~ Algorithms Homework 2 Repeat ~

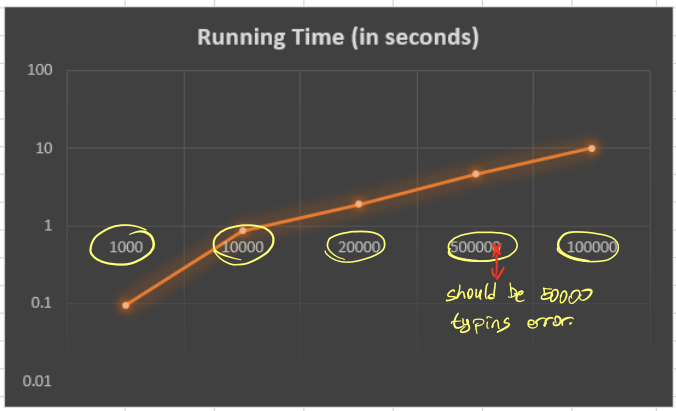
```
class minPriorityQueue : public MinHeap{
    public:
    Vehicle extract_min();
    void decrease_key(int pos, double key);
    void insertCar(Vehicle car);
};
Vehicle minPriorityQueue::extract_min(){
        if(heap_size < 1){ // if there is no car in the heap do not execute this operation
            cerr << "Heap underflow. Aborted... ";   O(1)
            exit(-1);
        }                                            O(1)
        Vehicle min = vehicles[0]; // get the root
        vehicles[0] = vehicles[vehicles.size()-1]; // put the last into first place   O(1)
        vehicles.pop_back(); // delete element   O(1)
        heap_size = vehicles.size(); // update size   O(1)
        minHeapify(0);   O(logn)
        return min; // return root, which is the quickest   O(1)
}
void minPriorityQueue::decrease_key(int pos, double key = -1){// Key is default arg and I assign -1, because the division is always > 0 logically
    pos--; // 0 index   O(1)
    if(pos>vehicles.size()) // check whether the given position is in range   O(1)
        cerr << "Pos value is out of range" << endl; // if not warn
    if (key > vehicles[pos].key_val) // check whether the given key is smaller than the value of vehicles[pos]
        cerr << "New key cannot be bigger than current key";   O(1)
    else
        vehicles[pos].key_val = key;   O(1)
    // check whether you reach the root, otherwise compare the parent_key with current_key and swap if the parent is larger
    while(pos>0 && vehicles[parent(pos)].key_val > vehicles[pos].key_val){
        // Traverse the tree from bottom up. This will take O(logn) times for which n is equal to n is the number of nodes. Which is same as car number
        swap(vehicles[pos], vehicles[parent(pos)]);
        pos = parent(pos);                           O(logn) → traversing the binary tree
    }
}
void minPriorityQueue::insertCar(Vehicle car){
    vehicles.push_back(car); // push the car to vector   O(1)
    heap_size = vehicles.size(); //update heap_size   O(1)   → O(logn)
    decrease_key(heap_size, car.key_val); //use decrease key (as it is given in slides)
}
```

1

Above, I inserted corresponding functions which are located in main.cpp To compute extract-min function, I write corresponding times in Big O notation. As can be seen from the figure running time for extract is $O(1) \cdot 6 + O(\log n)$ Same notion(idea) can be used for insert_car and decrease_key functions. In those functions everything is $O(1)$ time except decrease-key function for insert car and while loop in decrease-key functions. Both while loop and decrease-key function again takes $O(\log n)$. The reason why running time is $O(\log n)$ can be explained with the fact that height of binary tree (heap tree) is also $\log n$. However in the home work we are not using the number of vehicles for n. We are using operation counts as input argument and therefore our results is based on operation counts which is linear time. $O(n)$. Let x be the cars called without lucky-num and y be the cars with lucky-num. Then $2x + 3y = n$. $O(n)$ time is proved.

(2) Below Excel table and graph is inserted.

Calculation Time	1K	10K	20K	50K	100K
	0.097	0.872	1.922	4.909	9.413
	0.089	0.857	1.634	4.493	9.988
	0.106	0.813	1.853	4.822	9.94
	0.094	0.825	1.927	4.363	10.146
	0.095	0.935	2.128	4.502	10.24
average	0.0962	0.8604	1.8928	4.6178	9.9454



As I can tell from the graph and the logic of the code, running time is linear which is validated in the first part. We are computing running time according to the number of operations and the number of operation $2x + 3y = n$