BLG 336E - Analysis of Algorithms II
2019/2020 Spring Final Project Question 4

- You should write all your code in C++ language.
- Your code should be able to be compiled with default g++ compiler and run under Ubuntu OS. **Even if you are writing your code on a different OS, you should check it via ITU SSH.**
- This is a Final Course Project Assignment, cheating is absolutely unethical and morally unacceptable. It will be punished by a negative grade. Also disciplinary actions will be taken.
- Program must be run with given command line arguments. **The codes not using these arguments or giving output in a different layout will not be graded**.

1 Optimal Caching (25 pts)

When a computer request some elements from the memory during a process, it uses a cache to store a small amount of the data in a fast memory rather than access the main memory each time. So that frequently used elements are kept in cache memory in order to process faster.

When an element is requested, processor first looks into cache to see if this element exists there. If it is, then we call this as **cache hit** and we can access the element directly. However if the element does not exist in cache then it is called as **cache miss** and the related element is moved from main memory to the cache then we can access it. However, if the capacity of the cache is full when a cache miss occurs, then we need to evict one of the elements in the cache in order to add a new one. Please note that cache miss results in a cost since memory operations are needed.

Typically, caching is an on-line problem. That is, we have to make decisions about which data to keep in the cache without knowing the future requests. Here, however, we consider the off-line version of this problem, in which we are given in advance the entire sequence of upcoming requests. We can solve this off-line problem by a greedy strategy called **furthest-in-future**, which chooses to evict the item in the cache whose next access in the request sequence comes furthest in the future.

For example let's say we have [0, 1, 2, 3, 4] as elements, [0, 1, 2, 3] in cache and requests coming as [4, 0, 1, 2, 3 . . .]. When 4 is requested there will be a cache miss, then 3 will be evicted from the cache since its next access comes furthest in the future.

Implementation [10 pts]

Please implement the given pseudocode in C++ using the given template **q4.cpp** file. The code must be compiled and run with following commands:

```
g++ q4.cpp -o q4
./q4 test.txt
```

We will have n requests, k cache capacity and k+1 different elements.
Sample input as $k = 4$ and $n = 15$ in **test.txt**:

- **Elements = [0, 1, 2, 3, 4]** : Possible elements that can be requested. You can assume that elements are consecutive integers from [0, 1, . . . , k] in all test cases.
- **Cache = [0, 0, 0, 0, 0]**: Binary array where includes 1 if corresponding element is in cache, else 0. So cache = [0, 1, 1, 1, 1] includes 1, 2, 3, 4 in it and it is full since the capacity is 4. This type of implementation provides O(1) complexity when searching an element in the cache. Initially, the cache will be empty.
- **Requests = [0, 1, 2, 3, 4, 0, 1, 2, 3, 4, 0, 1, 2, 3, 4]**: Upcoming requests

```
Initialize elements, cache and requests array;
Create an extra double linked list for elements array;
Create address array to keep the addresses of the nodes in the list;
Create fif array to keep the farthest-in-future element for each request;
for i = n  0 do
    node = address[requests[i]];
    Move node to beginning of the linked list;
    fif[i] = the last element of linked list;
end
for i = 0  n do
    if requests[i] is not in the cache then
        cache miss;
        if cache is full then
            evict the fif[i] element from the cache;
            add the requests[i] to the cache;
        else
            add requests[i] to the cache;
            increment the number of elements in cache;
        end
    else
        cache hit;
    end
end
```

Algorithm 1: Farthest-in-future algorithm

The output should be the same as Fig. 1. Please be careful about the syntax.

```
cache miss
element 0 is added into the cache
cache miss
element 1 is added into the cache
cache miss
element 2 is added into the cache
cache miss
element 3 is added into the cache
cache miss
cache is full, element 3 is evicted
element 4 is added into the cache
cache hit
cache hit
cache hit
cache miss
cache is full, element 2 is evicted
element 3 is added into the cache
cache hit
cache hit
cache hit
cache miss
cache is full, element 1 is evicted
element 2 is added into the cache
cache hit
cache hit
```

Figure 1: Expected output

Report [15 pts]

Please answer the following questions in **2-3 sentences for each** considering the given problem. Submit your solutions in a .pdf file.

1. What is the subproblem of this greedy algorithm and what are we trying to optimize?
2. What is the time complexity of the given implementation in terms of n (number of requests) and m (number of elements)? Explain briefly.
3. The given algorithm in the question only works for the cases where the capacity of cache is k and the number of elements is k+1.
   a) Please try to show it with a counter example where the cache capacity is k and number of elements is k+2.
   b) Please briefly mention what kind of improvement would solve this problem.