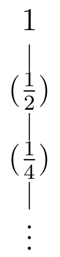
CSCI 3104 Spring 2017 Problem Set 3
Matthew Niemiec (06/24)
Mehmet Karaoglu (07/30)
Amulya Srivastava (09/02)

1. *(5 pts) Why do we analyze the average-case performance of a randomized algorithm and not its worst-case performance? Succinctly explain.*

   When analyzing the worst-case runtime of algorithms, we take an "adversarial input". This means that we imagine someone who wants to see our algorithm run really slowly and fail, and imagine the type of input that they might give us. However, when we randomize a choices, we take away the ability of our adversary to give us bad inputs, and instead leaves everything up to chance. This chance, which then determines the run time, is so small of being the worst that it is practically a statistical impossibility. Thus, being realistic, we judge the run time by its average run time and not its worst.

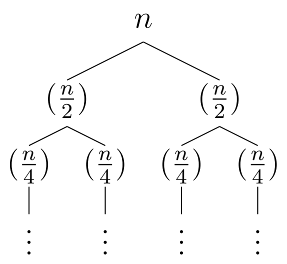
2. *(15 pts) Solve the following recurrence relations using the recurrence tree method; include a diagram of your recurrence tree. If the recurrence relation describes the behavior of an algorithm you know, state its name.*

   (a) $T(n) = T(\lceil n/2 \rceil) + 1$



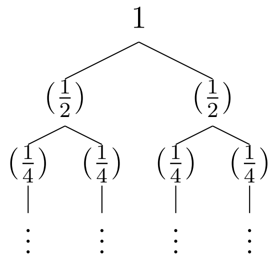
   This only has one recursive call for each recursion, and each stage does less and less. Thus the relation looks like $T(n) = 1 + \frac{1}{2} + \frac{1}{4} + ... = O(1) + O(1) + ... = \Theta(1)$

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



   Thus at each level we are performing $O(n)$ operations, and since it is a binary tree, there are $\lg n$ levels, thus the runtime is $O(n \lg n)$. This reminds us of the Merge Sort algorithm, which does half the n work with each level into the recursion, and not coincidentally also $= O(n \lg n)$.

   (c) $T(n) = 2T(n/2) + 1$



In this example we see that each level does $O(1)$ work. However, it is another binary tree, which means that there are $\lg n$ levels, so the runtime is just $O(\lg n)$

3. *(20 pts) Lemma 13.1 in CLRS states*

A red-black tree with n internal nodes has height at most $2\lg(n+1)$.

For a tree with n = 10, give a sequence of positive integers $\sigma = [\sigma_1, \sigma_2, ..., \sigma_n]$ for $\sigma_i \in [1, 10]$, such that inserting that sequence into an empty red-black tree achieves the maximum height. Justify your claim.

Typically in a binary search tree the worst-case input is a sorted list. So we tried that for the red-black tree as $\sigma = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]$. According to a red-black tree simulator[1] it gave us a length of 5 nodes + 1 null node (which is counted in the theorem) = 6 nodes. According to the formula, the length can be no longer than $2 * \lg(n+1)$, which evaluates to 6.92 at $n = 10$. Thus, $6 = floor(6.92)$, so that is the max height of a red-black tree starting from scratch with 10 nodes.

[1]Simulator found at https://www.cs.usfca.edu/ ~galles/visualization/RedBlack.html

4. *(30 pts) Professor McGonagall asks you to help her with some arrays that are peaked. A peaked array has the property that the subarray $A[1..i]$ has the property that $A[j] < A[j+1]$ for $1 \leq j < i$, and the subarray $A[i..n]$ has the property that $A[j] > A[j+1]$ for $i \leq j < n$. Using her wand, McGonagall writes the following peaked array on the board $A = [7, 8, 10, 15, 16, 23, 19, 17, 4, 1]$, as an example*

   (a) *Write a recursive algorithm that takes asymptotically sub-linear time to find the maximum element of A.*

```
findPeak(A, l, r){
    if l > r then return -1
    m = floor( (l+r)/2 )
    if m == 1 then m += 1
    if A[m-1] < A[m] < A[m+1] then return findPeak(A, m-1, r)
    if A[m-1] > A[m] > A[m+1] then return findPeak(A, l, m+1)
    else return A[1]
}
```

   (b) *Prove that your algorithm is correct. (Hint: prove that your algorithms correctness follows from the correctness of another correct algorithm we already know.)*

   We know that this will work because it is exactly the same algorithm as the Binary Search Algorithm[2]. The only difference is that instead of searching for an index with a certain associated value, we're searching for an index with a certain property. In this case the property is that the value is greater than the values to the left and right. If it is greater than the value to the left, but not the right, we go right to find the peak. If it is greater than the value to the right, but not the left, we go left. The only difference is that we have to check if m is 1 because of the floor() function, which would result in an error for $A[m-1]$. Note we do not need to check if it is A.length because the peak is strictly less than A.length.

