Beyza Ozan
ID: 150180056

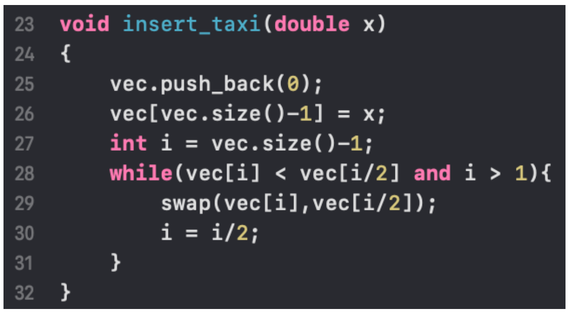
BLG 335E – Analysis of Algorithms I

Homework 2

Part 2 - Report

1) Explanation of the implementation:

First, let's consider "New distance can be added to the heap" in the function **insert_taxi**.

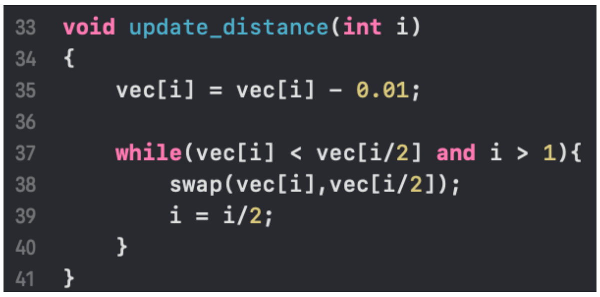


```
23  void insert_taxi(double x)
24  {
25      vec.push_back(0);
26      vec[vec.size()-1] = x;
27      int i = vec.size()-1;
28      while(vec[i] < vec[i/2] and i > 1){
29          swap(vec[i],vec[i/2]);
30          i = i/2;
31      }
32  }
```

Line 25, 26, 27 is a O(1). The while loop execute as many times as node i has ancestors, which is the depth of the tree. The depth of a binary tree is O(lg *n*).
In the loop, the work done is constant. Cost is O(1) + O(1) + O(lg n) = O(2) + O(3) + O(4 lg *n*) -> so running time is just O(lg *n*).

- The vector size is increased by one to add new taxi's location to the heap (I didn't use index 0 of vector). (line 25)
- The new location is assigned to the last index of the vector. (line 26)
- Then we compare child(new added location) with its parent(upwards). If child smaller than parent we swap them.
- We continue this process until the child is not smaller than its parent. (line 28-32)

Let' s consider "Some taxis' distances can be updated " in the function **update_distance**.



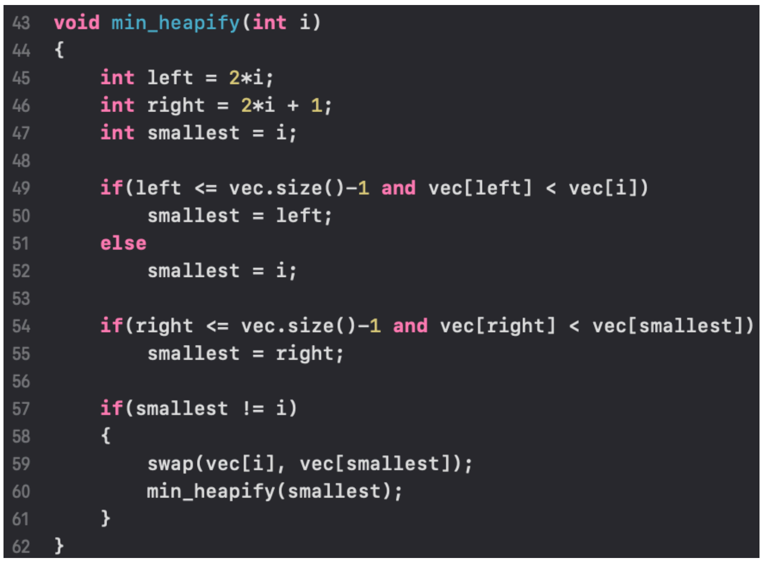
```
33  void update_distance(int i)
34  {
35      vec[i] = vec[i] - 0.01;
36
37      while(vec[i] < vec[i/2] and i > 1){
38          swap(vec[i],vec[i/2]);
39          i = i/2;
40      }
41  }
```

The while loop execute as many times as node i has ancestors. Constant amount of work is done in the loop (just like insert function). So the running time is O(3) + O(4 lg *n*) -> just O(lg *n*)

- In the homework, it was asked to decrease by 0.01 as an update operation.
- 'i' allows to find the desired location to be updated in the heap.
- First distance to the hotel is reduced by 0.01. (line 35)
- Then, due to the possibility of the order changing, the comparison is made again as in the insert_taxi. (Check the value of child and parent. If child is smaller than parent, swap) (line 37-41)

Let’s evaluate “The hotel desk clerk may call the nearest current taxi ” in the function **min_heapify** and **call_nearest_taxi**.

