CS170–Spring 2015 — Solutions to Homework 2

Kevin Chau, SID 23816929, cs170-pz

Collaborators: Howard Chiao

1. Recurrences

(a) $T(n) = 7T(n/2) + n^2 + \log n$
Answer: $T(n) = O(n^{\log_2 7})$
Since n^2 dominates $\log n$, we can reexpress this recurrence relation as $T(n) = 7T(n/2) + O(n^2)$. This recurrence splits each stage into 7 sublems of size $n/2$. Using Master theorem we have $a = 7$, $b = 2$, and $d = 2$. Since $\log_2 7 > 2$, Master theorem tells us that this recurrence has an upper bound run time of $O(n^{\log_b a}) = O(n^{\log_2 7}) \approx O(n^{2.8})$.

(b) $T(n) = T(n-1) + 2/n$
Answer: $T(n) = \Theta(\log n)$
Assume T(1) is just a O(1) constant time calculation. Plugging in for small values of n we quickly see that the recurrence relation can be re-expressed as $T(n) = 2 + 1 + 2/3 + 2/4 + ... + 2/(n-1) + 2/n = 2\sum \frac{1}{n}$. This is just the harmonic series, which we know runs in $\Theta(\log n)$ time.

(c) $T(n) = T(\sqrt{n}) + 10$
Answer: $T(n) = \Theta(\log \log n)$
Assume a base case $T(2) = O(1)$. Consider taking successive squares of 2 (4, 16, 256, ...) and plugging them into the recurrence relation. For example, $T(16) = T(4) + 10 = T(2) + 10 + 10$ For this simple case, the runtime is just $10 \times \log \log 16 = 20$. We can get this $\log \log n$ factor by noting our values for successive squares of 2 take the form of 2^{2^m} for $m = 0, 1, 2, 3....$ The number of 10s we get in the recurrence relation is just equal to m, since $m = \log \log 2^{2^m}$ is the number of square roots we must take to get down to the base case of T(2).

2. k-ary Search

The algorithm described for the k-ary generalization of binary search has the following pseudocode:

```
# Input: sorted array A[1...n], element X      Output: index of X
ksearch(A[1...n],X):
    if A[index] == X:     #when k search gets down to arrays of size 1
        return index      #if singlet array does not contain X, return failure
    for i from 1 to k-1:
        if X < in/k:
            ksearch(A[((i − 1)n)/k...in/k − 1])
```

Analysis of the recurrence relation of this divide and conquer algorithm is straightforward. The ksearch breaks up the n sized input matrix into k subarrays, each of size n/k elements. It then compares X to at most k-1 start indices of these subarrays, doing O(k) = O(n^0) = O(1) work per level of recursion. As soon as the algorithm finds a start index i whose element is greater than the element X, it proceeds to recurse on the array whose start element is i-1, since this array is guaranteed to have the element X because the input array is sorted. It only recurses on one size n/k subarray (in other words it does not have to recurse on all k subarrays). We can plug in these values into master theorem and obtain a = 1, b = k, and d = 0. Since $\log_b a = \log_k 1 = 0 = d$, master theorem predicts that k-ary search will take $O(n^d \log n) = O(\log n)$ worst case running time. This is the same runtime as binary search, which is just the case were k = b = 2.

3. Peak Elements

A simple O(n) algorithm would traverse an array A in a linear fashion and compare each element A[i] to its neighbors, returning the first element that satisfies the peak conditions. Each element it inspects would do O(1) comparison operations, and in the worst case the algorithm would look at every single element.
We can do better with a divide and conquer style algorithm based on how binary search breaks up a problem.
Main Idea:
Given an input array A of n distinct elements, Peak algorithm returns the index of the first peak found in A. A peak element is defined as an element which is greater than both of its neighbors, or its only neighbor if it is the first or last element in the array. This algorithm uses divide and conquer.

