
16          file >> taxi_lat; // latitude of the taxi (float)
17          getline(file, line, '\n'); // this is for reading the \n ↩
                character into dummy variable.
18          taxi_heap.add_taxi(Taxi(taxi_long, taxi_lat, hotel_long, ↩
                hotel_lat));
19          taxi_additions++;
20      }
21      if ((i + 1) % 100 == 0) {   // Call a taxi (remove from heap)
22          if(taxi_heap.call_taxi() != -1) {    // Check if the call was ↩
                successful
23              taxi_call++;
24          }
25      }
26  }
```

There are three important priority queue features that are required by the simulation and implemented in the binary min-heap structure. Extract the root (minimum) element, insert element and decrease key of the element. Let's explain each one in details with examples from the source code.

1) Extracting the root (minimum) element is the action of calling a taxi within the simulation. Therefore its method is implemented as `double call_taxi()` which returns the distance of the closest taxi. Even though my implementation does not use the returned value and prints the distance of the taxi before deleting the object, the caller of this method can make use of the distance variable.

   Furthermore, the theoretical running time of the `call_taxi()` method is $O(\log_2 n)$ where n is the number of nodes. The reason for this is that the `min_heapify(0)` recursive function on line 14 in the code snippet below builds the heap (heapify) starting from the index 0 which is root. Some might argue that time complexity of building a heap is $O(n)$, but it is clear that the worst case is not an issue when extracting the root. Since part of the subtrees of the root are already heapified, the upper bound function for the time complexity is indeed $\log_2 n$.

call_taxi Method

```
1       double Heap::call_taxi() {
2           // There is a p^100 chance of having an empty heap, ↩
                therefore it should be considered
3           if (taxis.empty()) {
4               // The reverse logic here is intentional due to ↩
                    compiler optimizations
5               return -1;    // Return a distinguishable value in ↩
                    case the heap is empty
6           } else {
7               // Store the distance of the taxi in a local variable
8               double distance = taxis.at(0).get_distance();
9               // Print the taxi that is being called
```

```
10              taxis.at(0).print_called_taxi();
11              // Delete the taxi from the list
12              taxis.erase(taxis.begin());
13              size = taxis.size();
14              // Reorder the heap (heapify)
15              min_heapify(0);
16              return distance;
17          }
18      }
```

2) Inserting an element into the queue is the action of adding a taxi within the simulation. Therefore its method is implemented as `void add_taxi(Taxi)` which takes a Taxi object as its only parameter and adds it to the queue. After adding the taxi to the queue, the heap structure needs to check whether the newly added node (taxi) satisfies the leaf element condition, if not it needs to be carried upwards towards the root. This is called the up-heapify and the method `compare_parents()` on line 7 in the given code snippet below implements this function. The time complexity of the function is again $O(\log_2 n)$ because it bubbles up towards to root of the binary tree and is dependent to the height of the tree.

add_taxi Method

```
1       void Heap::add_taxi(Taxi new_taxi) {
2           // Add new taxi to the list
3           taxis.push_back(new_taxi);
4           size = taxis.size();
5           // Get its index
6           int index = int(taxis.size() - 1);
7           compare_parents(index);
8       }
```

3) Finally, decreasing the key of an element from the queue is the action of decreasing the distance of a taxi within the simulation. Therefore its method is implemented as `void update_random_taxi(int, double)` which takes in two parameters. The first is the index of the taxi that needs to be updated and the second is the amount that will be subtracted from the distance of the taxi. All that it does is to decrease the distance of a taxi, then up-heapify it to make sure the heap property is preserved. On the other hand, there is nothing stopping the user from entering a negative value to increase the distance of the taxi when calling this function, which is likely to falsify the heap structure. Therefore the user must be careful and read the report carefully.

   As mentioned before, line 4 in the given snippet below up-heapifies the taxi starting at its index and therefore time complexity of the method is $\log_2 n$.

update_random_taxi Method

```
1       void Heap::update_random_taxi(int index, double ↩
            decrease_distance) {
```

