CMPS 102 Homework 1

Benjamin Gordon / bcgordon@ucsc.edu

4/21/2014

1 Problem 1

For the purposes of this proof, let us define "sequential" to be a property of an array such that its first element is smaller than its last element; "non-sequential" will henceforth be used to mean the exact opposite of "sequential" (I.E. the array's first element is greater than its last element).

1.1 Procedure

First, we check if the entire array is sequential. Though we are told that the array wraps around somewhere in the middle, it is possible that we could have an edge case in which it wraps around at the end of the array, leaving it perfectly ordered; this checks for such a case. If the array is sequential, then the answer is the last element in the array.

For each recursive frame, we divide the array in half, creating two sub-arrays - left and right - and check whether these new sub-arrays are sequential. At this stage, there are two possible cases:

1. One of the two sub-arrays is sequential, while the other is non-sequential.
2. Both of the sub-arrays are sequential.

If the first case is true, we disregard the sequential sub-array and use the non-sequential one as the input for a new recursive frame. We continue recursively dividing the arrays and checking whether they are sequential until we reach the second case (both arrays are sequential). At this point, we know that we have divided the array in between where it wraps; our solution is therefore the last element in the left sub-array.

1.2 Proof of Correctness

Because the numbers in the array must be distinct and strictly increasing, if array[n] $>$ array[n+1] for some index n, then array[n] is the largest number in the array. The array only wraps around at one point, so when we divide the array in half, only one of the halves can contain said point. When we check whether one of the two sub-arrays is non-sequential, we are actually trying to

find which sub-array contians the wrap point. As we are only concerned with the sub-array containing the wrap (and therefore the largest element), we can disregard the other sub-array.

However, it is also possible that neither array contains this point; if this is the case, then we have divided the array in between the point where it wraps. This corresponds to case 2 in the procedure. If we have, in fact, divided the array between the largest and smallest values, then it is obvious that the last element on the left side is the largest element. The final piece of the proof is to recognize that this algorithm calls for us to continue subdividing the array until we divide between the largest and smallest elements - so we must eventually find the largest element. *Q.E.D.*

1.3 Proof of Runtime

At each recursive step, we:

- Divide the data set in half,
- Do two constant-time comparisons, and
- Continue the recursion on only one half of the data set.

Therefore, our recurrence relation looks like this:

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

For this relation, we get *b=1* and *c=2*, giving us an E of 0. $\Theta\left(n^E\right) = \Theta\left(n^0\right) = \Theta\left(1\right)$, which is the same as our constant-time comparison C; therefore, by case 2 of the master theorem, our runtime is $\Theta\left(n^E \log n\right) = \Theta\left(\log n\right)$.

2 Problem 2

For the purposes of this proof, let us define "majority" to be a property of an array such that $\frac{n}{2} + 1$of its elements are equal to each other.

2.0.1 Procedure

To begin with, we will divide the array of ballots in half recursively until we are left with a number of pairs of ballots. When we begin collapsing the recursive tree back up, each sub-array will be returned along with an integer value - call it p. For each pair, we compare the two ballots. If they are equal, we return the pair to its parent recursive frame, along with a p of 2 to denote that a majority was found in this pair. If they are not equal, they are returned to their parent frame along with a p of 0, denoting that there was no majority.

Then we have two pairs of ballots to compare. If neither of them had a majority (they both had a p of 0), we can simply concatenate one to the other, and return them up the recursive tree without doing any comparisons. If, however,

one of the pairs had a majority, we take all the ballots from the non-majority pair and compare them to the first ballot of the majority pair. We count the number of equalities we find, and add that to the p that was returned to this frame along with the majority pair. As long as the resulting p value is greater than half of the current size of the array, there continues to be a majority in this sub-array. In such a case, we would concatenate the non-majority pair to the end of the majority pair, and return this new array along with the new p (if both pairs had a majority, the procedure is essentially the same - just pick one at random to act as the non-majority pair). If, however, the resulting p is less than or equal half the size of the current array, the algorithm returns a p of 0.

We continue this process of comparing all the ballots of one sub-array to the first value of the other, counting equalities, and checking that the majority still exists, until we have reconstructed the array completely. If the p value of the final, complete array is greater than $n/2$, then we know that we have a winner in the election; otherwise, we need to take another vote.

2.1 Proof of Correctness

The core of this algorithm's correctness is the following two theorems, which I will briefly prove before continuing.

2.1.1 Theorem 1 Proof

Claim: In order for any one candidate to have a majority over the whole array, they must have a majority in at least one of the halves of the array.

Proof by contradiction: Assume that there is a candidate who has a majority in the array, but does not have a majority in either half of the array. Then the greatest number of votes that said candidate can have in each half is $n/4$, with n being the size of the original array. If this is the case, then the total number of votes that this candidate has is $n/2$, which means that this candidate does not have a majority. *Q.E.D.*

2.1.2 Theorem 2 Proof

Claim: For any size array, we can always determine whether it has a majority in $O(n)$ as long as we know whether its two sub-arrays have a majority, and the size of the majority (if there is one).

Proof: There are three possible cases:

1. Neither sub-array has a majority.
2. One sub-array has a majority, while the other does not.
3. Both sub-arrays have a majority.

In the event of case 1, by the theorem proven above, we know - without doing any comparisons - that the newly combined array cannot possibly have a majority. Because no comparisons are necessary, this takes a constant amount of time.

