CSCI 3104, Fall 2017 Problem Set 3
Adam Ten Hoeve (09/25)
adam.tenhoeve@colorado.edu
Coding Language: C++

Problem 1

(30 pts) 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:

Widget A says	Widget B says	Conclusion
B is good	A is good	both are good, or both are bad
B is good	A is bad	at least one is bad
B is bad	A is good	at least one is bad
B is bad	A is bad	at least one is bad

(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.*

SOLUTION: The worst case would be that the bad widgets would always lie. Assume n/2 widgets are good and n/2 are bad. There are two possible cases for the results of widget testing.

1. If both say the other is good, either both are good or both are bad.
2. If both say that the other is bad, then one is good and one is bad.

The easiest way to narrow down the widgets is to remove them based on the response given. If both say bad, then we can determine that the test is inconclusive and both can be removed. If both say good then we know that they

are the same kind, whether bad or good, and can remove one of the widgets. By repeating this process, we can approach a base case where there are three widgets where 2 are bad and one is good, or all three are bad. Because of this, there is no way to tell which are good.

(B) *(Divide) 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 floor$(n/2)$ pairwise tests are sufficient to reduce the problem to one of nearly half the size.*

SOLUTION: Using the same method as above, we can reduce a problem of size n to one of at most n/2. Assuming that all the comparisons result in "good, good" then there will be half the original set size because one gets removed on each occurrence. If any comparisons result in "bad, bad" then both get removed so the set size decreases even faster. Because of this, it would take floor(n/2) pairing to reduce the problem to one of at most half the size.

(C) *(And Conquer!) Prove that the good widgets can be identified with $\Theta(n)$ pairwise tests, assuming that more than $n/2$ of the widgets are good. Give and solve the recurrence that describes the number of tests.*

SOLUTION: Using the logic above and assuming there are more good widgets than bad, we can reduce the problem down to one good widget. Each iteration, the problem size is being cut in half and the cost is being cut in half. Therefor our recurrence relation looks like:

$$T(n) = T(\frac{n}{2}) + \frac{n}{2}$$

Using the master method, we can see that a=1, b=2, and c=1. We can find that the root dominates because $log_2 1 = \frac{1}{2} < 1 = c$, therefor $T(n) = \Theta(\frac{n}{2})$. Dropping the coefficients, we find that the function has a runtime of $T(n) = \Theta(n)$.

Problem 2

(25 pts) Professor Dumbledore needs your help. He gives you an array A consisting of n integers $A[1]$, $A[2]$, $\cdots$, $A[n]$ 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 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 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 $\mathcal{O}(f(n))$ 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).*

SOLUTION: There are three loops in the pseudo-code. Two from the if statements and one from the summation from i to j. These loops can be expressed as a series of summations.

$$T(n) = \sum_{i=1}^{n}(\sum_{j=i+1)}^{n} (\sum_{k=i}^{j} 1))$$

These summations can be converted to equations using math and Wolfram Alpha.

$$T(n) = \sum_{i=1}^{n}(\sum_{j=i+1)}^{n} (-i + j + 1))$$

$$T(n) = \sum_{i=1}^{n}(n(-i + \frac{n}{2} + \frac{3}{2}) + (\frac{i}{2} - \frac{3}{2})i)$$

$$T(n) = \frac{1}{6}(n^3 + 3n^2 - 4n)$$

$$T(n) = \Theta(n^3)$$

So the upper bound on the function would be $\mathcal{O}(n^3)$.

(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.)*

SOLUTION: As proven above, the function is $\Theta(n^3)$. Therefor, the algorithm is also $\Omega(n^3)$.

(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 $\mathcal{O}(f(n)/n)$ (asymptotically faster) and prove its correctness.*

SOLUTION: In order to reduce the runtime from $\mathcal{O}(n^3)$ to $\mathcal{O}(\frac{n^3}{n}) = \mathcal{O}(n^2)$, we must remove a loop from the algorithm. The Easiest way to do this is to remove the summation loop.

```
studentSolve(A)
    for i=1 to n
        sum = A[i]
        for j=i+1 to n
            sum = sum + A[j]
            B[i,j] = sum
```

Assuming Dumbledore's solution is correct, we can show that our solution is also correct by proving both algorithms do the same thing. Both algorithms use the same initial loop to iterate through A and get values. They both use a nested for loop to iterate through the rest of the loop from the i^{th} position. The only difference between the algorithms is that Dumbledore's sums every element between i and j every time j iterates, whereas ours stores the sum in a variable and adds the j^{th} element every every time it iterates, removing the need for the extra loop. Therefor, because Dumbledore's solution is correct and our improved algorithm is the same, our algorithm is also correct.

Problem 3

(20 pts) 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:

```
1.  hagridSolve(A) {
2.     if(A.length()==0) { return 0 }
3.     return hagHelp(A, 1, A.length())
4.  }
5.
6.
7.  hagHelp(A, s, t) {
8.     if(s > t) { return 0 }
9.     if (s == t) { return max(0, A[s]) }
10.    m = (s + t) / 2
11.    leftMax = sum = 0
12.    for (i = m, i > s, i--) {
13.        sum += A[i]
14.        if (sum >= leftMax) { leftMax = sum }
15.    }
16.    rightMax = sum = 0
17.    for (i = m, i <= t, i++) {
18.        sum += A[i]
19.        if (sum > rightMax) { rightMax = sum }
20.    }
21.    spanMax = leftMax + rightMax
22.    halfMax = max( hagHelp(s, m), hagHelp(m+1, t) )
23.    return max(spanMax, halfMax)
24. }
```

Hagrid claims that his algorithm is correct, but Dumbledore says tut tut.

