CPSC 320 2014W2 Midterm #2 Sample Solutions

March 13, 2015

1 Canopticon [8 marks]

Classify each of the following recurrences (assumed to have base cases of $T(1) = T(0) = 1$) into one of the three cases of the Master Theorem—the cases in which **leaves** dominate, in which the **root** dominates, and in which the work is **balanced** across levels—or indicate that the Master Theorem **does not apply**.

1. $T(n) = 2T(\lceil n/4 \rceil) + 2^n$

   SOLUTION: $a = 2, b = 4, \log_b a = 0.5, f(n) = 2^n$. $2^n \in \Omega(n^c)$ for **any** constant c; so, as long as 2^n meets the regularity condition, this is the **root**-dominated case. Sure enough, $a \cdot 2^{n/b} = 2 \cdot 2^{n/4} \leq k \cdot 2^n$ for $n \geq 2$ and $k = 1/2$ (or a variety of other combinations of n-values and k choices). As a point of interest, the Master Theorem gives us a runtime of $\Theta(2^n)$ for this recurrence.

2. $T(n) = T(\lfloor n/100 \rfloor) + n^2$

   SOLUTION: $a = 1, b = 100, \log_b a = 0, f(n) = n^2$. $n^2 \in \Omega(n^2)$, and $2 > 0$; so this is again the **root**-dominated case (and so has a bound of $\Theta(n^2)$). We have already seen that the Master Theorem is usable on n^2 in previous problems, but we can recheck. $a(n/b)^2 = 2(n/4)^2 = n^2/8 \leq \frac{1/8}{n}^2$; so, $k = 1/8$ is a natural choice.

3. $T(n) = 16T(\lfloor n/4 \rfloor) + n^2$

   SOLUTION: $a = 16, b = 4, \log_b a = 2, f(n) = n^2$. $n^2 \in \Theta(n^2)$; so this is the **balanced** case. (Thus, the bound is $\Theta(n^2 \lg n)$.)

4. $T(n) = 81T(\lceil n/3 \rceil) + n^3$

   SOLUTION: $a = 81, b = 3, \log_b a = 4, f(n) = n^3$. $n^3 \in O(n^3)$, and $3 < 4$; so, this is the **leaf**-dominated case. (Thus, the bound is $\Theta(n^4)$. In case you had trouble with $3^4 = 81$ without a calculator, note that you could have stopped with $3^3 = 27 < 81$.)

2 Doctoring the Master Theorem [10 marks]

Answer the questions below about this recurrence:

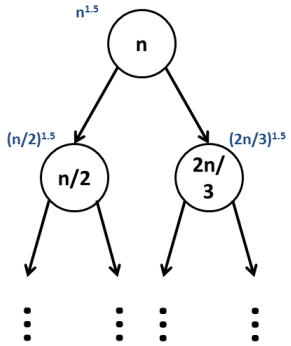
$$T(n) = \begin{cases} T(\lfloor n/2 \rfloor) + T(\lfloor 2n/3 \rfloor) + n\sqrt{n} & \text{when } n > 1 \\ 1 & \text{when } n \leq 1 \end{cases}$$

Throughout this problem, you may ignore floors and ceilings.

NOTE: the recurrence and some questions or parts of questions have been changed from practice version.

1. Draw a recursion tree for this recurrence. Include **at least two levels (root and children)**. Label your tree with: the **problem size** (in each node), and the **work per node** (next to each node). **[5 marks]**

   SOLUTION: Our tree is below. Note that we used $n^{1.5}$ in place of $n\sqrt{n}$. That makes it easier to avoid the common mistake of giving, e.g., the work at a node with problem size $n/2$ as $(n/2)\sqrt{n}$ as opposed to $(n/2)\sqrt{n/2}$.