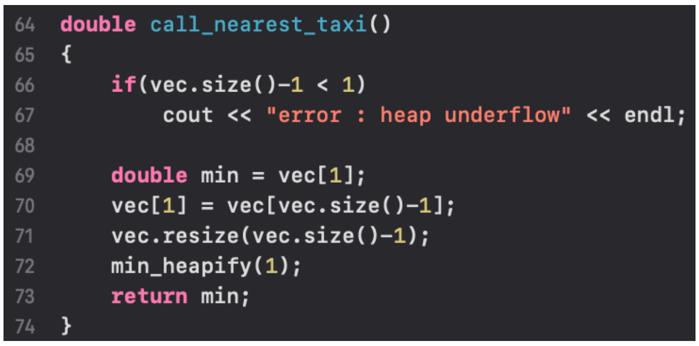
This function (shown below) rearranges a heap to maintain the heap property, that is, the key of the root node can be greater than the keys of its children.



```
43  void min_heapify(int i)
44  {
45      int left = 2*i;
46      int right = 2*i + 1;
47      int smallest = i;
48
49      if(left <= vec.size()-1 and vec[left] < vec[i])
50          smallest = left;
51      else
52          smallest = i;
53
54      if(right <= vec.size()-1 and vec[right] < vec[smallest])
55          smallest = right;
56
57      if(smallest != i)
58      {
59          swap(vec[i], vec[smallest]);
60          min_heapify(smallest);
61      }
62  }
```

- If the left child is smaller than its parent, assign the left to the smallest as a new current smallest one. (line 49,50)
- Otherwise the smallest remains as parent. (line 52)
- If the right is child smaller than its parent, assign right to the smallest as a new current smallest one. (line 54,55)
- If smallest value is changed, we should swap keys of index smallest and index i. (line 59)
- After that we do the same process again and again until i reach the leaf (recursive) (line 60)

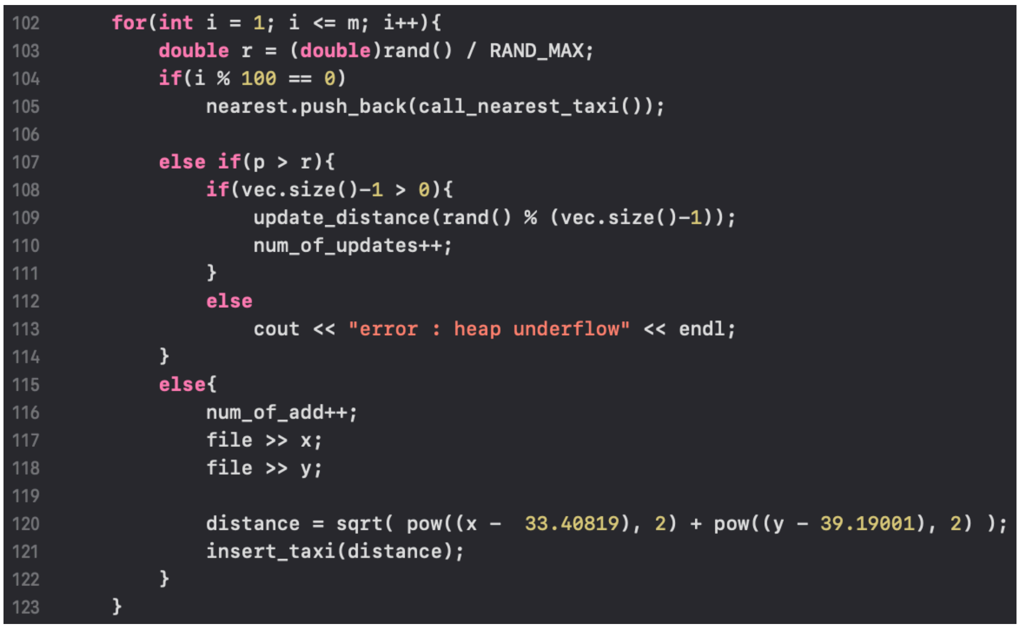
Steps in lines 45-59 do not depend on n and there is no loops. Therefore, the running time so far is Θ(1) .
But line 60, When we call min_heapify again, we already know that lines 45-59 cost Θ(1) steps.
But we may need to call min_heapify on a subtree rooted at one of the children of the current node, so we have to add the cost of doing that. Worst case is when the last row of the tree is half full on the left side and A[i] is their ancestor. The subtrees of the children of our current node have size at most $2n/3$. The running time of min_heapify can be described by the recurrence;
$T(n) \leq T(2n/3) + \Theta(1)$ —> This is Case 2 by the master method, so $T(n) = O(\lg n)$



```
64  double call_nearest_taxi()
65  {
66      if(vec.size()-1 < 1)
67          cout << "error : heap underflow" << endl;
68
69      double min = vec[1];
70      vec[1] = vec[vec.size()-1];
71      vec.resize(vec.size()-1);
72      min_heapify(1);
73      return min;
74  }
```