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

SOLUTION: There are mucho errors in Hagrid's Code:

1. On line 9, do not return the max of 0 or A[s] because if A[s] is negative, it will return 0. We just want to return A[s].
2. On line 11, leftMax = sum = A[m].
3. On line 12, i = m-1 and i¿=s.
4. On line 16, rightMax = sum = A[m+1]
5. On line 17, i=m+2.
6. On line 22, hagHelp needs to pass in A into the recursive functions.

```
1.  hagridSolve(A) {
2.     if(A.length()==0) { return 0 }
3.     return hagHelp(A, 1, A.length())
4.  }
5.
6.
7.  hagHelp(A, s, t) {
8.     if(s > t) { return 0 }
9.     if (s == t) { return max(A[s])}
10.    m = (s + t) / 2
11.    leftMax = sum = A[m]
12.    for (i = m-1, i >= s, i--) {
13.        sum += A[i]
14.        if (sum >= leftMax) { leftMax = sum }
15.    }
16.    rightMax = sum = A[m+1]
17.    for (i = m+2, i <= t, i++) {
18.        sum += A[i]
19.        if (sum > rightMax) { rightMax = sum }
20.    }
21.    spanMax = leftMax + rightMax
22.    halfMax = max( hagHelp(A, s, m), hagHelp(A, m+1, t) )
23.    return max(spanMax, halfMax)
24. }
```

(ii) *Prove that the corrected algorithm works*

SOLUTION:
Base Case:

(a) A.length = 0, there is no array and no maximum sum. The function returns 0, intended behavior.

(b) A.length = 1, A[0] is the only value, function returns A[0], intended behavior.

Refence pg 70 in textbook. Any subarray A[i..j] of A[s...t] must lie in one of the following places:

- Entirely in the subarray A[s...m] so that $s \leq i \leq j \leq m$.
- Entirely in the subarray A[m+1...t] so that $m + 1 \leq i \leq j \leq t$.
- The subarray crosses the midpoint so that $s \leq i \leq m < j \leq t$.

Therefor a maximum subarray of A[s...t] must lie within one of these places. We can find maximum subarrays of A(s...m) and A(m+1...t) recursively, because these two subproblems are smaller instances of the problem of finding a maximum subarray. All that is left is to find a maximum subarray that crosses the midpoint and find the maximum of the three. In order to keep the runtime of each subproblem to half the size, we split the array from case 3 into two subarrays: an array with elements left of the midpoint and an array of elements right of the midpoint. Then we can find the max of each subarray and add them together to find the total max of the array. The left subarray must include the mid point so leftMax is set to the midpoint. In order to compute the sums, the for loop on line 12 starts at mid-1 and increments down to s, or the left most index. The sum from that subarray is then saved, and if it is greater than leftMax, we update leftMax. The same process is done to the right subarray. In order to make sure the midpoint is not counted twice, we start the second subarray at m+1 and the loop iterates from m+2 to t, or the right most index. spanMax then adds leftMax and rightMax to account for case 3, where the subarray spans across the midpoint. All possible cases for the sum are accounted for, therefor the algorithm is correct.

(iii) *Give the recurrence relation for its running time*

Line 12 is a for loop to loop over the left subarray and Line 16 is a loop to iterate over the right subarray. Each of theses subarrays are half the size o the original array so we spend $T(\frac{n}{2})$ time on each subproblem. There are two subarrays which are each half the size of the origional problem, so the total runtime is around $2T(\frac{n}{2})$. For the base case, the runtime is constant so $T(n) = \Theta(1)$. This is when there is only one element in the subarray.

$$T(n) = \begin{cases} \Theta(1) & n = 1 \\ 2 \cdot T(\frac{n}{2}) + \Theta(n) & n > 1 \end{cases}$$

(iv) *Solve for its asymptotic behavior*

Using the master method, we can see that a=2, b=2, and c=1. From here, we can find that $log_b a = 1 = c$. This means that the work is the same at every recursive level of the algorithm. Therefor, The asymptotic solution is $T(n) = \Theta(nlog(n))$.

Reference: *Introduction to Algorithms* Chapter 4.1 The Maximum Subarray Problem

Problem 4

(10 pts) 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.*

SOLUTION: This problem can be solved by putting the worst case and best case into the same recurrence relation. When the worst case is performed, the array of size n is split into an array of n-1 and an empty array, with an end result of an array of n-1. The recurrence relation for this is $T(n) = T(n-1) + \Theta(n)$. For the best case, the array is split into two subarrays each of size n/2 and the pivot value is removed. The recurrence relation is $T(n) = 2T(\frac{n}{2}) + \Theta(n)$. Because the cases alternate, we can plug both recurrence relations into a total relation for the function. This come out to:

$$T(n) = 2(\frac{n-2}{2}) + \Theta(n)$$

Using the master method, we have a=2, b=2, and c=1 and can find that $log_b a = 1 = c$ so work is the same at every level and the asymptotic solution is $T(n) = \Theta(nlogn)$. Overall, this partition does not change the running time of QuickSort because that is also $T(n) = \Theta(nlogn)$ in the average case.