2. Give the depth of the deepest leaf (ignoring floors and ceilings). **[2 marks]**

   SOLUTION: The deepest leaf will be under the rightmost (all $2n/3$) branch, which goes down by a factor of only $3/2$ at each level. So, we'll reach the leaf when $\frac{n}{(3/2)^i} = 1$. Solving for i, we get a depth of $\log_{3/2} n$.

3. Use the Master Theorem to **give a Θ-bound on this related recurrence** $T_2(n) = 2T_2(n/2) + n\sqrt{n}$ (which gives an Ω-bound on $T(n)$). **Clearly indicate** a, b, and $f(n)$. **[2 marks] SOLUTION:** $a = 2, b = 2, \log_b a = 1, f(n) = n^{1.5}$. So, this is the root-dominated case. Does $f(n)$ meet the regularity condition? $2f(n/2) = 2(n/2)^{1.5} = n^{1.5}/\sqrt{2} \leq \frac{1}{\sqrt{2}} n^{1.5}$. So, yes, $f(n)$ meets the condition with $k = \frac{1}{\sqrt{2}}$ (and many other possible values!).

   Thus, the bound is $\Theta(n\sqrt{n})$.

4. Use a similar technique to construct a modified recurrence $T_3(n)$ from which we could derive an O-bound on $T(n)$. (**DO NOT** give a bound on your recurrence.) **[1 mark]**

   SOLUTION: Instead of "lowering the larger term" to match the smaller, we'll "raise the smaller term" to match the larger: $T_3(n) = 2T_3(2n/3) + n\sqrt{n}$.

3 The High Price of Plausible Deniability [11 marks]

You're solving the interval scheduling problem except **minimizing** the number of jobs performed rather than maximizing it. In particular, we define *the conflict set of a job* to be the set of all jobs that conflict with that job's time range. (Note that the conflict set of a job always includes the job itself.) Your solution should minimize the number of jobs performed while still performing exactly one job from each conflict set.

(Note: we consider two jobs' times to conflict even if the start time of one job is equal to the finish time of the other, i.e., they overlap at only one point.)

UNECESSARY FLAVOR TEXT: Your boss has just given you a list of jobs to perform. Each job has a start time and an end time. You can never do more than one job at a time. You're kind of tired; so, you'd like to do as few jobs as possible, but you can't just do **nothing** or you'll get fired. So, you want to find a list of the smallest number of jobs you can do so that every **other** job conflicts with (has times that overlap) at least one of the jobs you **are** doing.

A friend suggests breaking the problem down into two cases: one where we **include** the first job in the solution and one where we **do not** include it.

Your friend's algorithm takes array `ByStart` as input. `ByStart` is sorted in order of increasing start time. Each array entry is an object with a start time field `start` and finish time field `finish`; so, `ByStart[1].start` is the first job's start time and `ByStart[1].finish` is its finish time.

Here is the algorithm:

```
LazyISP(ByStart):
  Return LazyISPHelper(ByStart, 1)


LazyISPHelper(ByStart, i):
  If i > length(ByStart):
    Return 0
  Else:
    Let j be the smallest index for which            // For part 3 below,
      ByStart[j].start > ByStart[i].finish           // assume these three
      or length(ByStart)+1 if no such index exists   // lines take O(1) time
    Let v1 be 1 + LazyISPHelper(ByStart, j)
    Let v2 be LazyISPHelper(ByStart, i+1)
    Return min(v1, v2)
```

1. Give an instance **with no more than 2 jobs** that shows that this algorithm is **not correct**. Indicate **both** what the algorithm returns and the correct answer. **[1 mark]**

   SOLUTION: Actually, any non-empty instance will work. The trouble is that once there's at least one job, we are required to take at least one job. (Why? Well, that one job has a conflict set, and we need to take exactly one job from that conflict set. So, we need at least one job.)

   However, the algorithm always lets us choose between including the job and excluding it. In the latter case, we just solve the subproblem that doesn't include that job. However, the recurrence will let us do that over and over until we run out of jobs, reach the base case, and get 0 as a result. Then, the application of `min` will propagate that 0 to **every** entry in my table.

