BLG 335E – Analysis of Algorithms I
Homework 2

Priority Queue using Binary Heap

150170092 : Barış İncesu

25 DECEMBER 2020

Part 2. Report

2.1

The first function of the system is to create a vector structure to hold values for prioritization. It would be wise to keep the size variable as well as it will use index operations frequently.

```
vector<double> taxi;
int size = -1;
```

After the structure to keep the data is determined, the function **add()** appears. This function first pushes the taxi distances from the input file to the end of the vector with push_back(). After that, since the binary heap model is chosen, the new value should be placed in the appropriate place. shiftUp() is doing this insertion.

```
void add(double p)
{
    size = size + 1;
    taxi.push_back(p);

    shiftUp(size);
}
```

When imagining a virtual tree for the **Binary Heap model**, the data is considered to have a parent, leftchild and rightchild according to their index. According to the created heap model, there are the following relationships with the item with **i** index, respectively.

```
parent = (i - 1) / 2
leftChild = (2 * i) + 1
rightChild = (2 * i) + 2
```

Because the closest taxi is wanted in the scenario, the taxi distance at the **top of the binary heap should be the lowest**. This situation requires a replacement **if the children are less than parents**. however, the one-off operation of this operation does not make sense. This operation must be done through the entire vector to get the entire tree sorted correctly.

Thus, thanks to **shiftUp()**, the newly added distance reaches the correct index.

```
void shiftUp(int i)
{
    while (i > 0 && taxi[parent(i)] > taxi[i])
    {

        // Swap parent and current node
        swap(taxi[parent(i)], taxi[i]);

        // Update i to parent of i
        i = parent(i);
    }
}
```

The second desired situation in the scenario, **update()**, requires that a value currently in the vector be **reduced by 0.01**. The striking part here will be that the vector stays ordered again after the value changes. Since the updated value here will always be smaller, it is necessary to send this value to the correct place above.

```
void update(int i)
{

    taxi[i] = taxi[i] - 0.01;
    shiftUp(i);
}
```

The third operation is **call()**. The first element of the vector is to detach the top element of the binary heap from the vector. **pop_back()** can be used for breaking vectors. If **pop_back()** is done after swapping the first item with the last item, the initial item will be detached from the vector. However, since the last element comes first, the order is broken. For this, it is necessary to place this element in the right place, downwards. A **shiftDown()** is designed for this process, just like shiftUp().

```
double call()
{
    double called = taxi[0];

    // Replace the value at the root with the last leaf
    swap(taxi[0], taxi[size]);
    taxi.pop_back();
    size = size - 1;

    // Shift down the replaced element to maintain the heap property
    shiftDown(0);

    return called;
}
```

This helper operation examines situations where children are smaller than the parent. It is compared with the children by assigning the parent index to a current. **If it is lower**, it replaces the parent value with current. Then, if the principle selected value and the last selected value are not the same, that is, if a smaller value becomes current, a swap operation is performed. This process is done through the entire vector so that when the large value in the **call()** function reaches the top, it can come back to the correct index.

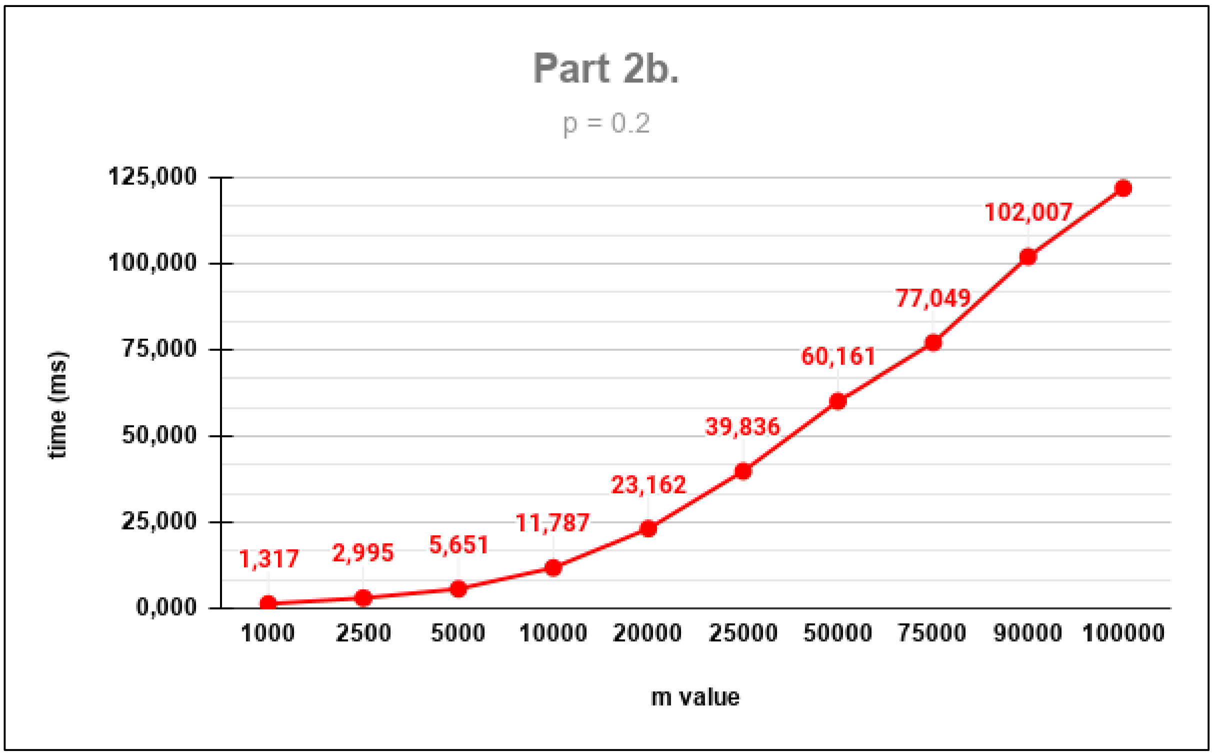
```
void shiftDown(int i)
{
    int current = i;

    // Left Child
    int l = leftChild(i);

    if (l <= size && taxi[l] < taxi[current])
    {
        current = l;
    }

    // Right Child
    int r = rightChild(i);

    if (r <= size && taxi[r] < taxi[current])
    {
        current = r;
    }

    // If i not same as current
    if (i != current)
    {
        swap(taxi[i], taxi[current]);
        shiftDown(current);
    }
}
```

