Homework 3

Matt Forbes

November 5, 2010

1 Problem One

In an array of numbers with the property given in this problem (each element is at most 1 greater or less than it's neighbors), then the following is true:

Given two indices in the array A, say i and n such that A[i] = a, and A[n] = b, then A[k] = x for some k in i . . . n and x between a and b.

This is true because the array A is 'continous' in the sense that because each element can only differ from the preceding by at most 1, then to get from A[i] = a, to A[n] = b, you must hit each value between on the way, in an unknown order.

Psuedo Code

```
def find_z(A, n, z):
    if A[1] == z: return i
    if A[n] == z: return n

    if z is not in A[1] \dots A[n]:
        raise 'bad input!'

    i = 1
    j = i + (n - i)/2
    while A[j] != z :
        min = A[i]
        max = A[n]
        mid = A[j]

        if min < z < mid:
            max = mid
            n = j
        else:
            min = mid
            i = j
```

```
        j = i + (n - i) / 2
    end while

    return j
```

Loop Invariant

A[k] = z for some k in i . . . n.

Proof

Basis:

> At the start of the first iteration, the L.I. holds by the problems definition: $x \leq z \leq y$ where x=A[i] and y=A[n].

Maintenance:

> Suppose we are in an iteration of the loop, and the L.I. has held up to this point. Then, at the beginning of this loop, A[k] = z for some k in i . . . n. Let j = middle index between i and n. Now, if z is between A[i] and A[j], then there exists an index k between i and j such that A[k] = z. In this case, the algorithm sets n = j, and continues. Because z is in this subarray, the L.I. holds after the iteration. Similarly with the case that z is between A[j] . . . A[n]. One of these cases must occur because the loop invariant says that A[k] = z for some k in i . . . n.

Termination

> In each iteration, A[k] = z for a k in i . . . j, but the gap between i and j is halved, so the loop is guaranteed to complete eventually (even if only one element remains in the subarray). The loop terminates when A[j] == z, so we have found our j.

Analysis

There are two comparisons done each iteration of this loop. So the number of comparisons (denoted C, and number of elements n) is:

$$C(n) = \left\{ \begin{array}{cc} 2 + C(n/2), & \text{for n} > 2 \\ 1, & \text{for n} \leq 2 \end{array} \right\}$$

This is the same recurrence relation from the last homework, and I found it to be O(log n).

2 Problem Two

a)

let x_i be an IRA denoting that the i^{th} guess was correct.

X be a random variable denoting the total number of correct guesses

E[x] be the expectation of X

c be the number of cards in the deck

n be the number of guesses, $n \leq c$

$$
\begin{aligned}
Pr\{x_i\} &= \frac{1}{c} \\
X &= \sum_{i=1}^{n} x_i \\
E[X] &= E\left[\sum_{i=1}^{n} x_i\right] \\
&= \sum_{i=1}^{n} E[x_i] \\
&= \sum_{i=1}^{n} Pr\{x_i\} \\
&= \sum_{i=1}^{n} \frac{1}{c} \\
&= \frac{n}{c}
\end{aligned}
$$

b) In this scenario, the probability of each guess is dependant on how many cards have already been flipped over. The probability of guessing the i^{th} card correctly is going to be $\frac{1}{c-i}$ where c is the number of cards in the deck. Besides that note, the calculation

of the expectation of X is the same as in part a. So we'll jump ahead a bit.

$$\begin{aligned} E[X] &= \sum_{i=1}^{n} Pr\{x_i\} \\ &= \sum_{i=1}^{n} \frac{1}{c-i} \\ &\leq \sum_{i=1}^{n} \frac{1}{n-i} \\ &\leq \sum_{i=1}^{n} \frac{1}{i} \\ &= O(\log n) \end{aligned}$$

Problem 3

a) The search algorithm for this data structure may need to search each individual array A_i to find an element.The elements of each array are only sorted in terms themselves, and are not related to the elements in the other arrays. There are always $\log n$ arrays in the structure, so we would do a binary search on each array until we found the element (or didn't) we were looking for.

