600.363 Introduction to Algorithms / 600.463 Algorithms I **Fall 2014**
Homework #3 **Due:** September 23, 2014, 1:30pm

Please start each problem on a new page, and include your name on each problem. You can submit on blackboard, under student assessment.

Remember: you may work in groups of up to three people, but must write up your solution entirely on your own. Collaboration is limited to discussing the problems – you may not look at, compare, reuse, etc. any text from anyone else in the class. Please include your list of collaborators on the first page of your submission. You may use the internet to look up formulas, definitions, etc., but may not simply look up the answers online.

Please include proofs with all of your answers, unless stated otherwise.

1 List Update (33 points)

Suppose we have n items $x_1, x_2, \dots, x_n$ that we wish to store in a linked list. The cost of a *lookup* operation is the position in the list, i.e. if we lookup some element x_i we pay 1 if it is the first element, 2 if it is the second element, 3 if it is the third element, etc. This models the time it takes to scan through the list to find the element.

For example, suppose there are four items and we lookup x_1 twice, x_2 three times, x_3 once, and x_4 four times. Then if we store them in the list (x_1, x_2, x_3, x_4) the total cost is $1(2) + 2(3) + 3(1) + 4(4) = 27$. On the other hand, if we stored them in the list (x_4, x_2, x_1, x_3) the cost would be $1(4) + 2(3) + 3(2) + 4(1) = 20$. It is easy to see that if we knew the number of times each element was looked up, the optimal list is simply sorting by the number of lookups, i.e. the first element is the one looked up most often, then the element looked up next most often, etc.

But what if we do not know how many times each element will be looked up? It turns out that a good strategy is *Move-to-Front* (MTF): when we lookup an item, we also move it to the head of the list. Let's say that moving an element is free (it is easy enough to implement with a constant number of pointer switches, in any case). So if, for example, we start with the list $(x_1, x_2, \dots, x_n)$ and then lookup x_4, we pay a cost of 4 and afterwards the list will be $(x_4, x_1, x_2, x_3, x_5, x_6, \dots, x_n)$. Then if we lookup x_4 again we only have to pay a cost of 1.

(a) Suppose $n = 4$ and we use MTF starting with the list (x_1, x_2, x_3, x_4). We perform the following operations: lookup(x_4), lookup(x_2), lookup(x_4), lookup(x_2). What is the list after these operation, and what was the total cost?

(b) Consider a sequence of m operations (think of m as being much larger than n). Let C_{init} be the cost of these operations if we used the original list $(x_1, x_2, \dots, x_n)$ the entire time (i.e. we do not use MTF). Let C_{MTF} be the cost if we use MTF on the same operations starting from the original list. Prove that $C_{MTF} \leq 2C_{init}$.

Hint: Consider a potential function equal to the number of inversions relative to the initial list. Equivalently, think of each item x_i as having a "piggy bank" in which the number of tokens is the number of elements before it in the current list that come after it in the initial list.

(c) Now let C_{static} be the smallest possible cost among all fixed lists. In other words, given the same sequence of lookups, each fixed list (without MTF) has some cost, and let C_{static} be the minimum of those costs. Note that this will correspond to the list that is sorted by the number of accesses, like in the first example. Prove that $C_{MTF} \leq 2C_{static} + n^2$.

2 More counters (33 points)

We saw in class that if we have a binary counter which we increment n times the total cost (measured in terms of the number of bits that are flipped) is $O(n)$, i.e. the amortized cost of an increment is $O(1)$. What if we also want to be able to decrement the counter? Throughout this problem we will assume that the counter never goes negative – at every point in time the number of increments up to that point is at least as large as the number of decrements.

(a) Show that it is possible for a sequence of n operations (increments and decrements) to have amortized cost of $\Omega(\log n)$ per operation (so the total cost is $\Omega(n \log n)$). This should hold even if we start from 0 and the counter never goes negative.

(b) To decrease this cost, let's consider a new way of representing numbers: a *redundant ternary* number system. A number is represented as a sequence of *trits* (as opposed to the more usual bits or digits), each of which is $0, -1$, or $+1$. The value of the number represented by $t_{k-1}, \ldots, t_0$ (where each t_i is a trit) is defined to be $\sum_{i=0}^{k-1} t_i 2^i$.

