1. (30 pts total) *Professor Snape has $n$ magical widgets that are supposedly both identical and capable of testing each other's correctness. Snape's 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.*

   Let $n$ be odd, and let $m_B = (n+1)/2$ count the number of bad widgets and $m_G = n - m_B$ count the good widgets. Let $B$ denote the set of all bad widgets, and $G$ denote the set of all good widgets.

   There are three types of comparisons we can make: intra-$B$ comparisons, intra-$G$ comparisons, and $GB$ comparisons; suppose we make them all. At worst, each of the intra-$B$ comparisons will yield a good-good result because those widgets conspire to lie. Similarly, each of the intra-$G$ comparisons will yield a good-good result because those widgets tell the truth. However, each of the $GB$ comparisons yields a bad-bad result, because each widget in $G$ tells the truth (the other widget is bad) and each widget in $B$ lies (the other widget is good). Thus, each widget in $B$ has $m_B$ votes for "good" and $m_G$ votes for "bad," while each widget in $G$ has the reverse. So long as $m_B \geq m_G$, the widgets in $B$ can conspire to falsely label the widgets in $G$ as bad, no matter how Snape compares widgets. That is, Snape can know that all of the widgets in either $G$ or $B$ are bad, but because the comparison results are all symmetric with respect to the labeling of $G$ and $B$, Snape cannot correctly tell that the labels are backwards. In contrast, if $m_B < m_G$, then the good widgets can force the labeling of widgets to be correct.

(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 $\lfloor n/2 \rfloor$ pairwise tests are sufficient to reduce the problem to one of nearly half the size.*

The last sentence is a hint that Snape should take the $n$ widgets, group them into pairs, conduct those pairwise tests and then throw away nearly half the pairs according to some rule on the results of the pairwise comparisons. So long as Snape does this in a way that ensures that a majority of the widgets that remain are good, then Snape can apply this strategy recursively to find a good widget. That is, Snape should use a divide and conquer approach. Let $c_i$ denote the $i$th widget, and let Snape compare $c_{2i}, c_{2i+1}$ for $i = 1...\lfloor n/2 \rfloor$ (the actual comparison structure doesn't matter since the ordering of the widgets can be arbitrary).

If the result of a comparison is anything but good-good, then one or both of the widgets is bad. If we discard all such pairs of non-good-good widgets, then we reduce the size of the current version of the problem without changing the majority-good property. (If the comparison is good-good, then either both widgets are good or both widgets are bad.) That is, we throw away some good widgets when we do this, but at worst we throw away one good widget for every bad widget. Finally, for all the pairs of widgets that remain, all of which reported good-good, we know that they are either both good or both bad, and discarding one widget from each pair suffices to reduces the problem by at least half while still preserving the majority good property.

(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.*

The algorithm is to first run the process described in (b) to find a single good widget $c_G$ and then test each of the remaining $n-1$ widgets to determine if they are good or bad; because $c_G$ is good, we can always trust its results.

The running time of this algorithm is $T(n) = T(n/2) + n/2$ for the first part and $\Theta(n)$ for the second part. One way to solve for the asymptotic form of $T(n)$ is to

recognize that

$$\begin{aligned} T(n) &= \sum_{i=1}^{\log_2 n} n/2^i \\ &= n2^{-\log_2 n}\left(2^{\log_2 n} - 1\right) \\ &= n - 1 \\ &= \Theta(n) \end{aligned}$$

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

   Claim: $f = n^3$, and thus the running time of Dumbledore's algorithm is $O(n^3)$.

   Proof 1: The cost of the first line inside the inner loop is proportional to $j - i + 1$, which is the number of elements in the subarray $A[i]$ through $A[j]$, and the cost of the second line is a constant. The inner loop thus costs

$$\text{inner loop cost} = \sum_{j=i+1}^{n} j - i + 1 \ .$$

Furthermore, this cost is paid once for each pass of the outer loop. Thus, the cost of the entire algorithm is

$$\begin{aligned}\text{total cost} &= \sum_{i=1}^{n}\sum_{j=i+1}^{n} j-i+1 \\ &< \sum_{i=1}^{n}\sum_{j=1}^{n} j-i+1 \qquad < \sum_{i=1}^{n}\sum_{j=1}^{n} j \\ &= O(n^3)\ .\end{aligned}$$

□

Proof 2: Starting with our expression for the total cost of the algorithm, we may calculate the exact cost, as a function of $n$:

$$\begin{aligned}\text{total cost} &= \sum_{i=1}^{n}\left(\sum_{j=i+1}^{n} j-i+1\right) \\ &= \frac{1}{2}\sum_{i=1}^{n}(i-n-3)(i-n) \\ &= \frac{n^3}{6}+\frac{n^2}{2}-\frac{2n}{3} \\ &= O(n^3)\ ,\end{aligned}$$

for some constant $c < 1/4$ and $n_0 = 2$. □

(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.)*

Claim: The running time of Dumbledore's algorithm is also $\Omega(n^3)$.

Proof: Starting with our expression for the total cost of the algorithm, we may

calculate the exact cost, as a function of $n$:

$$\begin{aligned}\text{total cost} &= \sum_{i=1}^{n}\left(\sum_{j=i+1}^{n} j-i+1\right)\\ &= \frac{1}{2}\sum_{i=1}^{n}(i-n-3)(i-n)\\ &= \frac{n^3}{6}+\frac{n^2}{2}-\frac{2n}{3}\\ &= \Omega(n^3)\ ,\end{aligned}$$

for some constant $c > 1/5$ and $n_0 = 2$. □

(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 inefficiency. Give an algorithm that solves this problem in time $O(f(n)/n)$ (asymptotically faster) and prove its correctness.*

To understand how to produce a more efficient algorithm, we must identify the source of the inefficiencies.