Pseudocode:
$Peak(a[1, ..., n], startIndex, endIndex)$ #returns the index of the first peak found
middleIndex = $\frac{1}{2}(startIndex + endIndex)$
if $a[middleIndex - 1] < a[middleIndex] > a[middleIndex + 1]$: #found peak
return middleIndex
else if $a[middleIndex - 1] > a[middleIndex]$: #peak is in lower half
return $Peak(a[], startIndex, middleIndex - 1)$
else if $a[middleIndex] < a[middleIndex + 1]$: #peak is in upper half
return $Peak(a[], middleIndex + 1, endIndex)$

Proof of Correctness:
The algorithm models itself after binary search. First we note the following facts for any element i in A that we pick arbitrarily. If A[i] is less than the element before it, it must be that a peak is in the array of elements before a[i]. If A[i-2] is less than A[i-1], then A[i-1] is a peak. Otherwise the subarray split before a[i] is either a monotonically decreasing array (for which the peak is just the first element) or we will have to do similar analysis on A[i-2] to determine whether it is a peak or creates another subarray to search. A similar argument can be used to show that if A[i] is less than A[i+1], then a peak must be in the array of elements after A[i]. The pseudo code implements this exact rule, with the exception that we choose the middle indexed element of the array (as opposed to any random i) for which we split the recursive subproblems. Notice that because of this choice of the middle index the problem size is halved every recursive call, and there is only one subproblem to solve (we choose to only recurse on either the lower or the upper half). Thus we are guaranteed to find a solution in $\log n$ recursive calls, in which case we hit the base case where A[middleIndex] is either a peak with two neighbors or a peak at the ends of the array. Hence the algorithm is correct.
Asymptotic Analysis:
We know the divide and conquer algorithm binary search recurses on a problem of half size, so we chose our splitting element A[i] to be the middle index of the array, so that it recurses in a similar manner. Since Peak is essentially just a modified binary search, we know it has a worst case run time of $O(\log n)$.
Justification:
Each subarray is a problem of size n/2. The recurrence relation only makes one recursive call on of the two possible sub problems. It takes O(1) comparisons per recursive call. In the formulation of master theorem, this corresponds to a = 1, b = 2, and d = 0. Of course, since $\log_2 1 = 0 = d$, master theorem tells us that the logarithm runs in $O(n^d \log n) = O(n^0 \log n) = O(\log n)$.

4. Overlapping intervals

Main Idea:
Given n intervals, this algorithm finds the pair of intervals with the greatest overlap.
Pseudocode:
Presort the invervals by left endpoint.
Split the set of intervals in half, creating two sub problems.
For each group of intervals, find the two intervals with greatest overlap. Call these two pairings A and B respectively.
In the group with smaller left endpoints, find the interval with the largest right endpoint, and then find the interval in other group of larger left endpoints that has the greatest overlap with it. Call this pairing C Return interval(A, B, C).
Proof of Correctness:
This is just a recursive divide and conquer algorithm. In the base case of two intervals it just returns the two intervals, assuming our set of intervals always has at least 1 overlapping pair. By taking the maximum overlapping pair from among A,B, and C (the first subgroups max overlapping pair, the second subgroups max overlapping pair, and the max overlapping pair between the subgroups), we are guaranteed to recursively find the max pair. Thus the algorithm is correct.
Running Time:
$O(n \log n)$
Justification:
The initial sorting can be done efficiently with mergesort in $O(n \log n)$. Since the problem is halved every recursion, we know that the algorithm has $\log n$ levels of recursion. It also takes O(n) work to effectively merge the solutions from the two problems, since it takes O(n) work to find the first subgroups interval with the largest right endpoint and O(n) time to find the largest overlap with that interval in the second subgroup. Thus it should take $O(n \log n)$ time. We can also get these results from master theorem using a = 2, b=2, d = 1.