In the event of case 2, it is possible that the array might have a majority; however, we need to do some analysis to be sure. Let us call the sub-array containing the majority a, and the other sub-array b; let us also give the candidate for whom there is a majority, the name x. Since the first element in a always contains a ballot for x (because we concatenate non-majority sub-arrays to the end of majority sub-arrays), by comparing each element of b to the first element of a, we can count how many ballots in b are for candidate x. If we add this value to a's p value, we get the total number of ballots in both arrays that are for x. If this value is greater than half of the length of $a+b$, then candidate x still holds a majority in the full array; otherwise, there is no majority. Because we don't have to compare each ballot to each other ballot - just half the ballots to a single ballot - this case takes $\Theta(n)$ time.

Case 3 is handled identically to case 2, with the exception that either array can be either a or b. Because it is handled the same way as case 2, case 3 also runs in $\Theta(n)$ time. *Q.E.D.*

2.1.3 Algorithm Proof

Since a pair of ballots only contains two ballots, if there is a majority in said pair, then the two ballots must be equal. Since we can check their equality with a single comparison, we can check for a majority with a single comparison. If there is a majority, we know exactly what the size of the majority is (it is 2).

Now we have all the information we need - we know whether the pair has a majority, and we know the size of the majority (if there is one). By applying theorem 2 at each recursive frame, we can determine whether the entire array has a majority or not. *Q.E.D.*

2.2 Proof of Runtime

At each recursive frame, we:

- Divide the current array in half, calling the algorithm recursively on each side, and
- Once the two child frames return, if one of them contains a majority, do n ballot comparisons.

Therefore, the recurrence equation looks like this:

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

For this relation, we get $b=2$ and $c=2$, giving us an E of 1. $\Theta\left(n^E\right) = \Theta\left(n^1\right) = \Theta(n)$, and $f(n) \in \Theta(n)$; therefore, by the master theorem, our runtime is $O\left(n^E \log n\right) = O(n \log n)$. I say $O(n \log n)$and not $\Theta(n \log n)$ because in many cases, the algorithm will skip a number of comparisons, or all of them, leading to a faster runtime.

3 Problem 3

3.1 Informal Argument for Correctness

For each number, consider where it should end up in the final, sorted array, and where it begins in the unsorted array. If the number begins in the upper third in the array, but it should be in the lower third in the sorted array, then it will be moved to the middle third during the second sort, and then down into the lower third during the third sort. If the reverse is true - the number begins in the lower third but should be in the upper third - then it will be moved to the middle third during the first sort, and then to the upper third during the second sort. In this way, each number will end up in its proper section in the array. Since the algorithm runs recursively, each subset of the array can be considered its own array in which these same properties hold - so the algorithm must return a fully sorted array.

3.2 Recurrence Relation

At each recursive step, we take $^2/_{3rds}$ of the original array, and we do so 3 times. Ultimately, since we continue dividing the arrays until they are of minimal size, and only sort the minimally sized arrays, the nonrecursive cost at each frame is trivial. Therefore, the recurrence relation looks like this:

$$T(n) = 3T\left(^{2n}/_3\right) + C = 6T\left(^n/_3\right) + C$$

Which gives us $b=6$ and $c=3$. $^{\log_2 6}/_{\log_2 3} \approx 1.63$, so by the Master Theorem, $T(n) \in \Theta\left(n^{1.63}\right)$.

4 Problem 4

4.1 Procedure

Base Case: We cannot find median+1 or median-1 in the array.

Recursive Step: Find the median of the indices we are currently considering (for the first run, this is every index). Check if median is equal to $\lfloor ^k/_2 \rfloor$, where k is equal to the sum of the first and last indices (not the values at those indices) we are currently considering. Now there are two cases:

1. The median is equal to $\lfloor ^k/_2 \rfloor$.
2. The median is greater than $\lfloor ^k/_2 \rfloor$.

In the event of case 1, search for median+1 in the array. If it is not found, we have reached the base case; median+1 is our answer. Otherwise, perform quicksort's partition function with median as the pivot; once this is done, the median is in the exact midpoint of the array, and all values greater than it are

after it in the array. Begin the recursive step again, this time considering only the indices after the median.

In the event of case 2, search for median-1 in the array. If it is not found, we have reached the base case; median-1 is our answer. Otherwise, perform quicksort's partition function with median as the pivot; once this is done, the median is in the exact midpoint of the array, and all values lesser than it are before it in the array. Begin the recursive step again, this time considering only the indices before the median.

4.2 Proof of Correctness

If the median is equal to $\lfloor k/2 \rfloor$, this means that in the array we are considering, no values lower than the median have been skipped; the skipped value must be greater than the median. Conversely, if the median is greater than $\lfloor k/2 \rfloor$, then a value lower than the median has been skipped. We can therefore shrink the number of values to consider to $\lfloor n/2 \rfloor + 1$, which will be an even number. By searching for median +1 or median -1, we can reduce this number to $\lfloor n/2 \rfloor$ - an odd number - without changing the overall asymptotic order. Ensuring that we always work with arrays of odd-number size makes things much simpler. Eventually, we will shrink the sub-array we have to consider down to a size of three - which means that we will always eventually reach a point where the skipped value is either median +1 or median -1 - which means we will always eventually find the skipped value.

4.3 Proof of Runtime

At each recursive frame, we:

- Find the median ($O(n)$),
- Compare that value to $\lfloor k/2 \rfloor$ ($O(1)$),
- Run quicksort's partition function ($O(n)$),
- Search for median +1 or median -1 ($O(n)$), and
- Continue recursing on one half of the array, or return median +1 or median -1 if it was not found.

Therefore, the recurrence relation looks like this:

$$T(n) = T(n/2) + 3n$$

Which is equal to:

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

Because $3n \in O(n)$. Thus we have $b = 1$, $c = 2$ and E $= 0$. $n^E = n^0 \in O(1)$ and $f(n) \in O(n)$, so by the Master Theorem, $T(n) \in O(n)$.