```
2           // Decrease the distance of the taxi at the index by 0.01
3           taxis.at(index).set_distance(taxis.at(index).get_distance↩
                () - decrease_distance);
4           compare_parents(index);
5       }
```

Question 2)

For different values of m between 1000 and 100000, execution times of the simulation is represented in Table 1 and in Figures 1-2 where $p = 0.2$. In Figure 1 you can see an exponentially upward trending execution time curve. However don't let this plot confuse you. This is an aggregated plot in order to fully capture the values of m. The plot given in Figure 2 represents the correct relationship between m and execution time. This behaviour is expected because as the number of operations increase, operations that are upper bounded by $O(\log_2 n)$ are summed.

A quick example can be given with a scenario where $p = 0$ for better understanding of the situation. When $p = 0$, there will be m number of taxi (node) additions which has a corresponding time complexity of $O(\log_2 n)$ where $n = m$ per node. With each addition of a new taxi, n grows larger and the total time taken (sum of time taken by each heap insert operation) by the execution grows faster than linearly. It can be represented mathematically in the form of sum of logarithms $t = \log_2 1 + \log_2 2 + ... + \log_2 n + c$ where t is the time required for the completion of the simulation and c a constant. Then, by using Stirling approximation, upper bound to t can be found as $O(n \log_2 n)$. Since the plot given in Figure 2 matches this upper bound, the running times of our simulation match the theoretical running times.

m	p	Time (ms)
1000	0.2	24.895
2000	0.2	50.991
4000	0.2	94.082
7000	0.2	158.982
10000	0.2	230.686
20000	0.2	457.376
40000	0.2	916.401
70000	0.2	1642.400
100000	0.2	2362.660

Table 1: Execution Times of the Simulation where $p = 0.2$

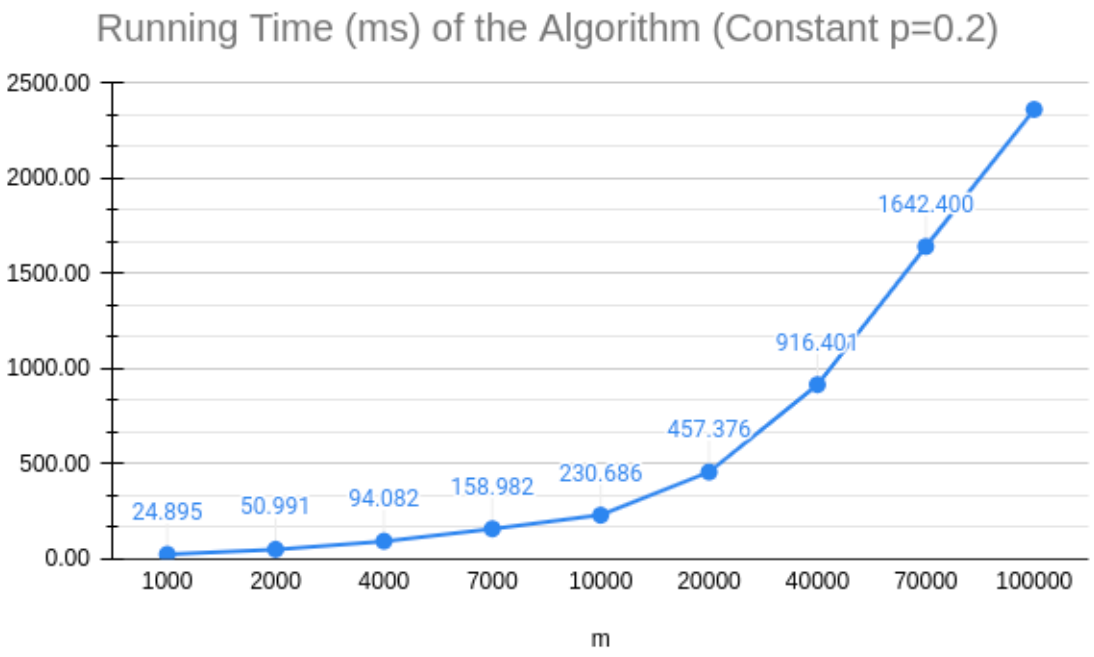

Figure 1: Aggregated Plot of Execution Time with Constant $p = 0.2$