- If size of vector is smaller than 1 (there is no element in the heap yet), print error message and return the function. (line 66)
- First index of vector has a minimum distance value in the heap. Therefore we assign *vec[1]* to the variable *min* to keep min value in the hand. (line 69)
- Then we assign last added distance to the first index position. (line 70)
- Vector size is decreased by one. (line 71)
- Because the order of the distances in the heap is shuffled, the heap must be rearranged by sending index 1 to the function **min_heapify.** (line 72)
- Finally, retained *min* value is returned from the function. (73)

Lines 65-71 running time is O(1). There is a no loop and cost of min_heapify is O(lgn). When we just sum up the times, the running time is O(6) + O(lg n) -> so just O(lg n).

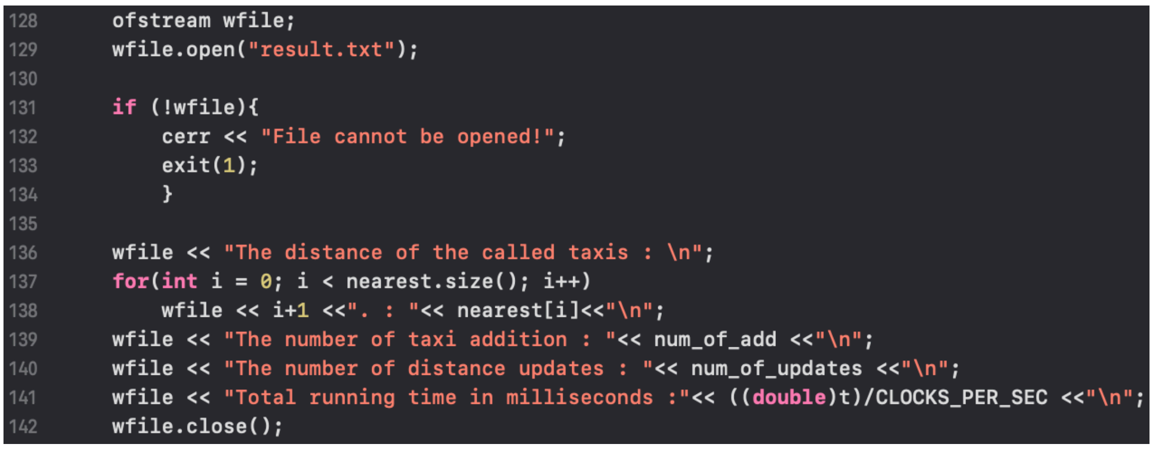


```
102     for(int i = 1; i <= m; i++){
103         double r = (double)rand() / RAND_MAX;
104         if(i % 100 == 0)
105             nearest.push_back(call_nearest_taxi());
106 
107         else if(p > r){
108             if(vec.size()-1 > 0){
109                 update_distance(rand() % (vec.size()-1));
110                 num_of_updates++;
111             }
112             else
113                 cout << "error : heap underflow" << endl;
114         }
115         else{
116             num_of_add++;
117             file >> x;
118             file >> y;
119 
120             distance = sqrt( pow((x -  33.40819), 2) + pow((y - 39.19001), 2) );
121             insert_taxi(distance);
122         }
123     }
```

The for loop runs as many as the number of operations (m) which is entered as an input.