   Bottom line: if my instance is this job — then the correct solution includes it while this algorithm excludes it.

   This is a good moment to mention, however, that we **can** get quite good performance on this problem from a DP solution. (Better than the practice problem approach!) There is a solution that runs in $O(n^2)$ time and $O(n)$ memory (for n jobs) in the worst case.

2. Give a good Θ-bound on the worst-case runtime of the algorithm if we rewrite it to use dynamic programming (**without** trying to fix it). **Assume** that the three lines beginning at `Let j be the smallest` take constant time. **[2 marks]**

   SOLUTION: There are n entries in the table. Each one requires choosing between two options (two table entry lookups with a small (constant) amount of logic and arithmetic on top). So, that's $\Theta(n)$ work overall.

3. Rewrite `LazyISP` below to use dynamic programming (**without** trying to fix it): **[8 marks]**

   SOLUTION: We rewrite it in place, leaving our `__answers__` in blanks for clarity:

```
LazyISP(ByStart):

  Let Soln be an array of length __Length(ByStart)__

  For i = __Length(ByStart) down to 1__:

    Let j be the smallest index for which
      ByStart[j].start > ByStart[i].finish
      or length(ByStart)+1 if no such index exists

    Let v1 be __if (j <= Length(ByStart)) then 1 + Soln[j] else 0__

    Let v2 be __if (i+1 <= Length(ByStart)) then Soln[i+1] else 0__

    __Soln[i] = min(v1, v2)__

  Return __Soln[1]__

                // Feel free to write a helper function below if you need one!
```

   Two notes on the solution. First, using a "SolnCheck"-style helper function to wrap array accesses would eliminate those awkward "inline" if-then-else expressions above. Second, note that **as ALWAYS**, the DP solution just sets up an ordering on the subproblems and then does **exactly** what the recursive solution does inside the loop.

   MANY people are struggling with trying to invent a whole new algorithm for their DP solution. That's not how DP works! It's **not** a new algorithm. It's just a trick to wrap around the original one and make it more efficient.

4 I'm a k, You're $O(k)$ [13 marks]

Suppose that we are given an array `A` with n distinct elements and a guarantee that no element in `A` is more than $k-1$ indexes away from the index it belongs at in a sorted array (for some positive integer $k \leq n$), and we want to sort the array.

For example, the array `[15, 3, 19, 12, 16, 10, 21, 18]` in sorted order is: `[3, 10, 12, 15, 16, 18, 19, 21]`. `15` is therefore 3 indexes out of place (would be moved 3 places to the right to put it in the "correct" spot), `3` is only 1 index out of place (would be moved 1 place to the left), and `19` is 4 indexes out of place. Overall, the smallest k we could provide for this array is 5 (because $5-1=4$, and both `19` and `10` are 4 indexes out of place).

1. Complete this divide-and-conquer algorithm to sort an array of length n given the array `A` and the parameter k described above so that it is well-described by the recurrence:

$$T(n) = \begin{cases} 2T(n/2) + T(2k+1) & \text{when } n > 2k+1 \\ k \lg k & \text{when } n \leq 2k+1 \end{cases}$$

Do NOT assume (as in the previous part) that $k \leq n \leq 2k+1$. **[4 marks]**

HINT: Consider an element that belongs in the right half of the sorted array (e.g., something larger than the median). What is the smallest index at which you could find that element in `A`?

