CSCI 3104 Fall 2017 HW 3
Austin Griffith (9/24)

---

1. *Professor Snape has n magical widgets that are supposedly both identical and capable of testing each others correctness. Snapes test apparatus can hold two widgets at a time. When it is loaded, each widget tests the other and reports whether it is good or bad. A good widget always reports accurately whether the other widget is good or bad, but the answer of a bad widget cannot be trusted. Thus, the four possible outcomes of a test are as follows:*

   (a) *Prove that if n/2 or more widgets are bad, Snape cannot necessarily determine which widgets are good using any strategy based on this kind of pairwise test. Assume a worst-case scenario in which the bad widgets are intelligent and conspire to fool Snape.*

   If n/2 widgets are bad assume that all bad widgets disguise themselves as good widgets and declare any good widgets as bad widgets. All the good chips will correctly report their partners as bad widgets by the terms of the problem. Suppose that there are k number of good chips so there is (n-k) bad chips which results in an inaccurate result. We can conclude that there is

   ```
   (n-k)  > k
   ```

   relation for bad to good chips from the provided givens of the problem. Of the possible outcomes, there are more instances of bad chips than good chips and it can be concluded that there will always be at least one bad chip. Therefore professor will not be able to distinguish which widget is good or bad because they are equally displaying opposing readings.

   (b) *Consider the problem of finding a single good widget from among the n widgets, assuming that more than n/2 of the widgets are good. Prove that n/2 pairwise tests are sufficient to reduce the problem to one of nearly half the size.*

   Assuming that n is even so we can match widgets in pairs. If we throw away all pairs that do not say (good,good) then we keep more good widgets than bad

widgets because all pairs thrown away contain at least one bad widget. Assuming n is odd we continue with this method and test all pairs to get rid of anything that does not report back a (good,good) widget combo.

Assuming we have discard any pairs of widgets that have at least one widget that reports a bad partner, from the first pair to floor(n/2) widgets, this leaves us with only (good, good) pairs of widgets left because we know that there are more than n/2 good widgets and we know there will be at least one true (good, good) pair. It also tells us that we are left with 4k+3 widgets for some number k. In this case there are 2k+2 good widgets and k+1 pairs of (good,good) and at most k pairs of (bad,bad) widgets.

Finally, assuming that we have discarded the pairs of widgets that report anything but (good,good), and we are left with 4k+1 widgets, for some integer k. Then, in this case there are at least 2k + 1 good widgets. Now, assume that one of the leftover pairs is bad; then there are at least 2k+2 good widgets, which is, k+1 pairs of (good,good) widgets and at most k-1 pairs of bad widgets. By taking a single widget from every pair at least k+1 good widgets and at most (k-1)+1 = k bad widgets we have a majority of good widgets and our final set is at most of size ceiling(k/2). On the other hand, assume that one of the leftover pair is good. Then we have at least k pairs of (good,good) widgets and taking a widget from each pair gives us a majority of good widgets and a final set of size at most ceiling(k/2).

(c) *Prove that the good widgets can be identified with (n) pairwise tests, assuming that more than n/2 of the widgets are good. Give and solve the recurrence that describes the number of tests.*

We can find at least two good widgets out of n/2 widgets based on conclusions from 1b if at least n=2 of the widgets are good. We use that one good widget to test the rest of the n widgets since we know the good widget wont lie. Thus finding a bad widget will take O(n) time. To find the original good pair of widgets we can display that the original good widget discovery can be represented by recurrence relation:
T(n) = T(ceiling[n/2]) + floor(n/2) .
T (1) = 0.
The solution to this recurrence is T (n) = O(n) and can be shown by T(n) is

greater than or equal to $2^{(}ceiling(logn))$

.

2. *Professor Dumbledore needs your help. He gives you an array A consisting of n integers A[1], A[2], . . . , A[n] and asks you to output a two-dimensional n n array B in which B[i,j] (for i ¡ j) contains the sum of array elements A[i] through A[j], i.e., the sum A[i] + A[i + 1] + + A[j]. (The value of array element B[i,j] is left unspecified whenever i j, so it doesnt matter what the output is for these values.) Dumbledore suggests the following simple algorithm to solve this problem:*

   (a) *For some function f that you should choose, give a bound of the form O(f(n)) on the running time of this algorithm on an input of size n (i.e., a bound on the number of operations performed by the algorithm).*

   Outer loop takes O(n) time
   Inner loop takes O(n) time
   Summing array elements takes O(n) time
   Worst case running time of algo = $O(n^3)$ time

   (b) *For this same function f, show that the running time of the algorithm on an input of size n is also (f(n)). (This shows an asymptotically tight bound of (f(n)) on the running time.)*

   $f(n) = \Omega(g(n))$ if $0 \leq \Omega(f(n)) \leq cf(n)$ for all $n > n_0$
   $g(n) = n^3 + 1$
   if we let $c = 1$ and $n_0 = 0$
   $n^3 + 1 \geq 1(n^3)$ for all $n > n_0$
   Therefor $f(n) = \Omega(n^3)$