   (c) *Now consider the multi-peaked generalization, in which the array contains k peaks, i.e., it contains k subarrays, each of which is itself a peaked array. Let $k = 2$ and prove that your algorithm fails on such an input.*

   Let's make the call $findPeak([1, 5, 4, 3, 2, 6, 5], 1, 7)$. In this case the algorithm will go straight to the middle element, 3, which is on a "downward slope", so the algorithm will search the left half of the list. At this point we already know that

[2]Algorithm borrowed from https://en.wikipedia.org/wiki/Binar_search_algorithm

the algorithm will fail because it will never check the right side of the list, which contains the max element, 6. (The algorithm will return 5, the largest element in the left side of the list)

(d) *(10 points extra credit) Suppose that k=2 and we can guarantee that neither peak is closer than n/4 positions to the middle of the array, and that the "joining point" of the two singly-peaked subarrays falls in the middle half of the array. Now write an algorithm that returns the maximum element of A in sublinear time. Prove that your algorithm is correct, give a recurrence relation for its running time, and solve for its asymptotic behavior.*

```
findLargestPeak(A){
    m = floor( (1+A.length)/2 )
            leftPeak = findPeak(A, 1, m)
            rightPeak = findPeak(A, m+1, A.length)
            if leftPeak >= rightPeak then return leftPeak
            else return rightPeak
}
```

Note that this algorithm divides the problem into two separate arrays, each of which contains one of the peaks. Thus since the last algorithm ran in $O(\log n)$ and this algorithm runs the original algorithm twice plus a contant, it runs in $O(2\log n + c) = \Theta(\log n)$.

5. *(30 pts total) 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 $n \times n$ 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 $A[i] + A[i+1] + \cdots + A[j]$. (The value of array element $B[i,j]$ is left unspecified whenever $i \geq j$, so it doesnt matter what the output is for these values.) Dumbledore suggests the following simple algorithm to solve this problem:*

```
dumbledoreSolve(A) {
    for i=1 to n
        for j = i+1 to n
            s = sum of array elements A[i] through A[j]
            B[i,j] = s
        end
    end
}
```

(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).*

In order to find an upper bound $O(f(n))$ for the running time of this algorithm, we first look at the running times of each line:

The first line (for i = 1 to n) has a $\Theta(n)$ running time because this line iterates n times (from 1 to n) and each iteration takes c_1 (some constant) time, so the running time will be $c_1 n = \Theta(n)$. The second for loop in line 2 runs $(n+1) - i$ times for each i, with running time c_2. The total time that this line contributes to $T(n)$ of the algorithm can be written as $\sum_{i=1}^{n-1} c_2(n-i) = c_2 \cdot \frac{n(n-1)}{2}$. Right after the second line, the elements of the array are summed up from $A[i]$ up to $A[j]$ for each iteration of j. This means that the algorithm uses a third for loop to do this summation. The running time of this loop can be represented as $\sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \sum_{k=i}^{j} c_3$. The last line gets executed the same number of times as the second line, and its running time is similar to it, with only a different running time constant: $c_4 \cdot \frac{n(n-1)}{2}$.

Adding up the running times for each line of the algorithm, we can write $T(n)$ as follows:

