CS170–Fall 2012 — Solutions to Homework 3

Ben Augarten, section 106, `cs170-bo`

September 13, 2012

1. (a) $T(n) = 2T(n/3) + 1$
   Branching factor of 2, depth of $\log_3(n)$, cost at each level is the number of problems. The last level dominates because in a geometric series the last term dominates (the cost goes up by a factor of 2 each level) so its just equal to the number of problems on the last level: $O(n^{\log_3(2)})$

   (b) $T(n) = 5T(n/4) + n$
   Branching factor of 5, depth of $\log_4(n)$, Cost at each level proportional to size of the problem, but number of problems grows faster than the size shrinks so the last level of the recursion tree dominates – $O(n^{\log_4(5)})$

   (c) $T(n) = 7T(n/7) + n$
   Branching factor of 7, depth of $\log_7(n)$. Each level does $O(n)$ equal work – $O(n \log_7(n))$

   (d) $T(n) = 9T(n/3) + n^2$
   Branching factor grows much faster than the problem gets smaller, but the amount of work is propertional to the square of the size of the problem (favoring higher levels). Results in each level requiring the same amount of work ($\frac{9}{3^2} = 1$) – $O(n^2 \log_3(n))$

   (e) $T(n) = 8T(n/2) + n^3$
   This is the same problem as (d) except now it branches a cube of how the problem shrinks (not sure how to actually say that), but the work it proportional to the cube of the size of the problem – same situation, each level does equal work. $O(n^3 \log_2(n))$

   (f) $T(n) = 49T(n/25) + n^{3/2}\log(n) = T(n) = 49T(n/25) + O(n^{76/50})$
   Now that we upper bounded the last term, we can apply masters – $\frac{49}{25^{3/2}} < 1$, so too is our upper bound for d ($25^{76/50}$), which we will use from now on. The first term dominates, so $O(n^{76/50})$. We can sort of see that this isn't the tightest upper bound we can provide. If we use the version of master's theorem on wikipedia, we see that for any $f(n)$, in this case $n^{3/2}\log(n)$, if it is $\Omega(n^{\log_b(a)+\epsilon})$ and $af(\frac{n}{b}) = O(f(n))$, then $T(n) = \Theta(f(n))$. $n^{3/2}\log(n) = \Omega(n^{\log_2 5(49)}) =$

$\Omega(n^{1.209})$ And $49f(n/25) = O(f(n))$. Therefore, $T(n) = \Theta(n\log(n))$, which is the tightest bound I can provide.

(g) $T(n) = T(n-1) + 2$
We have to visit every item from 1 to n and it takes O(1) for each item. O(n)

(h) $T(n) = T(n-1) + n^c$, $n \geq 1$
$n^c + (n-1)^c + (n-2)^c + ... + 2^c + 1^c = O(n^{c+1})$

(i) $T(n) = T(n-1) + c^n$, $c > 1$
$c^n + c^{n-1} + c^{n-2} + ... + c^2 + c$. Geometric series with $c > 1$ – series dominated by its last term, $O(c^n)$

(j) $T(n) = 2T(n-1) + 1$ This is the worst kind of recursion... each step you take, you double the problem, and you take n steps. That means there are 2^n steps, each taking $O(1)$ time, so you got an exponential on your hands – $O(2^n)$.

(k) $T(n) = T(\sqrt{n}) + 1$
Assuming that this terminates (there a base case at 1 if its floored or 2 if its not), then lets look at the depth of the recursion. For any n, you are done with the recursion when you get to $n^{1/\log_2(n)}$, so for any n, the recursion stops at $O(\log(\log(n)))$. Each step takes constant time so the total running time is $O(\log(\log(n)))$.

2. (a) Find an $O(n\log(n))$ way to find the majority element. Split the array into two halves. If an element is to be a majority element, it must be the majority element of at least one of these. Find the majority element of the first of the arrays. This takes $T(n/2)$. If this returns with a clear majority element (along with the number of times it appears), then we check the second half for the number of occurrences of this element in that half – if that number plus the first is greater than half of n then we return. If not, We find the majority element of the second of the arrays, taking $T(n/2)$. If it doesn't return an element, there is no majority element, else we search the first of the arrays for the second majority element. If the sum of the occurrences is greater than $n/2$ then return the element, else there is none. Lets evaluate the constant time factor at each step: At each step, once we have the answer to both subproblems (the majority element of each sub array), we can compute the majority element in $O(N)$ time. So the problem takes $T(n) = 2T(n/2) + O(n) = O(n\log(n))$

