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 ith 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?

     Solution: If the first l stacks are full, then time taken is proportional to 3^l, which is $\theta(n)$.

- 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.

  Solution: First note that if move an element from S_i to S_{i+1}, then it must be the case that S_i is full at this moment. But then $n \geq 3^i$ and so $i \leq \log n$. This shows that if the highest occupied stack is S_ℓ, then ℓ is $O(\log n)$. Notice that each element only moves to a higher stack, and so the total amount of movements of the n elements is at most $O(n \log n)$. Hence, the amortized cost is $O(\log n)$.

- 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.

  Solution: We again perform an aggregate analysis. We show that for a fixed i, the total number of movements of elements from S_i to S_{i+1}, call it M_i, is $O(n)$. Since the total number of movements from S_{i+1} to S_i cannot be more than $M_i + n$, it is enough to bound M_i.

  Now observe that when we move elements from S_i to S_{i+1} during a push operation at some time t, each of the stacks $S_1, \ldots S_i$ is one-third full after this push operation (because we move the contents of S_j to S_{j+1}). Now, in order for us to move elements from S_i to S_{i+1} again in a subsequent push operation at some time t_1, one of the following events must have happened: (i) we moved an element down from S_{i+1} to S_i, or (ii) no element moved down from S_{i+1} to S_i. In the first case, let t' be the first time after t when this event happens. Then we must have popped at least $\frac{1}{3} \times 3^i = 3^{i-1}$ elements (because S_i is empty and no element has moved up from SI during $[t, t']$, or (ii) the stacks S_i is full during the push operation at time t_1. Since no element moved down to S_i during $[t, t_1]$, and time t, the stacks $S_1, \ldots S_i$ hold together at most $\frac{1}{3}(1 + 3 + \ldots + 3^i) \leq 2 \cdot 3^{i-1}$ elements, at least 3^{i-1} push operation must have happened during $[t, t_1]$.

  In either case, we see that at least 3^{i-1} push or pop operations must have happened during $[t, t_1]$. Since at most 3^i elements move during t, we see that M_i is at most a constant times the total number of push or pop operations. Finally, there are only $O(\log n)$ levels, and so the result follows.

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 $O(1)$.

  The idea is to see what this operation does. Suppose we are calling this function at some counter i. If this counter is not equal to 2, then we just change this counter, and the process ends there. Now suppose the counter is 2, then the process can cascade. Some thought will reveal the following: suppose x_i is 2, and let $j \leq i \leq k$ be indices such that $x_j, x_{j+1}, \ldots, x_i, \ldots, x_k$ are all 2, but $x_{j-1}, x_{k+1} \neq 2$. Then this process will replace $x_j, \ldots, x_k$ by 1, and may change x_{j-1} and x_{k+1}. Further, the total number of addition operations is at most $2(k-j)+O(1)$. Therefore, we define the potential function Φ_i after i operations as twice the number of counters which are 2. Now it is easy to check that the time taken plus the change in potential is $O(1)$.

- What is the amortized time complexity in the above question if the global counter m is 2 instead of 3 ?

  In this case, the amortized time complexity after n operations is $O(n)$. The reason is the following: suppose counters $x_i, x_{i+1}, \ldots, x_j$ are all 1. Now if we call increment at any of these counters, the counters $x_{i-1}, x_i, \ldots, x_{j+1}$ will be 1, and the time taken will be at least $j-i$. So if perform such operations n times, the total time taken is $O(n^2)$.

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.

   Solution: Start at the root. If it is a local minimum, we can stop. Otherwise there is a child node with value less than that at the root. Now solve recursively at the child node. Since we spend constant time at each node, the running time is $O(n)$.

   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 $O(n)$.

   Solution: We will use divide and conquer and the recurrence $T(n) = T(n/2) + O(n)$. For a square S in the grid, let $B(S)$ be the squares which are not in S, but share an edge one of the squares in S (i.e., these are the squares which are next to the boundary of S). When we recurse on a squre S, we cannot just find a local minimum in S, but need to make sure that this is a local minimum when we also take $B(S)$ into account. So whenever we recurse on a square S, we will guarantee that there is a square in S which is a local minimum in $S \cup B(S)$ (and hence, a local minimum in the entire square).

We start with the original square. We divide it into four equal parts of size $n/2 \times n/2$. Let X denote the boundary squares (i.e., squares which lie on the boundary of one of these squares). Note that size of X is $O(n)$. We can also check in $O(n)$ time if any of these is a local minimum. If so, we can stop. So assume this case does not happen.