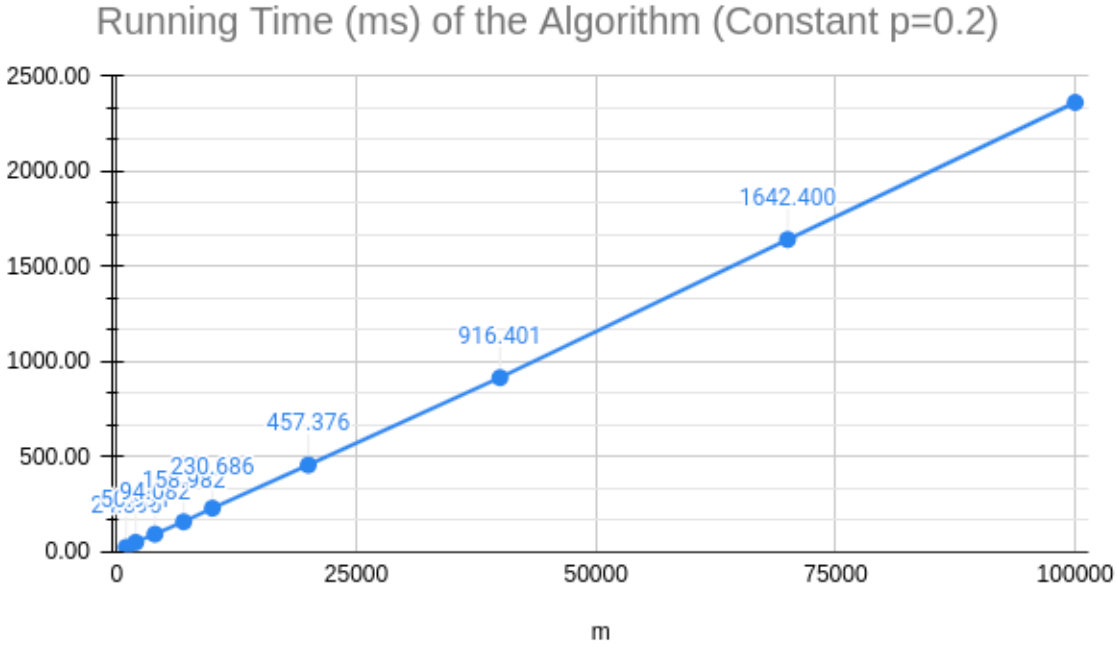

Figure 2: Plot of Execution Time with Constant $p = 0.2$

Question 3)

For values of $p = \{0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9\}$, execution times of the simulation is represented in Table 2 and in Figures 3-4 where $m = 10000$. As mentioned before, the first plot (Figure 3) is the aggregated version of the second plot (Figure 4) and helps with the better visualization of the data. According to the results given, running time of the simulation changes as p changes with a standard deviation of about 48 milliseconds.

Even though a decrease in running time is expected as p grows larger, it turned out not being the case for my implementation. A decrease in running time was being expected because as p gets bigger, less nodes will be added to the heap and n will be smaller relative to the previous run. Hence there will be less up-heapify operations needed.

However, since a new random index within the heap has to be generated in order to decrease the distance of a random taxi, my implementation samples this psuedo-random index from a uniform distribution that is generated by the Mersenne Twister engine. Although, this method of randomization is generally better and the preferred way of obtaining random numbers in C++, it is slower than many others such as `rand()`. Therefore there is an observable increase in the execution time of the simulation as p gets bigger. By using the code provided in the Appendix below, you can test the running times of different random number generating methods on your device. The given code produces integers between $[1, 100]$ for N number of times and prints the execution times of the methods. For $N = 100000$, output of an exemplary run on Ryzen 1600X CPU with clock speed of 4.00GHz on one core is given below. As it can quite clearly be observed, the Mersenne Twister engine is over 2500 times slower than C inherited `rand()`.

Output of the RNG Speed Test