$$
\begin{aligned}
T(n) &= c_1 n + c_2 \cdot \frac{n(n-1)}{2} + \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \sum_{k=i}^{j} c_3 + c_4 \cdot \frac{n(n-1)}{2} \\
&\quad Simplifying \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \sum_{k=i}^{j} c_3 : \\
&\quad \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \sum_{k=i}^{j} c_3 = c_3 \cdot \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} (j+1-i) = c_3 \cdot \sum_{i=1}^{n-1} (\sum_{j=i+1}^{n} j + \sum_{j=i+1}^{n} 1 - \sum_{j=i+1}^{n} i) \\
&\quad = c_3 \cdot \sum_{i=1}^{n-1} (\frac{n(n+1)}{2} - \frac{(i+1)\cdot(i+2)}{2}) \\
&\quad = c_3 \cdot \sum_{i=1}^{n-1} (\frac{n^2+n}{2} - \frac{i^2+3i+2}{2} - \frac{2n-2i}{2} - \frac{2in-2i^2}{2}) \\
&\quad = \frac{1}{2} \cdot c_3 \sum_{i=1}^{n-1} (n^2 + +3n - 3i^2 - 5i - 2in - 2) \\
&\quad = \frac{1}{2} \cdot c_3 (\sum_{i=1}^{n-1} n^2 + \sum_{i=1}^{n-1} 3n - \sum_{i=1}^{n-1} 3i^2 - \sum_{i=1}^{n-1} 5i - \sum_{i=1}^{n-1} 2in - \sum_{i=1}^{n-1} 2) \\
&\quad \ldots \\
&\quad = \frac{1}{2} \cdot c_3 \cdot \frac{n^3 + 3n^2 - 7n + 6}{3} \\
&\quad = c_3 \cdot \frac{n^3 + 3n^2 - 7n + 6}{6} \\
T(n) &= c_1 n + c_2 \cdot \frac{n(n-1)}{2} + c_3 \cdot \frac{n^3 + 3n^2 - 7n + 6}{6} + c_4 \cdot \frac{n(n-1)}{2} \\
&= \frac{1}{6} \cdot (n^3 \cdot c_3 + 3n^2 \cdot (c_2 + c_3 + c_4) + n \cdot (-3c_2 + 6c_1 - 7c_3 - 3c_4) + 6c_3) \\
&\leq c \cdot n^3 \\
T(n) &= O(n^3)
\end{aligned}
$$

After we have simplified our function for the running time of our algorithm, we have shown that $c \cdot n^3 \geq (\frac{d \cdot n^3 + \ldots}{6})$ for a constant c (for example, when $c = 2$). This means that our algorithm's total running time will be bounded above by n^3 and therefore, it will be:

$T(n) = O(n^3)$

The idea for the triple-summation for the three nested loops was drawn from: http://math.stackexchange.com/questions/1519578/write-the-following-for-loop-as-a-double-summation

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

We can use our same $T(n)$ that we found in 5a) to find a lower bound for the running time of our algorithm. Our $T(n)$ was:

$T(n) = \frac{1}{6} \cdot (n^3 \cdot c_3 + 3n^2 \cdot (c_2 + c_3 + c_4) + n \cdot (-3c_2 + 6c_1 - 7c_3 - 3c_4) + 6c_3)$

We can further simplify this function and write:

$$\begin{aligned} T(n) &= \frac{1}{6} \cdot (n^3 \cdot c + 3n^2 \cdot a + nb + 6d) && (1) \\ &\geq \frac{1}{6} n^3 && (2) \\ T(n) &= \Omega(n^3) && (3) \\ & && (4) \end{aligned}$$

(c) *Although Dumbledores algorithm is a natural way to solve the problem - after 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.*

An algorithm that solves the problem in $O(f(n)/n)$ time is:

```
dumbledoreSolve(A) {
    for i=1 to n
```

```
        s = A[i]
        for j = i+1 to n
            s = s + A[j]
            B[i,j] = s
        end
    end
}
```

Proving the algorithm:

If we look at this new algorithm, we can see that it is a little different than Dumbledore's original algorithm. Instead of summing up each element $A[i] + A[i+1] + A[i+2] + ... + A[j]$ for each j, the new algorithm declares a variable named "s" right below the first for loop and sets the value of it to $A[i]$. Then for each $j = i+1$ to $j = n$, it adds on to s the value of $A[j]$ for that particular j and sets $B[i.j]$ to s. This way it's keeping track of the sum from $A[i]$ upto $A[j-1]$ when it's trying to compute the sum from $A[i]$ to $A[j]$ instead of repeating the same additions over and over again.

The running time of the first line of this algorithm is the same: $c_1 n = \Theta(n)$. The second line takes gets executed the same number of times as the first line. The second for loop also has a running time that is equal to its running time in the original algorithm since we didn't change anything about it: $c_3 \cdot \frac{n(n-1)}{2}$. The last two lines get called the same number of times as the for loop statement, but with different constants. We can represent $T(n)$ as follows:

$$\begin{aligned}
T(n) &= c_1 n + c_2 n + c_3 \cdot \frac{n(n-1)}{2} + c_4 \cdot \frac{n(n-1)}{2} c_5 \cdot \frac{n(n-1)}{2} \\
&= n(c_1 + c_2) + \frac{n(n-1)}{2}(c_3 + c_4 + c_5) \\
&= n(c) + \frac{n(n-1)}{2}(d) \\
T(n) &= \Theta(n^2)
\end{aligned}$$

We have shown that the $T(n)$ for our new algorithm is actually equal to $\Theta(n^2)$, which is $O(T(n)/n)$ faster. And since it's $\Theta(n^2)$, it's also $O(n^2)$.

