CS345 : Algorithms II
Semester I, 2017-18, CSE, IIT Kanpur

Assignment 5

Deadline : 6:00 PM on 15th November, 2017

Important Guidelines:

- It is only through the assignments that one learns the most about the algorithms and data structures. You are advised to refrain from searching for a solution on the net or from a notebook or from other fellow students. **Before cheating the instructor, you are cheating yourself**. The onus of learning from a course lies first on you and then on the quality of teaching of the instructor. So act wisely while working on this assignment.
- There are three exercises in this assignments. Each exercise has two problems- one *easy* and one *difficult*. Submit exactly one problem per exercise. It will be better if a student submits a correct solution of an easy problem that he/she arrived on his/her own instead of a solution of the difficult problem obtained by hints and help from a friend. Do not try to be so greedy :-).
- The answer of each question must be formal, complete, and to the point. Do not waste your time writing intuition or idea. There will be penalty if you provide any such unnecessary details.

1 Array lover

The aim of the following two problems is to make you realize that the concept of amortized analysis can be used to design efficient data structures and algorithms as well. In particular, we may get rid of height balanced binary search trees using a collection of arrays provided we can tolerate slight increase in query and/or update time. Isn't it amazing ?

You have to submit the solution of only one of the following two problems.

1.1 Binary search under insertions

(*marks=20*)

Binary search of a sorted array takes logarithmic search time, but the time to insert a new element is linear in the size of the array. We can improve the time for insertion by keeping several sorted arrays. Design a data-structure which is a collection of arrays only that can support any sequence of n insertions in $O(n \log n)$ time. The worst case time for the search is $O(\log^2 n)$.

Hint: Get *inspiration* from a binary counter, especially the way the bits are flipped during an increment operation. Although you are strongly advised to solve this problem on your own, a more explicit hint is given at the end of this assignment in case you need it desperately.

1.2 Binary search and predecessor/successor queries under deletions

(*marks=40*)

There is a set S storing n elements. We need an efficient data structure that can support the following operations efficiently.

- Search(x, S): search for element x in the present set S.
- Predecessor(x, S): report the largest elements in S which is smaller than x.
- Delete(x, S): Delete element x from S.

It is trivial to use a height balanced binary search tree to solve this problem. But as you might have realized that it is too complex to implement compared to keeping just an array. So your data structure **must** use only a couple of arrays. The space must be close to $O(|S|)$ at every stage. The time to answer any Search query or Predecessor query must be worst case $O(\log n)$ and time to perform a Deletion operation has to be amortized $O(\log n)$.

Advice: Don't waste your time searching for solution of this problem on the net. Firstly you are unlikely to find it. Secondly, and more importantly, even if you find it, you are depriving yourself from the joy of discovering a very elegant solution of this last gem problem on your own. Choice is totally yours.

2 Amortized analysis

Provide solution of exactly one of the following two problems.

2.1 Analysis of lazy Binomial Heap

(*marks = 15*)

Recall from the implementation of Binomial Heap where we take a lazy approach to implement each operation. Provide complete details of Extract_min operation in this implementation and prove that its amortized time complexity is $O(\log n)$.

2.2 Dynamic Reachability

(*marks = 30*)

Let G be a directed graph on n vertices with no edges in the beginning. Let s be a designated source vertex. We receive a sequence of m edge insertions. Our aim is to maintain a Boolean array R with the following property after each edge insertion.

$R[j]$ = true if and only if there is a path from s to j consisting of edges present in the graph.

In the beginning $R[s]$ =true, and $R[i]$ =false for each $i \neq s$. Upon insertion of an edge (i, j), we invoke Procedure Update-R(i, j) to update R. This is a recursive procedure sketched below. Fill in the blanks appropriately.

```
if ....................and................ then
    ..................← true;
    foreach neighbor q of ................. do
        ...................................;
    end
end
```

Procedure `Update-R(`i, j`)`

Though the worst case time of handling an edge insertion may be quite large, the total time for processing m insertions is $O(m)$. To show this, do the following tasks.

1. State a potential function such that the amortized cost of handling an edge insertion is $O(1)$.
2. Express the actual cost of handling an edge insertion formally and precisely.
3. Show using the potential function defined above that the amortized cost of handling an edge insertion is indeed $O(1)$.

3 Fibonacci heap

3.1 degree versus size of a tree

(*marks* = 15)

Prove rigorously that the degree of any tree in a Fibonacci Heap of size n is $O(\log n)$.

3.2 Tinkering with the marked nodes

(*marks* = 30)

In the Fibonacci heap discussed in the class, as soon as a marked node v loses its second child, the subtree rooted at v is cut from its parent and added to the root list. What if the subtree rooted at a marked node v is cut from its parent and added to the root list only when it loses its 3rd child ? Will all the bounds of Fibonacci heap still hold ?

Explicit Hint for 1(a) problem:

Suppose that we wish to support **search** and **insert** on a set of n elements. Let $k = \lceil \log(n+1) \rceil$, and let the binary representation of n be $\langle n_{k-1}, n_{k-2}, ..., n_0 \rangle$. We have k sorted arrays $A_0, A_1, ..., A_{k-1}$, where for $i = 0, 1, ..., k-1$, the length of A_i is 2^i. Each array is either full or empty, depending upon whether $n_i = 1$ or $n_i = 0$, respectively. The total number of elements held in all k arrays is, therefore, $\sum_{i=0}^{k-1} 2^i = n$. Although each individual array is sorted, elements in different arrays bear no relationship to each others. Each insertion will involve removing some arrays and creating another array. Use the idea from merge sort to do this task efficiently.