CSCI 3104 Spring 2018 Problem Set 3
Bailon, Erika (09/28)

Problem 1

(5 pts total) For parts (1a) and (1b), justify your answers in terms of deterministic QuickSort, and for part (1c), refer to Randomized QuickSort. In both cases, refer to the versions of the algorithms given in Lecture 3.

(a) *What is the asymptotic running time of QuickSort when every element of the input A is identical, i.e., for $i \leq i; j \leq n$, $A[i] = A[j]$?*

If all the elements in the array are the same value, then it means that we are in a worst case scenario for QuickSort because it won't matter what pivot is picked, it is the same number in the entire array, so we would go through all the array swapping the elements because of the if statement. We already saw in class that the asymptotic running time of QuickSort in the worst case scenario is $\boldsymbol{\Theta}(\mathbf{n^2})$.

(b) *Let the input array A = [9, 7, 5, 11, 12, 2, 14, 3, 10, 6]. What is the number of times a comparison is made to the element with value 3?*

The total number of comparison is **4**. We obtain this number because:

First our pivot is 6, our $i = 0$ and out $j = 1$, we start comparing each term of the array to see if they're smaller than our pivot, which is 6.
9 and 7 are not smaller so j keeps increasing and we do not swap anything.
The we get to 5 which is smaller than 6 so we increment i and then swap with j.
We keep doing this until we get with j to the number 3.
We then have our **first** comparison, but keep moving till the end of the array, then we do the partition. We are going to end up with the array [5 2 3].
Now our pivot is 3!!!! Therefore we have **second** and **third** comparisons because we compare 3 with 5 and then we compare it with 2.
We then go through a partition again and we end up with the array [2 3].
This is when we do our last and **fourth** comparison.

(c) *How many calls are made to **`random-int`** in (i) the worst case and (ii) the best case? Give your answers in asymptotic notation.*

(**i**) For the worst case scenario we will consider the case where the array has n elements, therefore there will be n pivots so there will be n calls to $random - int$. This gives us $n-1$ calls to $partition$ so we get the recurrence $T(n) = T(n-1) + \Theta(1)$. Therefore the worst case scenario is $\mathbf{\Theta(n)}$.
(**ii**) For the best case scenario the partition will divide in subproblems of size $\frac{n}{2}$. Therefore the recurrence is $T(n) = 2T(\frac{n}{2}) + \Theta(1)$. Therefore the best case scenario is $\mathbf{\Theta(n)}$.
$\Rightarrow$ The number of calls made in the `random-int` is both cases which is $\mathbf{\Theta(n)}$.

Problem 2

*(30 pts total) Professor Trelawney has acquired **n** enchanted crystal balls, of dubious origin and dubious reliability. Trelawney needs your help to identify which crystal balls are accurate and which are inaccurate. She has constructed a strange contraption that fits over two crystal balls at a time to perform a test. When the contraption is activated, each crystal ball glows one of two colors depending on whether the **other** crystal ball is accurate or not. An accurate crystal ball always glows correctly according to whether the other crystal ball is accurate or not, but the glow of an inaccurate crystal ball cannot be trusted. You quickly notice that there are four possible test outcomes:*

crystal ball i glows	crystal ball j glows		
red	red	$\Rightarrow$	at least one is inaccurate
red	green	$\Rightarrow$	at least one is inaccurate
green	red	$\Rightarrow$	at least one is inaccurate
green	green	$\Rightarrow$	both accurate, or both inaccurate

(a) *Prove that if $n/2$ or more crystal balls are inaccurate, Trelawney cannot necessarily determine which crystal balls are accurate using any strategy based on this kind of pairwise test. Assume a worst-case scenario in which the inaccurate crystal balls contain malicious spirits that collectively conspire to fool Trelawney.*

The professor Trelawney would not be able to determine which crystal balls are accurate and which ones are inaccurate due the fact that we have two sets of crystal balls, but one set is inaccurate, which can cause both groups to be symmetric and this will affect the operations of pairwise comparison, making it impossible to determine which set of balls are accurate or which set of balls are inaccurate. I got to this conclusion because if we assume that we have a set of crystal balls with n balls, such that x of them are accurate and $n-x$ are inaccurate, and $n-x \geq x$, this guarantees that we would be able to find two sets of crystal balls that are of the same size and different accuracy. Even if we pair all the balls to either be two red, or two green, and get rid of them with certain algorithm, we are going to end up with a number of crystal balls that at the time of the test, the comparison would not be accurate because we are still going to have malicious balls that behave depending on the other ball. So again, the pairwise comparison is not accurate **therefore** there is no way to determine which balls are accurate and which ones are inaccurate.