The key observation is that the sum of elements $A[i]$ to $A[k]$, for $i < k$, contains the sum of elements $A[i]$ to $A[j]$, for $i < j < k$. That is, let $s_k$ denote the sum of the elements $i$ through $k$. The sum of the elements $i$ through $k+1$ is simply $s_k + A[k+1] = B[i,k] + A[k+1]$. Thus, we can save a lot of time by using the previously computed sums, which are stored in $B$, to compute the new sums, rather than summing them up afresh each time.

A more efficient algorithm is thus:

```
for i = 1 to n
   for j = i+1 to n
      if j == i+1
         B[i,j] = A[i]+A[j]
      else
         B[i,j] = B[i,j-1]+A[j]
      end
   end
end
```

Because the cost of the innermost statements is now constant, the algorithm's running time is

$$\text{total cost} = \sum_{i=1}^{n} \sum_{j=i+1}^{n} 1 = \Theta(n^2) \ ,$$

which is asymptotically faster than Dumbledore's algorithm.

Furthermore, we claim that this algorithm computes returns the same matrix $B$. We will prove this by proving that the state of $B$ is exactly the same in this algorithm, after each pass through the inner loop, as in Dumbledore's algorithm.

Here is the proof. When $j = i+1$ in Dumbledore's algorithm, $B[i, j]$ is set to be the sum of elements $A[i]$ and $A[j] = A[i + 1]$. When $j = i + 1$ in our new algorithm, the first branch of the `if` block is triggered, setting $B[i, j] = A[i] + A[j] == A[i] + A[i + 1]$. Thus, in this case, the state of $B$ is the same.

When $j > i+1$ in Dumbledore's algorithm, $B[i, j]$ is set to be the sum $\sum_{k=i}^{j} A[k] = A[i] + A[i + 1] + \cdots + A[j]$. In our new algorithm, this case triggers the second branch of the `if` block, setting

$$\begin{aligned} B[i, j] &= B[i, j - 1] + A[j] \\ &= B[i, j - 2] + A[j - 1] + A[j] \\ &= A[i] + A[i + 1] + \cdots + A[j] \qquad \text{for } j > i \ , \end{aligned}$$

where we have recursively applied the definition of $B[i, j]$. Thus, in this case, the state of $B$ is also the same, and the algorithm's correctness follows from the correctness of Dumbledore's algorithm. □

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 $O(n \log n)$ 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, all pretty minor.

First, the base case in `hagHelp` returns `max(0, A[s])`, but Dumbledore's definition of the task does not say that the smallest largest value is 0 (which would be okay if $A$ contained only non-negative values). Hence the correct base case would `return A[s]`.

Second, in the loop below `leftMax = sum = 0`, the limiting loop index is incorrect; the correct loop should be `for (i = m, i >= s, i--)`.

Third, in the loop below `leftMax = sum = 0`, the initialization of the loop index is incorrect; the correct loop should be `for (i = m+1, i <= t, i++)`.

(ii) The proof of correctness is given in Chapter 4.1 of CLRS, since `hagridSolve` is an equivalent implementation of the CLRS recursive algorithm for solving the **maximum subarray** problem. A proof here can be done using strong induction.

(iii) We can write the running time of `hagridSolve` as the general recurrence relation $T(n) = aT(g(n)) + f(n)$. Because `hagridSolve` just calls `hagHelp`, the running time of the former is given by the running time of the latter. First, we identify $a$ in the recurrence relation. In `hagHelp`, each time we enter a call to `hagHelp`, either we are in a base case or we will eventually make exactly two further calls to `hagHelp`. Thus, $a = 2$. Now, we identify $g(n)$. The two recursive calls are to problems of size $n/2$ because they are to the subarrays $A[s..m]$ and $A[m+1..t]$, where $m = (s+t)/2$. If $t - s + 1 = n$, then the size of the first call is $m - s + 1 = (s + t - 2s + 2)/2 = (t - s + 2)/2 = (n + 1)/2$ and the size of the second call is $t - (m + 1) + 1 = (2t - s - t)/2 = (n - 1)/2$. Thus, $g(n) = n/2$. Finally, we identify $f(n)$. When we are not in a base case, we perform two `for` loops both starting at $i = m$, but one of which iterates from $m$ down to $s$ and the other of which iterates from $m$ up to $t$. Thus, together the two loops examine every element $A[s..t]$. The work done within each for loop take constant time. Therefore $g(n) = \Theta(n)$, and $T(n) = 2T(n/2) + \Theta(n)$.

(iv) The recurrent relation $T(n)$ is the same as Mergesort, which we can solve using a recurrence tree or the master theorem, yielding $T(n) = O(n \log n)$.

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.*

Picking the best-possible pivot means choosing the median, so that when `Partition` returns, it has divided the problem into two subproblems of size $n/2$ each. Picking the worst-possible pivot means choosing the minimum or maximum, so that when `Partition` returns, it has divided the problem into one part of size 0 and one part of size $n - 1$.

Thus, the recurrence relation is $T(n) = T(n/2) + 2n$ because effectively, we run the `Partition` algorithm twice for each time we divide the size of the problem in half. Applying the Master method yields $T(n) = O(n \log n)$, which is the same running time as the best case complexity. Basically, this weird version of `Partition` just double the depth of the recursion tree for QuickSort, which makes the depth $2 \log_2 n$ rather than $\log_2 n$. But, the depth is still $\Theta(\log n)$ and hence the running time is asymptotically the same as for an algorithm always chooses the best pivot.