   (c) *Although Dumbledores algorithm is a natural way to solve the problemafter all, it just iterates through the relevant elements of B, filling in a value for each it contains some highly unnecessary sources of inefficiency. Give an algorithm that solves this problem in time O(f(n)/n) (asymptotically faster) and prove its correctness.*

```
dumble_dore_Solved(A)
    currentSum = 0;
    for(int i = 0 to A.length-1)
    {    currentSum = A[i];
         for (int j = i+1 to A.length)
         {      currentSum = currentSum + A[j]
                B[i,j] = currentSum
         }
    }
```

Outer loop takes O(n) time
Inner loop takes O(n) time
Summing currentSum takes O(1) time
Assigning array element takes O(1) time
Worst case running time for dumbleSolve is O(n^2) time

Proving Correctness:
We start by setting a new variable labeled currentSum to A[i] in each iteration of i. For each iteration of j, we store the current sum of A[i] to A[j] each time, so for each "new" j = j+1, all you have to do to add A[j] to the current sum. Since the value in currentSum is A[i] + A[i+1] + ... + A[j-2] + A[j-1] from the previous iteration, so adding A[j] to the current sum will give us the desired sum A[i] + A[i+1] + ... + A[j]. The step of adding A[j] to the current sum can be done in constant time, so now the inner loop has a constant time as well. Just like in part b, the total number of iterations of the inner loop is: $(1/2)n^2 - (1/2)n$,
$sothetotalrunningtimeis$:
$O(n^2)$.

3. *With a sly wink, Dumbledore says his real goal was actually to calculate and return the largest value in the matrix B, that is, the largest subarray sum in A. Butting in, Professor Hagrid claims to know a fast divide and conquer algorithm for this problem that takes only O(n log n) time (compared to applying a linear search to the B matrix, which would take O(n2) time).*

   (a) *i:Identify and fix the errors in Hagrids code*

```
Errors found:

Line 10: for( i = m, i >= , i--)
Line 12: if(sum > leftMax) {leftMax = sum}
Line 15: for( i = m + 1, i < t, i++)
```

   (b) *ii:prove that the corrected algorithm works*

```
Base case: A.length = 1 so 0 <= i < 1 and 0 <=j < 1 trivially.

Inductive Step: Let a be a contiguous sub-sequence of A that has
the largest sum which we will call
v where max(maxLeft, maxRight,spanMax) = v

Case 1:
If a = A[s,t] maxLeft and maxRight are sums
of other sub-sequences and thus <= v


Case 2:
If a lies entirely to the left of m  then by our hypothesis
maxLeft = v and
max(maxLeft, maxRight,spanMax) = v
```

```
Case 3:
If a lies entirely to the right of m
then by our hypothesis maxRight = v
and max(maxLeft, maxRight,spanMax) = v

In all cases max(maxLeft, maxRight,spanMax) = v
```

(c) *iii:give the recurrence relation for its running time*

T(n) = 2T(n/2) + Θ(n)
Problem is split in to two problems both with n/2 size.

Proof f(n) = Θ(n):

n = t-s+1
t = n+s-1
first iteration m = (s+t)/2
for (s-m+1 iterations)
s - (s+t)/2 + 1 //sub in t
s - (s + n + s -1)/2 + 1
= s - s + n/2 + 0
= n/2 = Θ(n/2) = Θ(n)

(d) *iv:solve for its asymptotic behavior.*

a = 2; b = 2; f(n) = Θ(n)

Master Method 2:

f(n) = $\Theta(n^{log_2 2})$ = Θ(n)
Therefor: T(n) = $\Theta(n^{log_2 2}\lg n)$ = Θ(nlgn) 1 ‘

4. *Suppose that we modify the Partition algorithm in QuickSort in such a way that on alternating levels of the recursion tree, Partition either chooses the best possible pivot or the worst possible pivot. Write down a recurrence relation for this version of QuickSort and give its asymptotic solution. Then, give a verbal explanation of how this Partition algorithm changes the running time of QuickSort.*

   To solve this problem we need to cover the two scenarios that are present. The recurrence relation, as well as the time complexity of both, is going to be different. One scenario is when we chose the best pivot possible and second scenario is when we chose worst pivot possible

   Best Pivot:

   When we are choosing the best pivot the recurrence relation will be

   $T(n)= 2T(n/2)+ f(n)$

   and the time complexity

   $T(n)= O(n \log n)$

   Worst Pivot:

   When we choose the worst pivot possible then the recurrence relation is going to be

   $T(n)= T(n-1)+ f(n)$

   time complexity is going to be

   $T(n)= O(n^2)$.

Verbal Explanation:

The partition algorithm which changes the running time of quicksort is because the pivot is controlling the run time of the algorithm. Partition decides whether it is going to be best, average or worst case, so when the algorithm decided that best pivot will be provided, it shows us that there won't be a case when we would or in reverse order be solving absolute reversal of the input array. When we are choosing the worst pivot, we are making sure that we are sorting it from the reversed order and it will give us $O(n^2)$ time complexity. That is why randomized quick sort was introduced to create randomness in choosing the pivot even if the list is in reverse order.