(b) *Suppose Trelawney knows that more than $n/2$ of the crystal balls are accurate, but not which ones. Prove that $\lfloor n/2 \rfloor$ pairwise tests are sufficient to reduce the problem to one of nearly half the size.*

Knowing that more than half of the crystal balls are accurate will help us find the accurate balls either n is even, or n is odd.
For the first case, let's assume that n is even. If n is even we will have 2 sets of evenly distributed crystal balls. We first run the test on one set comparing the balls by pairs. We are going to throw away all the pairs of balls that are NOT $Green-Green$ and from those pair of balls that are $Green-Green$ we are going to throw away ONE ball of the two. This process will give us more accurate balls than inaccurate balls because we throw away AT LEAST one inaccurate ball from all the pairs that contained AT LEAST one inaccurate ball. Now we have more than half chance of keeping an accurate ball from the $Green-Green$ pair. We repeat the process, and after we recursively run the same process we will be able to determine which balls are accurate.
In the second case, if n is an odd number, we are going to test the $floor$ of $\frac{n}{2}$ and do the same test we did when we had an even number. However, we are going to hold on to one of the crystal balls. After the test we are going to have certain number of $Green-Green$ crystal balls, let's call it $2b$ PLUS that one crystal ball we kept. This will imply that we have at least $2b+1$ accurate balls.

(c) *Now, under the same assumptions as part (2b), prove that all of the accurate crystal balls can be identified with $\Theta(n)$ pairwise tests. Give and solve the recurrence that describes the number of tests.*

Under the assumption that at least $\frac{n}{2}$ crystal balls are accurate and the solution from 2.b is correct, we can find one accurate ball. Once we have that accurate ball we can start the test by pairing that accurate crystal ball with all the other ones. Since we know which ball is the accurate ball and that from each par we have at least one accurate ball, we can determine if the ball we are testing it with, is accurate or not.This test will win up to $n-1$ which gives us a recurrence of $T(n) = n$ and from part be we have $T(n) = T(\lceil \frac{n}{2} \rceil) + \lfloor \frac{n}{2} \rfloor$. However $T(n) \leq T(\lceil \frac{n}{2} \rceil) + \lfloor \frac{n}{2} \rfloor \Rightarrow$ The solution for this can be found in $\boldsymbol{\Theta}(\mathbf{n})$

Problem 3

(20 pts) Professor Dumbledore needs your help. He gives you an array A consisting of n integers $A[1], A[2] \dots, A[n]$ 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., $B[i,j] = A[i] + A[i+1] + \cdots + A[j]$. (The value of array element $B[i,j]$ is left unspecified whenever $i \geq j$, so it doesn't 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(A[i..j]) // look very closely here
      B[i,j] = s
}}}
```

(a) *For some function g that you should choose, give a bound of the form $\Omega(g(n))$ 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).*

To find the best case scenario we look at the base where we have one element. We see that the first loop goes through n times. Then the following loop has the form to go through it $(n-1) + (n-2) + ... + 1 + 0$. From class we recognize this type of summation to which solution is $\frac{n(n-1)}{2}$. Assuming we have the best case scenario the next loop will be an atomic operation. Therefore we get:
$\mathbf{T(n) = n \cdot \frac{n(n-1)}{2} = \Omega(n^3)}$

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

For the worst case scenario we assume we have an array with $n > 1$. Therefore we go through the first loop at most n times. Then the second for loop we also go at most n times. Then we have the summation that has to go also at most n times. The other operation takes a constant time. After we have the times we go through each we can see that we have a recurrence of the form:
$T(n) = n \cdot n \cdot n + c$ which is $\mathbf{O(n^3)}$.

(c) *Although Dumbledore's 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 ineffciency. Give an algorithm that solves this* *problem in time $\mathcal{O}(g(n)/n)$ (asymptotically faster) and prove its correctness.*