After all these operations were completed, the appropriate codes for the desired simulation were written into the **main()** function. If we examine the priority queue and binary heap here, we can see that we do **shiftUp()** recursively at each add operation. Just like other sorting algorithms, when we consider it as sorting only, each added distance to the structure takes ***logn***, and when ***n*** distances are added, theoretical operation is expected as Big O Notation as **Θ*(*nlogn*)***.

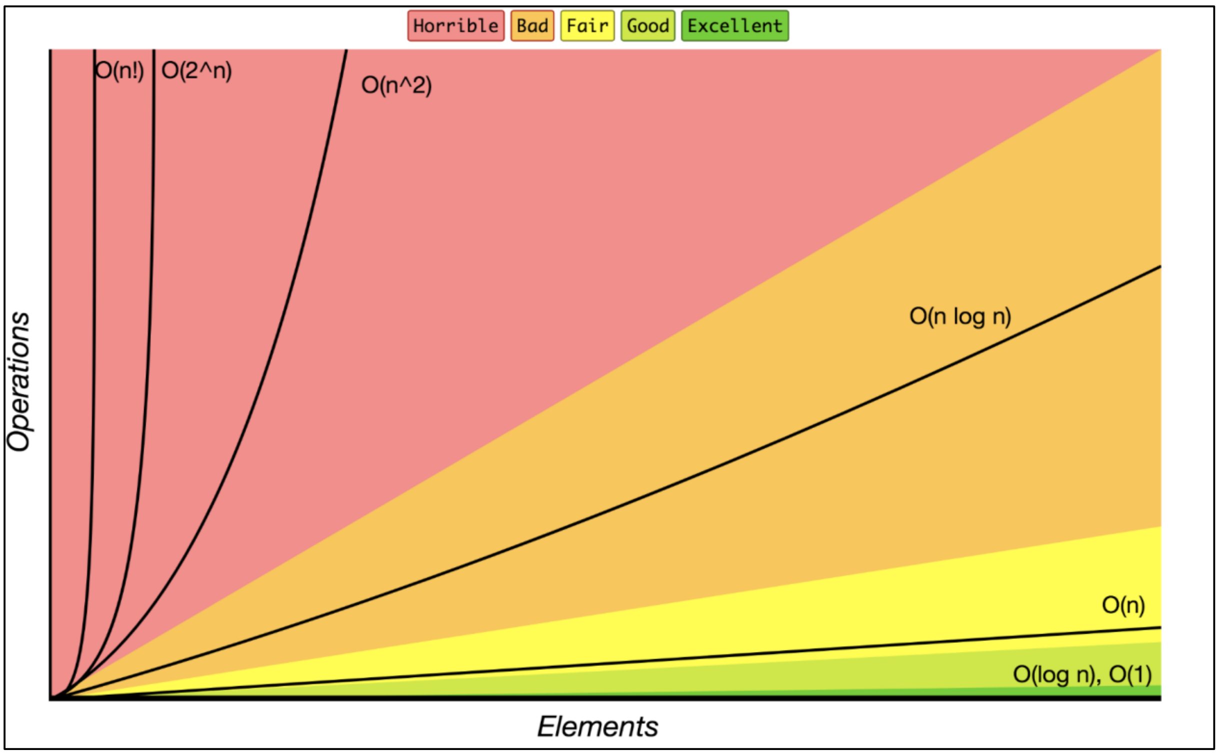
2.2

Table 1: the effect of the m choice on the running time

m	p	time (ms)
1000	**0.2**	1,317
2500	**0.2**	2,995
5000	**0.2**	5,651
10000	**0.2**	11,787
20000	**0.2**	23,162
25000	**0.2**	39,836
50000	**0.2**	60,161
75000	**0.2**	77,049
90000	**0.2**	102,007
100000	**0.2**	121,920