5. Missing Integer

Main Idea:
An algorithm for finding the missing integer in an array A of n elements with all integer from 0 to n except for one. The algorithm returns an integer B, where B_i is the i-th bit.
Pseudocode:

```
FindMissingInt(A[1...n]):
    for i from log n to 1:     #start from MSB
        if length(A) == 1:
            if i-th bit of A[1] == 0:
                B_i = 1 else 0
        look at i-th bit of all elements
        partition the array into two sub arrays depending on i-th bit
        If size(partition with i-th bit == 1) < size(partition with i-th bit == 0):
            B_i = 1 else B_i = 0
        remove from A all items in the partition with more elements.
```

Proof of correctness:
It's obvious that for the simple base case where the input array only has one element that we can find the missing integer by just doing a bitflip. It should be self evident that by dividing the array into two subgroups based on the i-th bit that we are looking at, we can determine the i-th bit of the missing integer. An array of that isn't missing an integer has an equal amount of 1's and 0's at any bit position that we look at. Therefore the i-th bit is just the value of the i-th bit for the subparitition with fewer elements. The algorithm does exactly this counting of 1's and 0's for each bit, but by iterating on smaller subproblems it is able to do it in better than $n \log n$ time.

Running Time: O(n)

Justification:
As specified in the problem, this algorithm only looks at O(n) bits, so we should expect it to take O(n) time. To see that it does in fact only look at O(n) bits total, notice that in the first scan of the array we look at n MSB bits, then in the second pass we look at n/2 next MSB bits, then n/4 bits, for a total of n + n/2 + n/4 + n/8 + ... We know that this series just converges to 2n, at most we look at O(n) bits. Of course, we only really have log(n) iterations so the summation only has log(n) terms. If we use the geometric sum formula up to log(n), we still recover an O(n) total amount of bits. A third way to get this run time is by using master theorem. The for loop effectively describes a recursive search on 1 single sub problem of size n/2, doing O(n) bit scans per iteration. Master theorem says that for a=1,b=2,and d=1 that the runtime should be $O(n^d) = O(n)$.

6. Polynomial and Complex Numbers

(a) Let $n = 4k$ for some integer k. The complex n-th root of unity ω is $(1, \theta)$, where $\theta = 2\pi m/n = 2\pi m/4k$ Then $\omega^{2015/n} = e^{i(2\pi m/4k)(2015(4k)/4)} = e^{i(2015/2)\pi m} = e^{i\frac{\pi}{2}m}$ for any integer m
$\underline{\omega^{\mathbf{2015n/4}} = \mathbf{1}, -\mathbf{1}, \mathbf{i}, -\mathbf{i}}$

(b) $1+\omega^3+\omega^6+...+\omega^{3n} = (1+\omega^3+\omega^6+...+\omega^{3n})\frac{1-\omega^3}{1-\omega^3} = \frac{1}{1-\omega^3}(1+\omega^3+...+\omega^{3n}-\omega^3...-\omega^{3n}-\omega^{3n}\omega^3) = \frac{1}{1-\omega^3}(1-\omega^{3n}\omega^3) = \frac{1}{1-\omega^3}(1-(\omega^n)^3\omega^3) = \frac{1}{1-\omega^3}(1-1^3\omega^3)$
$= \frac{1-\omega^3}{1-\omega^3} = 1$
$\underline{\mathbf{1} + \omega^{\mathbf{3}} + \omega^{\mathbf{6}} + ... + \omega^{\mathbf{3n}} = \mathbf{1}}$

(c) $\omega^{3n/2+1} + \omega = (\omega^{n/2})^3\omega + \omega = \omega((\omega^{n/2})^3 + 1)$
Since $\omega^n = 1$, $(\omega^n)^{1/2} = \omega^{n/2}$ is either +1 or -1. Evaluating the expression for these values, $\omega^{3n/2+1} + \omega = \omega((-1)^3 + 1) = 0$ or $\omega^{3n/2+1} + \omega = \omega((1)^3 + 1) = 2\omega$.
$\underline{\omega^{3n/2+1} + \omega = 0 \text{ or } 2\omega}$