In this loop, the operation to be done is decided according to the probability of p entered as an input.

- When we reached 100th operation or its multiples, we call the nearest taxi. (we push it to nearest vector for listing them later) (line104-105)
- If ‘i’ is not a multiple of 100 and entered ‘p’ is greater than the random value ‘r’, then we pick a random distance from the heap and update it. Also number of updates (num_of_updates) should be increase by 1. In addition, if vector size smaller than 1 (there is no distance in the heap), the error message is written. (line 107-114)
- If ‘i’ is not a multiple of 100 and entered ‘p’ is smaller or equal to the random value ‘r’, then new latitude and longitude values of the new taxi are read from the file (line 117-118). The distance between hotel and new taxi is calculated and inserted to the heap with inserted_taxi() function.   (line 120-121).



```
128     ofstream wfile;
129     wfile.open("result.txt");
130 
131     if (!wfile){
132         cerr << "File cannot be opened!";
133         exit(1);
134         }
135 
136     wfile << "The distance of the called taxis : \n";
137     for(int i = 0; i < nearest.size(); i++)
138         wfile << i+1 <<". : "<< nearest[i]<<"\n";
139     wfile << "The number of taxi addition : "<< num_of_add <<"\n";
140     wfile << "The number of distance updates : "<< num_of_updates <<"\n";
141     wfile << "Total running time in milliseconds :"<< ((double)t)/CLOCKS_PER_SEC <<"\n";
142     wfile.close();
```

At the end of the code (in the main function), the requested information is written to the file.

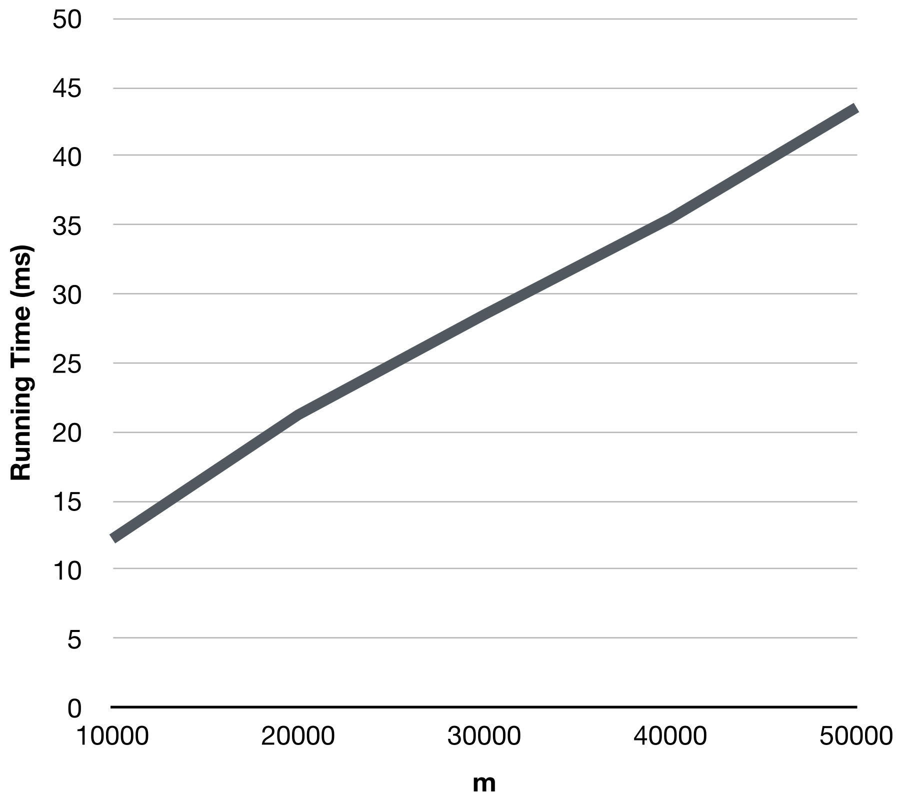
- First, the distances of the called taxis are listed line by line, first in first out respectively. (line 137-138)
- Then, the number of taxi addition is written. (line 139)
- Then, the number of updates is written. (line 140)
- Finally, the total running time in milliseconds is written to the file. (line 141)

All operations (insert, update, call) can be done on a set of size *n* in O(lg *n*) time. When we put these operations in for loop (line 102), O(lgn) run m times. It means that the running time is

O(mlg *n*).