SOLUTION: We rewrite it in place, leaving our `__answers__` in blanks for clarity:

```
BoundedSort(A, k):
  BSHelper(A, k, 1, length(A))

BSHelper(A, k, lo, hi):
  If hi - lo + 1 <= 2k + 1:
    MergeSort the portion of A from lo up to hi
  Else:
    Let mid = lo + ceiling((hi - lo)/2)

    BSHelper(A, k, __mid - k__, __mid + k__)

    BSHelper(A, k, __lo__, __mid - 1__)

    BSHelper(A, k, __mid__, __hi__)
```

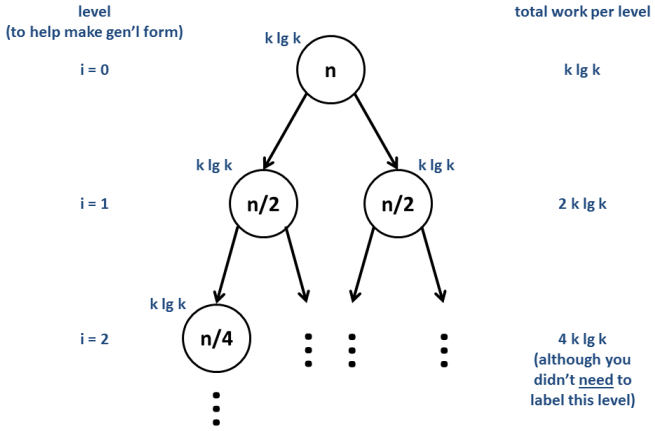
Various very similar alternatives (e.g., `lo` to `mid` and `mid+1` to `hi` instead) would be fine.

2. Draw a recursion tree for the closely related recurrence relation below. Include **at least two levels (root and children) and one node from the next level**. Label your tree with: the **problem size** (in each node), the **work per node** (next to each node), the **total work per level**, and a **general form for the work per level** (labelling the level i to avoid confusion with k above). **[6 marks]**

The recurrence is:

$$T(n) = \begin{cases} 2T(n/2) + k \lg k & \text{when } n > 2k+1 \\ k \lg k & \text{when } n \leq 2k+1 \end{cases}$$

SOLUTION: Here is our tree:



3. Use your tree to show that the recurrence above is in $O(n \lg k)$. **Show your work.** You may (correctly!) assume that the height of the tree is $\lg(\frac{n}{k})$. *Reminder:* $\sum_{i=0}^{n} 2^i = 2^{n+1} - 1$. **[3 marks]**

SOLUTION: We'll sum the work per level to get the total work in the tree. (Technically, we are off-by-one or so in our depth calculation, but it would not turn out to matter; so, we kept it simple!)

$\sum_{i=0}^{\lg(\frac{n}{k})-1} 2^i k \lg k = k \lg k \sum_{i=0}^{\lg \frac{n}{k}-1} 2^i = k \lg k (2^{\lg \frac{n}{k}} - 1) = k \lg k (\frac{n}{k} - 1) = n \lg k - k \lg k \in O(n \lg k)$

5 I Want the Truth [8 marks]

For each statement below, circle **one** answer to indicate whether the statement is **always** true, **never** true, or **sometimes** true for the circumstances indicated. So, if every possibility indicated causes the statement to be true, answer "always". If none causes the statement to be true, answer "never". If some cause the statement to be true and others cause it to be false, answer "sometimes".

1. Evaluate this statement over the possible input arrays of integers with length of at least 10 passed to the algorithm: The `RandomizedQuickSelect` algorithm picks the smallest element as its pivot.

   SOLUTION: Sometimes it could pick the smallest. It picks the pivot uniformly at random and so has just as much chance of picking one order statistic as another.

2. Evaluate this statement over the set of all realistic divide-and-conquer algorithms: Divide-and-conquer algorithms solve non-base-case inputs by: dividing the input into two subproblems, solving the two subproblems, and then computing a solution based on the subproblems' solutions.

   SOLUTION: MergeSort does this, but BinarySearch does not. Some algorithms divide into more or fewer than two subproblems.

3. Evaluate this statement over the legal instances of the closest pair of points problem with at least four points: A pair of points is less than δ apart within the δ-wide strip on the **left side** of the dividing line (i.e., both points are on the left side) on the top-level recursive call to the divide-and-conquer closest pair of points algorithm.