Note that the same number might have multiple representations. This is why this system is a *redundant* ternary system. For example, `1 0 1` and `1 1 -1` both represent the number 5.

Incrementing and decrementing work as you would expect. When incrementing, we add 1 to the low order trit. If the result is 2, then we change it to 0 and propogate a carry to the next trit. This is repeated until no carry results. Similarly, when we decrement we subtract 1 from the low order trit. If the result is -2, we set it to 0 and propogate a borrow (i.e. subtract 1 from the next lowest order trit). Again, we repeat this until no borrow is necessary.

The cost of an increment or decrement is the number of trits that change in the process. Suppose that we perform a sequence of n increments and decrements, starting from 0. Prove that the amortized cost of each operation is $O(1)$, i.e. the total cost is $O(n)$. Hint: think about a "potential function" or "bank account" argument.

3 Better analysis of Union-Find (33 points)

In this problem we will prove an amortized bound of $O(\log^* n)$ on the Union-Find data structure when using both union by rank and path compression. Here $\log^* n$ is the iterated logarithm function, which informally is the number of times we need to apply the logarithm function to n before we get a value that is at most 1. Formally, we use the definition

$$\begin{aligned} \log^* n &= 0 && \text{if } n \leq 1 \\ \log^* n &= 1 + \log^*(\log n) && \text{if } n > 1 \end{aligned}$$

So we will want to show that if there are m operations, of which at most n are Make-Set (so there are only n elements), the total run time is $O(m \log^* n)$ as long as we use both union by rank and path compression.

(a) Prove that throughout the algorithm, if u is the parent of v then the rank of u is strictly larger than the rank of v.

(b) Let u be an arbitrary node with rank r. Prove that at some point u was the root of a tree that had at least 2^r nodes.

(c) Let u be an arbitrary node. Prove that for any rank r, there is at most one node of rank r that is ever an ancestor of u.

(d) Using the previous parts, prove that there are at most $n/2^r$ nodes of rank r.

(e) Let's set up some more notation. For any integer i, let $2 \uparrow i$ be 2 to itself i times. So $2 \uparrow 0 = 1$, $2 \uparrow 1 = 2$, $2 \uparrow 2 = 2^2 = 4$, $2 \uparrow 3 = 2^{2^2} = 16$, $2 \uparrow 4 = 2^{2^{2^2}} = 2^{2^4} = 2^{16} = 65536$, etc. Note that $\log^* n$ is just the smallest integer i so that $2 \uparrow i \geq n$.

Now let's think of $\log^* n$ buckets, where bucket i contains all nodes with rank at least $2 \uparrow i$ and at most $(2 \uparrow (i+1)) - 1$. So i ranges from 0 to $\log^* n - 1$. Bucket 0 contains all nodes of rank 1, bucket 1 contains all nodes with rank 2 or 3, bucket 2 contains all nodes with ranks in $[4, 15]$, etc. We'll also have a special bucket -1 for nodes of rank 0. Note that all of this bucketing is going on *only in the analysis*; the algorithm itself doesn't use the bucketing or keep track of it.

Prove that the number of elements in bucket i is at most

$$\frac{2n}{2^{(2\uparrow i)}} = \frac{2n}{2 \uparrow (i+1)}.$$

(f) We will now start to bound the total cost of Find operations (clearly Union operations cost only $O(1)$ each). Note that there are at most m Find operations, and the cost of each Find operation is (up to a constant factor) the length of the path from the node to the root of its tree. Let's divide the edges of such a path into two parts: edges between nodes in different buckets, and edges between nodes in the same bucket.

Prove that in any Find path (a path from a node to the root of its tree) the number of edges that go between buckets is at most $O(\log^* n)$, and thus the total number of such traversals is $O(m \log^* n)$.

It turns out that this is now enough, although it's a little tricky to prove. But intuitively, note that although there can be many edges between nodes in the same bucket in a given path, if there's even a single edge we traverse between nodes in different buckets then path compression forces all of the edges below that edge to be inter-bucket. So in fact the inter-bucket edges dominate when we amortize, so the total cost of all m Find operations is $O(m \log^* n)$, and thus the amortized cost of a Find operation is only $O(\log^* n)$ as we wanted!