6. *(20 points extra credit) 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 $O(n^2)$ time).*
Hagrid says his algorithm works like this:

   - Divide the array A into left and right halves
   - Recursively find the largest subarray sum for the left half
   - Recursively find the largest subarray sum for the right half
   - Find largest subarray sum for a subarray that spans between the left and right halves
   - Return the largest of these three answers

   On the chalkboard, which appears out of nowhere in a gentle puff of smoke, Hagrid writes the following pseudocode for his algorithm:

```
hagridSolve(A) {
    if(A.length()==0) { return 0 }
    return hagHelp(A,1,A.length())
}

hagHelp(A, s, t) {
    if (s > t) { return 0 }
    if (s == t) { return max(0, A[s]) }

    m = (s + t) / 2

    leftMax = sum = 0
    for (i = m, i > s, i--) {
            sum += A[i]
            if (sum >= leftMax) { leftMax = sum }
    }

    rightMax = sum = 0
    for (i = m, i <= t, i++) {
            sum += A[i]
```

```
                if (sum > rightMax) { rightMax = sum }
        }

        spanMax = leftMax + rightMax
        halfMax = max( hagHelp(s, m), hagHelp(m+1, t) )
        return max(spanMax, halfMax)
    }
```

Hagrid claims that his algorithm is correct, but Dumbledore says "tut tut."
(i) Identify and fix the errors in Hagrid's code, (ii) prove that the corrected algorithm works, (iii) give the recurrence relation for its running time, and (iv) solve for its asymptotic behavior.

(i)

There are three errors in Hagrid's code: first of all, the i > s in the for loop for the left subarray isn't correct. It should be i $\geq$ s to include $A[s]$. The next error is the sum $\geq$ leftMax in the for loop for the left subarray. This should be changed to sum > leftMax because it is unnecessary to assign the value of sum to leftMax when they're already equal to each other. The last error in the algorithm is the i = m in the for loop for the right subarray. This should be changed to i = m + 1 because $A[m]$ is already used in the process of solving for the left subarray.

(ii)

Proof of algorithm:

The case where $n = 0$: If n is zero, which means that our array doesn't have any elements in it. In this case, we return 0 from the hagridSolve() function. This is correct because if you don't have any elements, your maximum is 0.

The case where $n = 1$: If $n = 1$, we either return $A[s]$ or 0 (if the element is negative) from the hagHelp function. This is also correct.

The case where $n > 1$: When $n > 1$, we actually start using our recursive method. First we separate our problem into two halves and and we find the max subarray which spans both the left half and the right half. We do this by going from m down to s, and incrementing leftMax as we find values that will make our sum larger if we add them to our current sum. We do the same thing for the right half. After that, we add the leftMax and the rightMax and find the max subarray spanning both the left and right halves. Then, we make two recursive calls: we create two new problems that are half of the size of our original problem. We do the same things mentioned above until we are down to the trivial case, in which $n = 1$. Then we combine all of the trivial cases until we go all the way back to the top level of our recursion. At that point we have found our max subarray for both the left and the right side. Then we compare our 3 subarrays(left, right, and spanning), and return the maximum one, which makes sense because the max subarray of our array is either between s and m, $m+1$ and t, or it spans both halves.

(iii)

The recurrence relation for this algorithm can be given as:

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

because our problem is being split into two halves at each level of recursion and finding leftMax and rightMax take $\Theta(n)$ in total since we are going from m down to s and from from $m+1$ to t (n steps in total, with each step having $\Theta(1)$ running time). All the other lines take $\Theta(1)$ time.

(iv)

Each level i of the tree has 2^i nodes and each node at level i has size $\frac{n}{2^i}$. The recurrence tree has height $i = \log_2 n$. Using this information, each level will have a running time of $c \cdot 2^i \frac{n}{2^i} = cn$ That means the total running time will take $cn \cdot \log_2 n = \Theta(nlog_2 n)$ time.