Assignment 10

Done By: Nayan Man Singh Pradhan

Problem 10.1 - Hash Tables

a)

Given,
$seq = \{3, 10, 2, 4\}$
$size\ (m) = 5$
$h_1(k) = k\ mod\ 5$
$h_2(k) = 7k\ mod\ 8$

We know, double hash uses the hash function:

$$h(k, i) = (h_1(k) + i\ .\ h_2(k))\ mod\ m$$

First, we insert 3:
Here, the value of $i = 0$ as it is the initial iteration for this key. Therefore,
$h(k, i) = (3\ mod\ 5 + 0)\ mod\ 5 = 3\ mod\ 5 = \mathbf{3}$
Now, we check position 3 in our hash table. Since position 3 is empty, we insert key 3 in position 3.

Then, we insert 10:
The value of $i = 0$ as it is the initial iteration for this key. Therefore,
$h(k, i) = (10\ mod\ 5 + 0)\ mod\ 5 = 0\ mod\ 5 = \mathbf{0}$
We check position 0 in our hash table. Since position 0 is empty, we insert key 10 in position 0.

Then, we insert 2:
The value of $i = 0$ as it is the initial iteration for this key. Therefore,
$h(k, i) = (2\ mod\ 5 + 0)\ mod\ 5 = 2\ mod\ 5 = \mathbf{2}$
We check position 2 in our hash table. Since position 2 is empty, we inert key 2 in position 2.

Finally, we insert 4:
The value of $i = 0$ as it is the initial iteration for this key. Therefore,
$h(k, i) = (4\ mod\ 5 + 0)\ mod\ 5 = 4\ mod\ 5 = \mathbf{4}$
Finally, we check position 4 in our hash table. Since position 4 is empty, we insert key 4 in position 4.

Therefore, there are no collisions while inserting the given data.

b)

Hash Table has been implemented in *hashTable.cpp* (execute make to run).

I selected linear probing with $h(k,i) = (h'(k) + i) \ mod \ m$, where $h'(k) = key \ mod \ maxSize$ as I am only inserting a small number of keys (5 keys) to test the program, and hence, my algorithm will not suffer from a large amount of primary clustering (if collision occurs).

Problem 10.2 – Greedy Algorithm

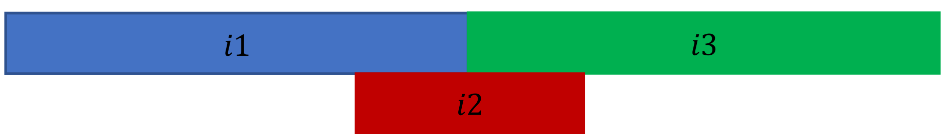
a)

The greedy algorithm in the activity-selection problem may fail at producing a globally optimal solution.

An example of this scenario is given below:

$\boldsymbol{i}$ **(index)**	**1**	**2**	**3**
$\boldsymbol{s_i}$**(start time)**	1	5	6
$\boldsymbol{f_i}$**(end time)**	6	7	12

A rough diagram to illustrate the table:



The greedy algorithm always chooses the solution that is locally optimal. In this case, the greedy algorithm chooses the index (i) with the shortest duration, i.e. $i2$. Even though $i2$ is the locally optimal solution, it is clearly not the globally optimal solution (as seen from diagram). In our example, the globally optimal solution is $\{i1, i3\}$.

Hence, we prove a contradiction, and therefore, prove that selecting the activity with shortest duration may fail at producing a globally optimal solution.

Therefore, proved!

b)

Algorithm implemented in c++ file "greedy.cpp" (execute "make" to run). I have added a screenshot of the part that returns the final greedy solution with explanations in comments below (from "greedy.cpp" in zip file).

```cpp
// returns final solution
ListOfActivity ListOfActivity::returnListSolution() {
    ListOfActivity Solution; // creating a list of activity called solution
    while (!activityList.empty()) { // while activity list is not empty
        Activity latestStartTime; // creating a temp activity
        // the start and finish time of latestStartTime is initialzied to 0
        latestStartTime.setStart(0);
        latestStartTime.setFinish(0);
        int pointer = 0; // just an index
        // finding the latest start time
        for (int i = 0; i < activityList.size(); i++) {
            if (activityList[i].getStart() > latestStartTime.getStart()) {
                latestStartTime.setStart(activityList[i].getStart());
                latestStartTime.setFinish(activityList[i].getFinish());
                pointer = i; // pointer used later
            }
        }

        // if the latest is the temp activity, stop
        if (latestStartTime.getStart() == 0 &&
                latestStartTime.getFinish() == 0) {
            break;
        }

        /*
            bool to check if the latestStartTime activity overlaps with other
            activities in the final solution
        */
        bool overlapCheck = false;
        for (int i = 0; i < Solution.size(); i++) { // goes through activities in sol
            if (latestStartTime.getFinish() > Solution.elemAt(i).getStart()) {
                overlapCheck = true; // if overlap occurs
            }
        }

        // if overlap does not occur, add activity to solution list
        if (overlapCheck == false) {
            Solution.addActivity(latestStartTime);
        }

        // now erase the latestStartTime activity from the list of activities
        activityList.erase(activityList.begin()+pointer);
    }

    /*
        The above returns the solution in reversed order.
        To print in proper order, I just copied the activities from back
        to front in a new ListOfActivity.
    */
    ListOfActivity InCorrectOrderSolution;
    for (int i = Solution.size()-1; i >= 0; i--) {
        InCorrectOrderSolution.addActivity(Solution.elemAt(i));
    }
    return InCorrectOrderSolution;
}
```