   SOLUTION: δ is the closest distance between any pair of points, where both points are on the same side of the dividing line (i.e., the minimum of the closest distance between a pair of points on the left and the closest distance between a pair of points on the right).

   Given that, there **cannot** be two points where **both are on the same side** that are closer than δ!

4. Evaluate this statement over the instances of weighted ISP (interval scheduling problem) in which **all weights are the same** (e.g., all 3 or all 7): Running the greedy algorithm for **unweighted** ISP on the instance (with the weights deleted) selects as a solution a set of jobs that would also be optimal if selected in the original instance.

   SOLUTION: This set of jobs will always be optimal, because the unweighted ISP problem acts like a weighted ISP with weights all equal to 1, and uniformly scaling the weights (to any other positive constant) has no impact on the relative value of different solutions.

6 Bonus [Up to 4 Bonus Marks]

Bonus marks add to your exam total and also to your course bonus total. What course bonus points are worth **still** isn't clear, however! **WARNING**: These questions are too hard for their point values. We are free to mark these questions harshly. Finish the rest of the exam before attempting these questions. Do not **taunt** these questions.

1. In "I'm a k, You're $O(k)$", justify each of the following statements: **[2 marks]**

   (a) $k \leq n$
   SOLUTION: k is one more than the maximum displacement of any element from its sorted location. Can we displace by n indexes (or more)?

   (b) If $n \leq 2k+1$, then using `MergeSort` to sort the whole array runs in $O(k \lg k)$ time. (Justify this **carefully** for credit!)
   SOLUTION: It's **very** tempting to say something like $n \in \Theta(k)$ (which is both true and useful!), but this isn't enough in general. (See the next problem.) So, you have to either prove that this is enough (a pain) or take it step by step from `MergeSort`'s bound of $n \lg n$ through steps like $n \lg n \leq (2k+1)\lg(2k+1)$ until you get to a $O(k \lg k)$ bound. (Technically, I should have written something like $cn \lg n$.)

   (c) Give an example of a function $f(n)$ such that $f(n) \notin O(f(k))$ even when $n \in \Theta(k)$.
   SOLUTION: Try $f(n) = 2^n$. (Yes, $f(n)$ **can** be in $O(f(k))$ even in cases like this. Indeed, if you're allowed to **pick** the relationship of n and k, you can just pick $n = k$ (which is consistent with $n \in \Theta(k)$ of course!) and there's **no** breaking example. But... this is a bonus problem. We leave you to figure out that that's not what we were aiming at.)

2. Imagine we have $n \geq 0$ indistinguishable (i.e., look-alike) balls and $k \geq 1$ indistinguishable jars. How many different ways are there to put the balls in the jars? Your solution must run in $O(kn)$ time. **[2 marks]**

   Explanation: Consider $n = 2, k = 2$, that is, two balls and two jars. What we mean by indistinguishable balls is that all you know about a jar is how many balls are in it, not which ones. So, if you have 2 balls and 2 jars, putting ball A in jar A and ball B in jar B does not count as a different way to organize the balls from putting ball B in jar A and ball A in jar B (because you cannot tell the balls apart). Furthermore, the jars are indistinguishable; so, putting both balls in jar A and none in jar B is not distinguishable from putting both balls in jar B and none in jar A (because you cannot tell the jars apart, either). So, with $n = 2, k = 2$, there would only be two possible ways to arrange the balls: both in one jar or one in each jar.

   SOLUTION: This is a delightful problem. We can give several hints.

   First: You can always choose to sort the jars from most to least full. (Any permutation of such an ordering is indistinguishable from that ordering; so, you don't need to bother generating all such permutations.)

   Second: If you sort and find the last jar is empty, can you break the problem down into a subproblem? What's the other possibility for the last jar that you haven't handled?

   Third: If you sort and find that the last jar has at least one ball in it, what does it mean about the number of balls in the other jars? Can you use this to induce a subproblem?

   Finally: "Integer Partition".