```
rand(): 695 microseconds
mt: 1851799 microseconds
```

m	p	Time (ms)
10000	0.1	209.963
10000	0.2	228.713
10000	0.3	246.853
10000	0.4	262.700
10000	0.5	289.285
10000	0.6	298.305
10000	0.7	316.265
10000	0.8	330.011
10000	0.9	352.860

Table 2: Execution Times of the Simulation where $m = 10000$

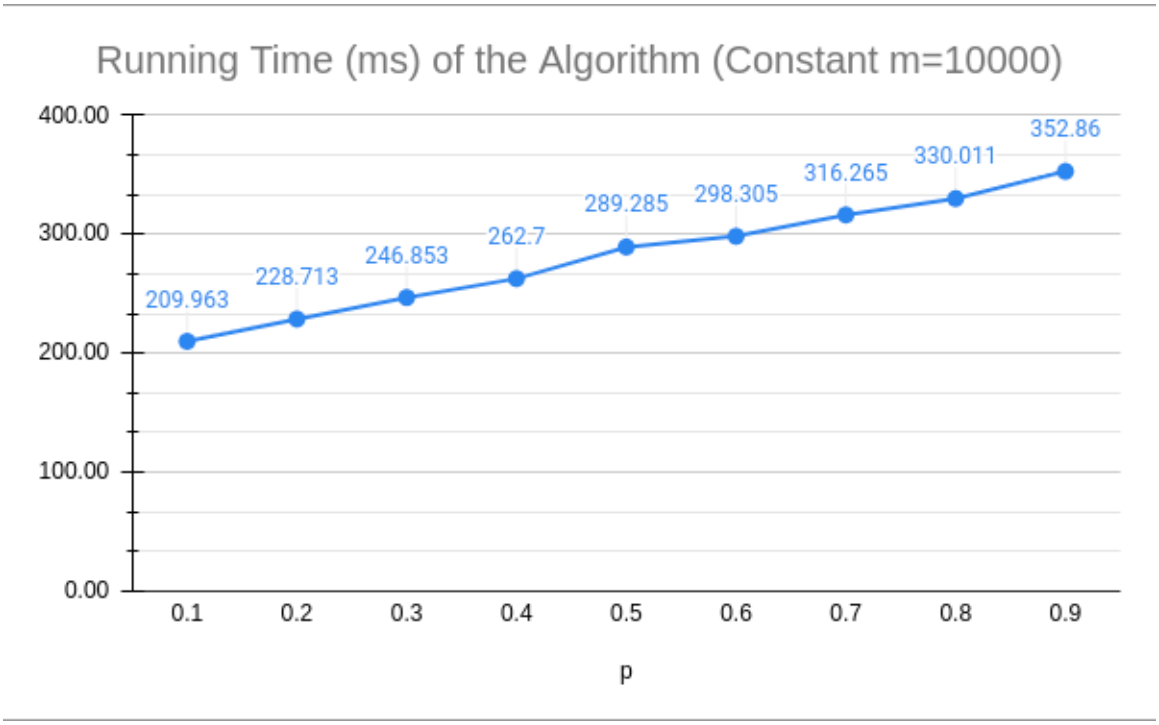

Figure 3: Aggregated Plot of Execution Time with Constant $m = 10000$

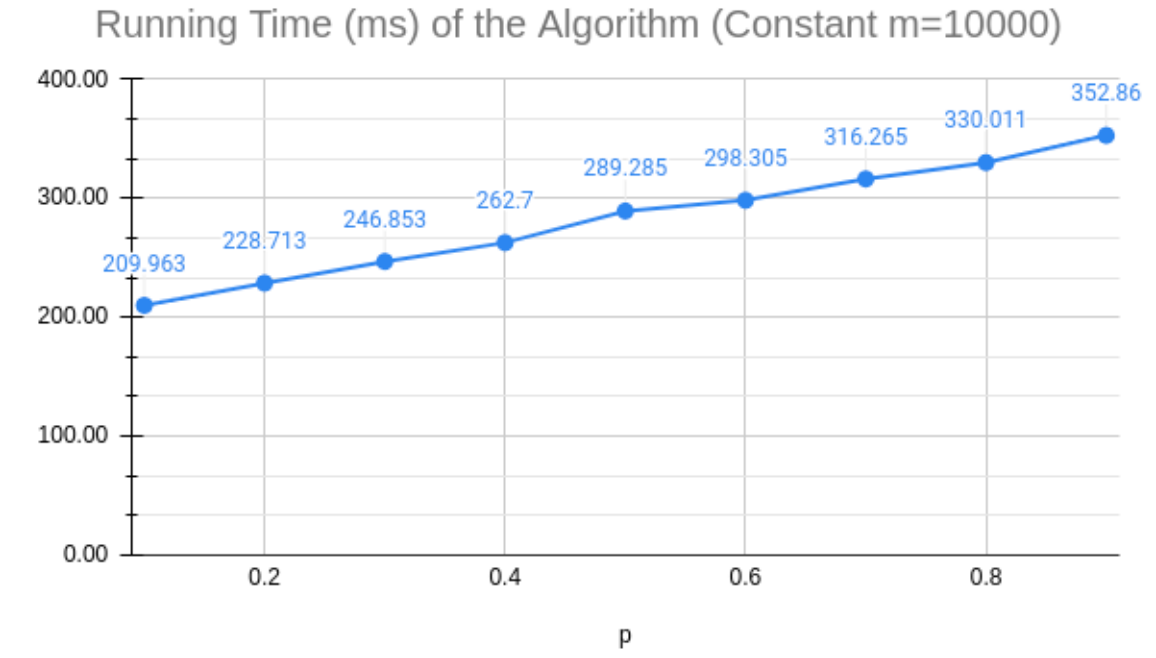

Figure 4: Plot of Execution Time with Constant $m = 10000$

Appendix

Random Number Generation Execution Speed Test

```
#include <iostream>
#include <chrono>
#include <random>
#include <stdlib.h>

using namespace std;

template <typename T>
T get_random_int(T start, T end) {
    random_device rand_device;
    mt19937 generator(rand_device());
    uniform_int_distribution<T> distribution(start, end);
    return distribution(generator);
}

int main(){
    int N = 100000;
    srand(time(nullptr));

    auto start_time = chrono::high_resolution_clock::now();
    for (int i = 0; i < N; ++i) {
        rand() % 100 + 1;
    }
    auto stop_time = chrono::high_resolution_clock::now();
    auto duration = chrono::duration_cast<chrono::microseconds>(↩
       stop_time - start_time);
    cout << "rand(): " << duration.count() << " microseconds" << endl;

    start_time = chrono::high_resolution_clock::now();
    for (int i = 0; i < N; ++i) {
        get_random_int(0, 100);
    }
    stop_time = chrono::high_resolution_clock::now();
    duration = chrono::duration_cast<chrono::microseconds>(stop_time -↩
        start_time);
    cout << "mt: " << duration.count() << " microseconds" << endl;

    return 0;
}
```