Let S_1, S_2, S_3, S_4 be the squares obtained by removing the squares in X. Let p be the smallest element in a square in X. Since p is not a local minimum, there is a square in one of $S_1, \ldots, S_4$ which is smaller than p. Say it is S_1. Note that there is a square in S_1 which is the local minimum with respect to the entire square (for example, the smallest element in S_1 will have this property). So we recurse in S_1. Thus, we have the recurrence $T(n) = T(n/2) + O(n)$,.

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.

   Solution: We use the recurrence $T(n) = T(n/2) + O(n)$. Find the first bit of all the numbers. One of the following two cases will happen : (i) There are 2^{l-1} numbers with first bit 1, and $2^{l-1} - 1$ numbers with first bit 0: in this case recurse on the latter set of numbers (from now on ignore the first bit, and so these 2^{l-1} numbers will be distinct and exactly one $l-1$ bit number will be missing, (ii) this case is symmetric and handled in the same manner.

   (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.

   Solution: We use the same idea, but details are more non-trivial. Our recursive procedure *Find*(S, h, m) will have three arguments – S will be a set of distinct h bit numbers, with m numbers missing (i.e., $|S| = 2^h - m$) and we want to output these m numbers). Note that initially we call *Find*(A, l, k), where A is the set of all numbers in the array. Let us describe the procedure *Find*(S, h, m). As above, we look at the first bit of all the numbers in S. Let S_1 be the ones which have first bit as 1, and S_0 be the ones with first bit being 0. Say $|S_1| = 2^{h-1} - m_1, |S_0| = 2^{h-1} - m_0$. Then $m_0 + m_1 = m$. Now we ignore the first bits from S_0 and S_1 and think of these as $h-1$ bit numbers. If $m_0 \neq 0$, we recursively call *Find*$(S_0, h-1, m_0)$. Similarly, if $m_1 \neq 0$, we recursively call *Find*$(S_1, h-1, m_1)$.

   We now analyze the running time. First observe that the running time is at most $O(nl)$. Indeed, the recursion tree has depth l and we pay $O(n)$ at each level. So if $k \geq 2^{l-1}$, we are done. Since $n + k = 2^l$, we can assume from now on that $n \geq 2^{l-1}$. So we will now show that the running time is $O(2^l \log k)$. Let $T(l, k)$ be the running time when we have l bit strings out of which k strings are missing. The recurrence is

$$T(l, k) = T(l-1, k_1) + T(l-1, k_2) + 2^l,$$

where $k_1+k_2=k$. We assume $k_1, k_2 > 0$, otherwise we don't have a recursive term for this part. We show by induction that the solution is $T(l,k) \le 2^{l+1}\log k$, where all logarithms are to the base 2. By induction assume $T(l-1,k_1), T(l-1,k_2)$ are of this form. Also assume that $k_1 \le k_2$. In this case, $k_1 \le k/2$ and so $\log k_1 \le \log k - 1$. So we get (by induction hypothesis)

$$T(l,k) \le 2^l \log k_1 + 2^l \log k_2 + 2^l \le 2^l(\log k - 1) + 2^l \log k + 2^l = 2^{l+1}\log k.$$

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 $A[i\ldots j]$.

   Solution: We use dynamic programming. Let $T(k)$ denote the maximum sum subarray of A which ends at $A[k]$. Once we know the $T(k)$ values, we can output their maximum. Suppose we have computed $T(1),...,T(k-1)$. Now there are two choices for $T(k)$: either the maximum subarray ending at $A[k]$ contains $A[k-1]$, in which case $T(k)=T(k-1)+A[k]$, or it does not, in which case $T(k)=A[k]$. Thefore,

   $$T(k) = \max(T(k-1)+A[k], A[k]).$$

   Another way of saying this is that if $T(k-1) < 0$, then $T(k) = A[k]$ else it is $T(k-1)+A[k]$.

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

   Solution: Again same idea as above, but because product can change the sign, we maintain two arrays: $P(k)$ denotes the maximum positive product (which could be 0) one can get from a subarray which ends at $A[k]$, and similarly $N(k)$ is the maximum in absolute value negative product one can get from a subarray which ends at $A[k]$. Now two cases arise depending on whether $A[k]$ is positive or negative. If it is positive, then $P(k) = \max(P(k-1)\cdot A[k], A[k])$, $N(k) = \max(N(k)\cdot A[k], A[k])$. If $A[k]$ is negative then $N(k) = \max(P(k-1)\cdot(-A[k]), -A[k])$.