In this part, the experiments were carried out by keeping the **p** number constant and changing the **m** number. Since the number of p is constant, it can be said that even if it has a random function, the number of m directly changes the number of nodes to be added to the vector. This situation can be thought of as a sort of sorting in the last paragraph of the explanation above. In this case, a graph similar to the ***nlog*** graph according to the expected Big O Notation should appear.

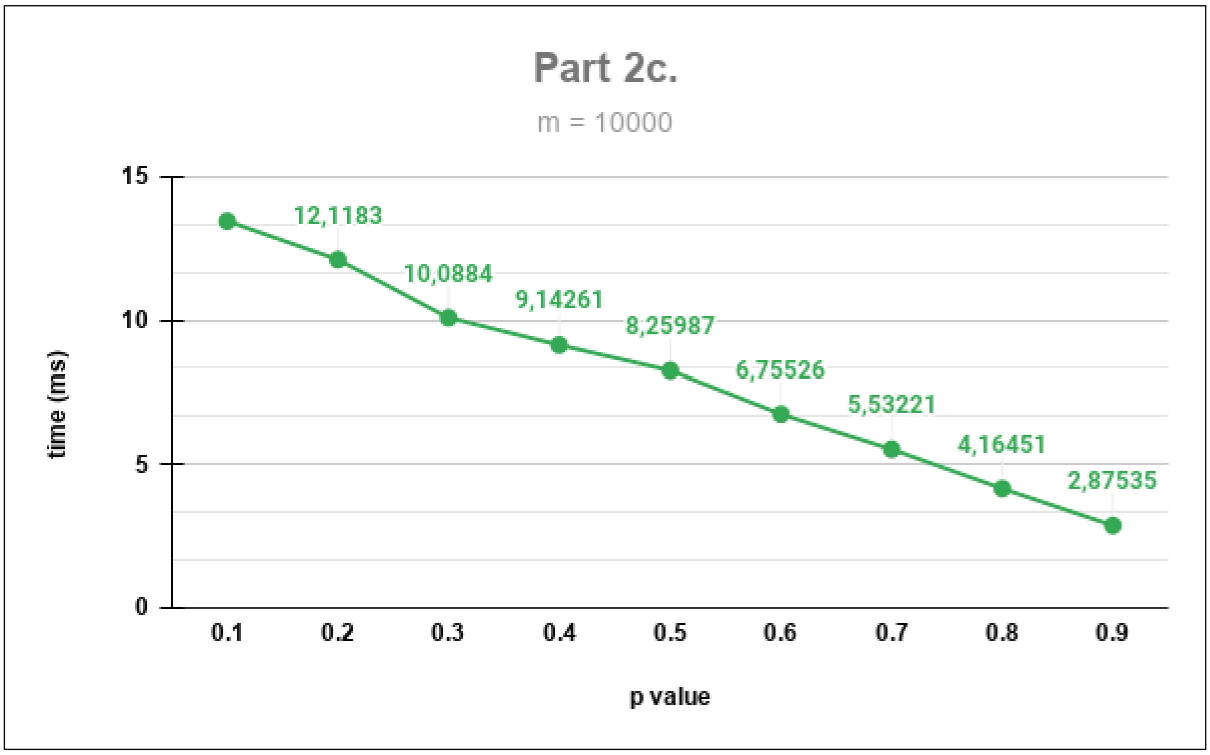


When compared with the graph above, it can be said that the result is as expected.

2.3

Table 2: the effect of the p choice on the running time

m	p	time (ms)
10000	0.1	13,4571
10000	0.2	12,1183
10000	0.3	10,0884
10000	0.4	9,14261
10000	0.5	8,25987
10000	0.6	6,75526
10000	0.7	5,53221
10000	0.8	4,16451
10000	0.9	2,87535



In these trials, the number of p represents the update probability in the simulation. Accordingly, increasing the **p ratio** in **m operation decreases the entry of new elements into the vector**. This situation has also been observed in trials. In case of **0.1**, **9000** add() is made in 10000 operations, while in **0.9** this number is only **1000**. Since the number of add() operations is directly proportional to the running time of the program, the time decreases as the p ratio increases. This situation is similar to the previous step, if the duration of the add() operation is ***logn***, the time will be ***nlogn*** here, but the time will gradually decrease.