**COL 702**

# Homework III

*Due on Oct 21*

Justify your answers with proper reasonings/proofs.

1. A multi-stack consists of an infinite series of stacks $S_0, S_1, \ldots$ where the $i$th stack $S_i$ can hold upto $3^i$ elements. The user always pushes and pops elements from the smallest stack $S_0$. However, before any element can be pushed onto any full stack $S_i$, we first pop all the elements of $S_i$ and push them into $S_{i+1}$ (and if $S_{i+1}$ gets full, we need to recurse). Similarly, before any element can be popped from any empty stack $S_i$, we first pop $3^i$ elements from $S_{i+1}$ and push them into $S_i$. Again if $S_{i+1}$ becomes empty during this process, we recurse. Assume push and pop operations for each stack takes $O(1)$ time. The pseudo-code is as follows:

```
MultiPush(x):

  i = 0;
  while (S_i is full)
     i = i+1;
  while (i > 0) {
     i = i-1;
     for j=1 to 3^i
        Push(S_{i+1}, Pop(S_i));
  }
  Push(S_0, x);


MultiPop():
   i = 0;
   while (S_i is empty)
      i = i+1;
   while (i > 0) {
      i = i-1;
      for j=1 to 3^i
          Push(S_i, Pop(S_{i+1});
   }
   return Pop(S_0);
```

   - In the worst case, how long does it take to push one more element onto a multistack containing $n$ elements?
   - Prove that if the user never pops anything from the multistack, the amortized cost of a push operation is $O(\log n)$, where $n$ is the maximum number of elements in the multistack during its lifetime.

- Prove that in any intermixed sequence of pushes and pops, each push or pop operation takes $O(\log n)$ amortized time, where $n$ is the maximum number of elements in the multistack during its lifetime.

2. Suppose you are faced with an infinite number of counters $x_i$, one for each integer $i$. Each counter stores an integer mod $m$, where $m$ is a fixed global constant. All counters are initially zero. The following operation increments a single counter $x_i$; however, if $x_i$ overflows (that is, wraps around from $m$ to 0), the adjacent counters $x_{i1}$ and $x_{i+1}$ are incremented recursively. Here is the pseudocode:

```
Nudge_m(i)

  x_i = x_i + 1;
  while (x_i >= m) {
      x_i = x_i - m;
      Nudge_m(i+1);
      Nudge_m(i-1);
  }
```

   - Suppose we call $Nudge_3$ $n$ times starting from the initial state when all counters 0. Note that each call can start from any of the counters. Show that the amortized time complexity is $O(1)$.
   - What is the amortized time complexity in the above question if the global counter $m$ is 2 instead of 3 ?

3. You are given a tree $T$ where each vertex $v$ has an integer $val(v)$ stored in it (you can assume that all the integers involved are distinct). A vertex $v$ is said to be a *local minimum* if $val(v) \leq val(w)$ for all the neighbours $w$ of $v$. Show how to find a local minimum in $O(n)$ time, where $n$ is the number of vertices in $T$.

   Solve the same problem when the graph is an $n \times n$ grid graph. An $n \times n$ grid graph has vertices labelled $(i, j)$, where $1 \leq i, j \leq n$ and $(i, j)$ is adjacent to $(i-1, j), (i+1, j), (i, j-1), (i, j+1)$ (with appropriate restrictions at the boundary). The time taken by the algorithm should be $O(n)$.

4. (a) Let $n = 2^l - 1$ for some positive integer $l$. Suppose someone claims to hold an unsorted array $A[1 \ldots n]$ of distinct $l$-bit strings; thus, exactly one $l$-bit string does not appear in $A$. Suppose further that the only way we can access $A$ is by calling the function $FB(i, j)$, which returns the $j^{th}$ bit of the string $A[i]$ in $O(1)$ time. Describe an algorithm to find the missing string in $A$ using only $O(n)$ calls to $FB$.

   (b) Now suppose $n = 2^l - k$ for some positive integers $k$ and $l$, and again we are given an array $A[1 \ldots n]$ of distinct $l$-bit strings. Describe an algorithm to find the $k$ strings that are missing from $A$ using only $O(n \log k)$ calls to $FB$.

5. Suppose you are given an array $A[1 \ldots n]$ of numbers , which may be positive, negative, or zero, and which are not necessarily integers.

   (a) Describe and analyze an $O(n)$ time algorithm that finds the largest sum of elements in a contiguous subarray $A[i \ldots j]$.

   (b) Describe and analyze an $O(n)$ time algorithm that finds the largest product of elements in a contiguous subarray $A[i \ldots j]$.