   (b) Find an $O(n)$ solution. Pair up each of the numbers arbitrarily, if they are the same, keep one of them, if they are different, discard both. After one pass of doing this, the maximum number of elements you can have is $n/2$ in the worst case if all of the elements are the same, then you have to keep one for every two elements that you have, otherwise you throw away more than $n/2$. The resulting set still has the same majority element. For every instance of the majority element you throw away in the process, you throw away another element. There can't be enough other numbers to throw away all of the majority elements, and then you'd have no other elements left. Likewise, if another number gets paired with itself often, there must be more than that number of pairings in the majority element or else it wouldn't be the majority.
   This recursion experiences time equal to $T(n) = T(n/2) + O(n)$ because it takes $O(n)$ to pair every element up and then throw them out. That equals $O(n)$, the first term dominates.

3. Let v' be the top half of v, v'' be the lower half of v.
   $H_k * v = \begin{pmatrix} H_{k-1}v' + H_{k-1}v'' \\ H_{k-1}v' - H_{k-1}v'' \end{pmatrix}$
   We have split the problem into 4 subproblems, but there are only 2 unique subproblems, each which is $n/2$. The time it takes to reconstruct the answer is $O(n)$ because addition and subtraction are constant time in the problem, and we have add n entries. $T(n) = 2T(n/2) + O(n) = O(n \log(n))$ according to master's thm.

4. (a) Four distinct points define a quadrilateral. Draw lines connecting each of the four points such that each doesn't have an intersection (the perimeter of the quadrilateral that the points make). Each of the segments must be at least d in length. That means the smallest possible area of the quadrilateral would be d^2 if the points make a square, which takes dxd plane. If you add one more point and try to space it d apart from the other points, its area would have to be larger than d^2, unless of course $d = 0$ but then the points wouldn't be distinct, and the points of L must be distinct.

   (b) If the closest pair is in L then it will be p_L, q_L and will be in the final three pairs. Likewise if the closest pair is in R. If the pair is split between L and R we can safely discard all points with $x_i < x - d$ in L because their x distance from any given point in R is greater than d, and any $x_i > x + d$ in R because their x distance from any point in L is greater than d. Now sort the remaining based on y value. For each point in the left half, in order for another point to exist that is less than d away, it needs to fall in a $2dxd$ rectangle with the point on the edge of the rectangle. We can just for simplicity sake say there are two dxd rectangles and say that there can only be 8 points in both of these that are d away, one of which has to the be the point in question, so we only have to look at at most 7 others. One of these pairs must be the closest pair.

   (c)
```
def closest_pair(pairs):
  if len(pairs) < 2:
    # error
  if len(pairs) == 2:
    return dist(pairs[0], pairs[1])
  x = getMiddleXValue(pairs)
  dl, pl, ql = closest_pair([pair for pair in pairs if pair.x < x)
  dr, pr, qr = closest_pair([pair for pair in pairs if pair.x > x)
  d = min(dl, dr)
  pairs = [pair for pair in pairs if pair.x > x-d and pair.x < x + d]
  pairs = sortYValues(pairs)
  mindist = 10000*d
  minpair = ()
  for pair in pairs:
    for otherpair in pairs.getNext7Elements(pair):
      if dist(pair, otherpair) < mindist
        mindist = dist(pair, otherpair)
        minpair = (pair, otherpair)
  return the minimum pair corresponding to min(dl, dr, mindist)
```

   The recurrence is because each time we split the problem in half and make two new problems ($2T(n/2)$). Once we have this result, we have to sort which takes

$O(n\log(n))$, find the median in the beginning $O(n\log(n))$ and finally the loop is $O(n)$ so the cost at each step is $O(n\log(n))$, giving the recurrence $T(n) = 2T(n/2) + O(n\log(n))$. If you draw out the recurrence tree (I'm not sure how to make a tree in latex), you see that at each level, i, there are 2^i problems, each of size $\frac{n}{2^i}$, each costing $c\frac{n}{2^i}\log(\frac{n}{2^i})$. The total cost for each level then is $cn\log(\frac{n}{2^i})$. Sum over all levels produces the following sum: $cn\sum_{i=1}^{\log(n)}\log(\frac{n}{2^i}) = \Theta(n\log^2(n))$.