```
severusSnape
{
   sumSoFar = 0;
   for i=1 to n
   {
        sumSoFar = A[i]
        for j=i+1 to n
        {
             sumSoFar +=  A[j]
             B[i,j] = sumSoFar
        }
    }
 }
```

With this algorithm we have two $forloops$ and one atomic operation that will take constant time, therefore we have $T(n) = n \cdot (n + C)$. Let's set this recurrence to $g(n)$.

To find the asymptotic relation we have $\lim_{n\to\infty} \frac{n^3}{n^2} = \frac{n}{1} = \infty \Rightarrow \Omega(g(n))$ which means that $g(n)$ is asymptotically faster.

To prove our algorithm we find the loop invariant. At the start of the *ith* iteration, for all $j < i$ it is true that $A[j]? = i$

Initialization: We need n to be at least 2 to be able to prove with the loop invariant. We are going to enter the function and initialize our summation variable to 0. We enter the first loop with $i = 1$ and we set our summation to the first spot in the array which is $A[i]$, then we enter our second loop setting the location of j to the second element in the array. We will then do the addition and position the addition in the first entry of the matrix created. We repeat until the end and we will get out of the *loops* with a matrix made.

Maintenance: When we have more than 2 elements in the array, we are going to start with i being the first element in our array and we are going to go trough the array doing the additions and inserting the summation in the matrix B. We will repeat the same addition process until we finish adding the array. The algorithm will work because we are increasing by 1 and changing the spot of $B[i.j]$ with each iteration.

Termination: The algorithm will only stop when we have rech the end of our array after we have added the spots to create the matrix B.

Problem 4

(15 pts 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 $\mathcal{O}(n \log n)$ time (compared to applying a linear search to the B matrix, which would take $\mathcal{O}(n^2)$ 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 A[s] }

  m = (s + t) / 2

  leftMax = sum = 0

  for (i = m-1, i >= s, i--) {
    sum += A[i]
    if (sum > leftMax) { leftMax = sum }
  }
  rightMax = sum = 0

  for (i = m+1, i <= t, i++) {
    sum += A[i]
    if (sum > rightMax) { rightMax = sum }
  }

  spanMax = leftMax + rightMax
  halfMax = max( hagHelp(s, m-1), 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.

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

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

   m = floor (s + t) / 2 **First mistake, they didn't
         have floor so we wouldn't be sure of which value was grabbing**

   leftMax = sum = 0
   for (i = m, i >= s, i--) {     *** Here I found the second mistake,
              they had "m-1" and I changed it to "m" so it can also
              take in consideration the "m" value in the array***
       sum += A[i]
       if (sum > leftMax) { leftMax = sum }
   }

   rightMax = sum = 0
   for (i = m+1, i <=t, i++) {
       sum += A[i]
       if (sum > rightMax) { rightMax = sum }
   }
   spanMax = leftMax + rightMax
   halfMax = max( hagHelp(A,s, m), hagHelp(A, m+1, t) ) *** Here was the other
               mistake they had again "m-1" so changed to "m" and also
               pass the array in the function because they didn't
               have it pass before ***
   return max(spanMax, halfMax)
}
```

(ii) **Initialization**: We can have a base case of an array with $n = 0$ elements. We will get into the algorithms and return a 0 because of the first if statement, which means that the array was empty. For $n = 1$ we will return the only element in the array with the second if statement. When we have 3 elements in the array we will enter the operation that will divide the array with the floor value. Then we will enter the first for loop to obtain the sum of the elements in the right side of the array. Same-wise we find the sum on the left. The next step we add those value to make sure we span all values and then we call the function again on the right side and the left side. The function will find again a sum and then compare which one is the maximum value. After that we will compare that maximum value we found from the subarrays and compare it with the value we obtain from the part of the array that contained m. Therefore we will get the maximum value overall the additions in the array.
Maintenance: For an array with n element greater than 3, the function ill split the array into 2 each time we have enough values to keep dividing it until we find singles additions to be able to compare with the other additions on our way back to the additions of the entire array. It will ten find the subarray with the maximum value of the addition and it will return it.
Termination: At the end of the loops we have must found a maximum number when we compare all the additions we did with the subarrays. The function will get to the point where there cannot be more divisions and it will compare and return the larger value of the additions.

(iii) and (iv) The recurrence relation for this algorithms is $T(n) = 2T(\frac{n}{2}) + \Theta(n)$ and by the Master's Theorem we get that the complexity of this algorithms is $\mathbf{O(nlogn)}$.

[1]

[1]Worked with: Eric Oropeza, Selena Quintanilla and office hours