2) Demonstrate using a table of the effect of the m choice on the running time (p constant) :

m (p = 0.2)	**Running Time** (milliseconds)
1.000	**1.274**
2.000	**3.001**
5.000	**7.313**
10.000	**11.818**
20.000	**21.518**
50.000	**45.132**
100.000	**78.685**



It is clear that as the number of m increases, the running time also increases.

In the first question, we explained that the theoretical running time is O(nlgn). Let's look at the ratios of our measured results in the tables,

For m = 1000, the theoretical running time = 1000.lg1000 ≅ 10965 measured = 1.274

For m = 2000, the theoretical running time = 2000.lg2000 ≅ 21930 measured = 3.001

Rate of theoretical = 21930 / 10965 = 2

Rate of measured = 3001 / 1274 ≅ 2,35

For m = 10.000, the theoretical running time = 10000.lg10000 ≅ 132870 measured = 11.818

For m = 20.000, the theoretical running time = 20000.lg20000 ≅ 285740 measured = 21.518

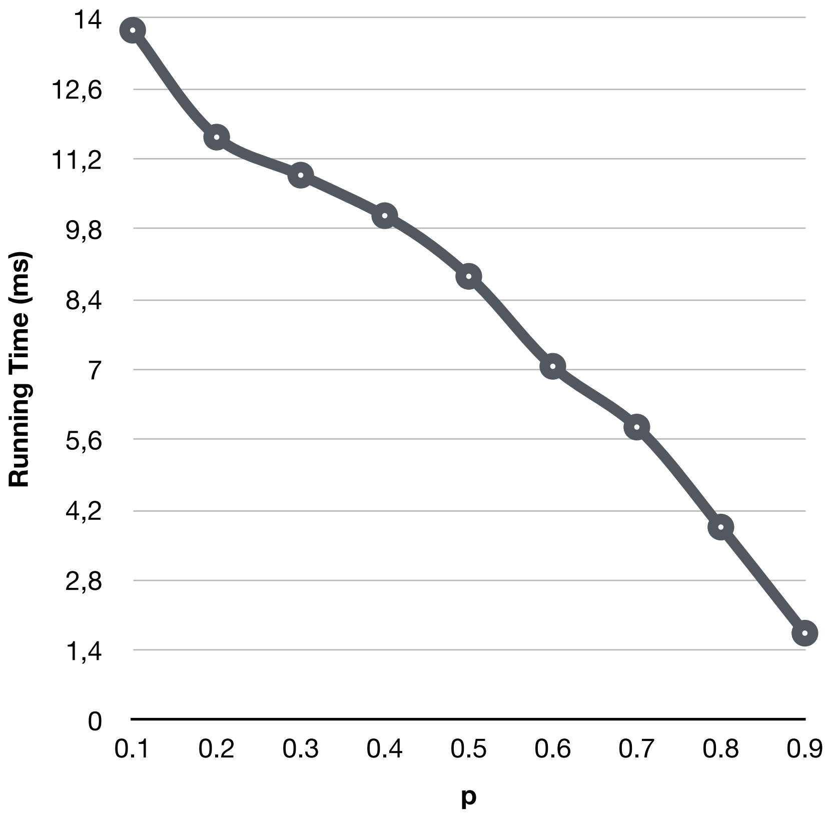
Rate of theoretical = 285740 / 132870 = 2,150

Rate of measured = 21518 / 11818 ≅ 1,82

The table as we expected. The running times in the tables are close to the theoretical results

3) Demonstrate using a table the effect of the p choice on the running time (m constant) :

P (m = 10000)	**Running Time** (milliseconds)
0.1	**13.767**
0.2	**11.634**
0.3	**10.875**
0.4	**10.067**
0.5	**8.857**
0.6	**7.062**
0.7	**5.850**
0.8	**3.855**
0.9	**1.739**



It is obviously seen that, change in p **affects** the running time significantly. As **p value increases** (p approaches 1), the **running time decreases** according to the table and graph.
p is the probability of the update operation and 1 - p is the probability of the insert new distance operation. When p approaches from 0 to 1 (as the running time gets shorter), the number of updates increases and at the same time the number of inserts is decreases. Therefore, it can be said that insert operation is slower than update operation.
Insert operation includes the update operation (when we insert any distance in the heap we should update its position), which can be shown as a reason why insert is slower. Also, unlike update, insert operation reads values from file, calculates distance, expands vector size.