Each array A_i has 2^i elements, so a binary search on that array would have a running time of $\log_2(2^i) = i$. The total running time t of the search procedure would be:

$$\begin{aligned} \text{let k} &= \lceil logn \rceil \\ t &= \sum_{i=0}^{k} \log_2 2^i \\ &= \sum_{i=0}^{k} i \\ &= \frac{k(k+1)}{2} \\ &= \frac{1}{2}\left(\log^2 n + \log n\right) \\ &= O(\log^2 n) \end{aligned}$$

b) Insertion - The basic idea of inserting in to this structure is to start from the left and move all of the elements in the 'filled' arrays in to the first 'empty' array. We keep the sorted order of these elements by doing a sort of merge similar to what merge_sort performs. By moving the left-most filled elements and the element to insert in to the first empty array preserves the properties that are defined for this structure.

Psuedo Code

```
// A is the sorted structure
// e is element to insert
// size(A) is number of elements in A
def insert(A, e):
    n = size(A)
    N = binary representation of n, LSB is N[0]
    j = the index of the first occurence of 0 in N
    ixs = new array sized j-1 elements all set to 0
    inserted = false
    insert_ix = 0

    // merge subarrays
    for i = 1 ... 2^j:
        if not inserted:
           min = e

        for ii = 1 ... (j-1):
            ix = ixs[ii]
            if ix <= 2^(ii)  and  A[ii][ix] < min:
                min = A[ii][ix]
                insert_ix = ii
        endfor

        A[j][i] = min

        if min == e:
            inserted = true
        else:
            ixs[insert_ix]++
    endfor

    size(A)++

end
```

To make analyzing this procedure easier, I have calculated the running time of a single insert parameterized by the value of j (index of the first 'empty' array). The main loop of the procedure has 2^j iterations, and it's inner loop has $j-1$ iterations. So the running time is roughly $j2^j - 2^j$.

Worst Case analysis

Using the above calculation, the worst case would be when the first empty array is indexed at $\log n$, which is the largest possible value of j. The running time (T) of that insert would be:

$$\begin{aligned} T &= (\log_2 n)2^{log_2 n} - 2^{log_2 n} \\ &= n \log n - n \\ &= O(n \log n) \end{aligned}$$

Amortized Analysis

The worst case analysis is a horrible upper bound of what the running time would actually be. In this amortized analysis, we will find the running time of performing n insertions, and then divide by n to find the average running time of a single insertion.

The book provides a handy observation that when doing n increments of a binary number, the bit b_i is flipped exactly $\frac{n}{2^i}$ times. Well we know what the running time of an insertion based on which bit is flipped, so we can calculate the running time of all n of these insertions. For each bit index, we know the running time of an insertion in that state, and how many times that happens.

Running time of n insertions T(n):

let k = $\lceil \log_2 n \rceil$

$$\begin{aligned} T(n) &= \sum_{i=1}^{k} (\frac{n}{2^i})(i2^i - 2^i) \\ &= \sum_{i=1}^{k} ni - n \\ &= \sum_{i=1}^{k} n(i-1) \\ &= n \sum_{i=1}^{k} i - 1 \\ &= n \left(\sum_{i=1}^{k} i - \sum_{i=1}^{k} 1 \right) \\ &= n \left(\frac{k^2 + k}{2} - k \right) \\ &= n \left(\frac{\log^2 n + \log n}{2} - \log n \right) \end{aligned}$$

Average running time of a single insertion T(1):

$$\begin{aligned} T(1) &= \frac{T(n)}{n} \\ &= \frac{\log^2 n + \log n}{2} - \log n \\ &= O(\log^2 n) \end{aligned}$$

c) Deletion - The element being deleted could be anywhere in the structure. Once that element is removed, the properties of the structure are no longer maintained because the subarray that contained that element is no longer full. So the goal now is to juggle these subarrays and get them in to the correct form.

A potential algorithm for deleting an element would be to basically do the reverse of the insertion procedure, except we have to refill the subarray that had an element deleted before doing the 'decrement.'

First, we would find the first bit set to 1 starting from the LSB in N, and save that index as j. This subarray is going to be emptied and will be used to fill the preceding empty subarrays behind it. It has one too many elements to fill these subarrays, so we use the first or last element to plug the hole that we created by removing an element. Remembering that all the subarrays need to be sorted, we would want to find a quick way of refilling that subarray while keeping it sorted.

Now the j^th subarray has just enough elements to fill the preceding subarrays and it is in sorted order, so we can just split it up and use consecutive slices to fill them.

The running time of this delete would be probably depend on how efficient moving that one extra element to fill the whole takes, because the refilling of the preceding subarrays is really quick, because there is no merging or sorting.