(d) If we presort the list based on the y coordinates before beginning the algorithm, then we won't need the $O(n\log(n))$ step of sorting in each level of the recurrence. Now the costs at each level are finding the median, which we can do in $O(n)$ as shown in class, filtering the list ($O(n)$) and traversing the sorted list of L and R in $O(n)$ time. That makes the new recurrance $T(n) = 2T(n/2) + O(n) = O(n\log(n))$. Or you can just put the y coordinates in buckets of d length at each step, and then examine elements in the the bucket that the coordinate is in along with the two buckets next to it. Putting the elements in buckets takes $O(n)$, so it still brings down the total time to $O(n\log(n))$ and you don't have to worry about keeping the list sorted throughout.

5. (a) $\omega = e^{2\pi i/8}$
   $\omega + \omega^7 = e^{\pi i/4} + e^{7\pi i/4} = \frac{\sqrt{2}}{2} + \frac{\sqrt{2}}{2}i + \frac{\sqrt{2}}{2} - \frac{\sqrt{2}}{2}i = \sqrt{2}$

   (b) Note that each one is just a rotation around the unit circle, and then add 1.
   $p(x) = x^2 + 1$, $p(1) = 2$, $p(\omega) = 1 + i$, $p(\omega^2) = 0$, $p(\omega^3) = 1 - i$

   (c) I feel like there should be a way to use the fact that ω^2 is a 4th root of unity, but we only have $A(1), A(\omega), A(\omega^2), A(\omega^3)$... so I'm just going to brute force the other way.
   I inputted:
   $[[1,1,1,1],[1,e^{2*pi*i/8},e^{4*pi*i/8},e^{6*pi*i/8}],[1,e^{4*pi*i/8},e^{pi*i},e^{12*pi*i/8}],$
   $[1,(e^{2*pi*i/8})^3,(e^{2*pi*i/8})^6,(e^{2*pi*i/8})^9]]^{-1} * [[3],[1+\sqrt{(}2)*i],[1],[1+\sqrt{(}2)*i]]$
   into wolfram alpha, which multiplies the $A(\omega)$ matrix by the inverse of $\begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & \omega & \omega^2 & \omega^4 \\ 1 & \omega^2 & \omega^4 & \omega^6 \\ 1 & \omega^3 & \omega^6 & \omega^9 \end{pmatrix}$.
   This process gets the coefficients of the polynomial, $\begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \end{pmatrix} = \begin{pmatrix} 1 \\ 1 \\ 0 \\ 1 \end{pmatrix}$
   $q(x)$, which equals $q(x) = x^3 + x + 1$

6. The first problem is how to analyze whether a plan is optimal and how it can be optimal. Obviously, at each node we want to expand its child whose latency will be most first, to minimize the total latency. This requires sorting some type of score associated with each node. We can sort in $O(n\log(n))$, where n is the number of children a node has. Before we get into the sorting of children, we first have to score nodes. The latency it takes at a leaf node is 1. The latency it takes at any given node is equal to the max of 1+ the child with the largest latency and the number of children a node has. There are two possible scenarios – if a node has a ridiculously large number of children, but each child is of latency 1, then the node takes latency equal to the number of children. On the other hand, if the latency of a particular child is equal to or greater than the number of children the node has, then that node will limit the total latency of the branch. Therefore, we can compute recursively the latency of each node doing a simple traversal, scoring each leaf node first and then working our way back up. The traversal takes $O(N)$ where N is the number of nodes in the tree. Now we need to come up with a schedule for calling. For each node, sort its children, in $O(n\log(n))$ time. Lets say we have to do this k times. That means each of the sorts are on average $O(\frac{n}{k}\log(\frac{n}{k}))$, or $O(k\frac{n}{k}\log(n/k)) = O(n\log(n))$ for all of the sorting. In the worst case, we have to sort $\Theta(N)$ elements at once, but if we do this in one pass, then the rest of the sorts are $\Theta(1)$ and so it all of the sorting still turns out to take $O(n\log(n))$. Now that each of the node's children are sorted, we can simply output a list, the node's label followed by its children, sorted from highest latency to lowest latency. The total time is equal to $O(n)+O(n\log(n))+O(n)$ (the last $O(n)$ from outputting the list but we could've combined it with the sorting stage). The total running time, then is $O(n\log(n))$.