

CPSC 320 Midterm #2 Practice Problems

March 2, 2015

- $\sum_{y=1}^x y = \frac{x(x+1)}{2}$, for $x \geq 0$.
- $\sum_{y=1}^x y^2 = \frac{x(x+1)(2x+1)}{6}$, for $x \geq 0$.
- $\sum_{y=0}^x 2^y = 2^{x+1} - 1$, for $x \geq 0$.

For a recurrence like $T(n) = aT(\frac{n}{b}) + f(n)$, where $a \geq 1$ and $b > 1$, the Master Theorem states three cases:

1. If $f(n) \in O(n^c)$ where $c < \log_b a$ then $T(n) \in \Theta(n^{\log_b a})$.
2. If for some constant $k \geq 0$, $f(n) \in \Theta(n^c(\log n)^k)$ where $c = \log_b a$, then $T(n) \in \Theta(n^c(\log n)^{k+1})$.
3. If $f(n) \in \Omega(n^c)$ where $c > \log_b a$ **and** $af(\frac{n}{b}) \leq kf(n)$ for some constant $k < 1$ and sufficiently large n , then $T(n) \in \Theta(f(n))$.

- $f(n) \in O(g(n))$ (big- O , that is) exactly when there is a positive real constant c and positive integer n_0 such that for all integers $n \geq n_0$, $f(n) \leq c \cdot g(n)$.
- $f(n) \in o(g(n))$ (little- o , that is) exactly when for all positive real constants c , there is a positive integer n_0 such that for all integers $n \geq n_0$, $f(n) \leq c \cdot g(n)$.
- $f(n) \in \Omega(g(n))$ exactly when $g(n) \in O(f(n))$.
- $f(n) \in \omega(g(n))$ exactly when $g(n) \in o(f(n))$.
- $f(n) \in \Theta(g(n))$ exactly when $f(n) \in O(g(n))$ and $f(n) \in \Omega(g(n))$.

1 Practice Intro

These problems are meant to be generally representative of our midterm exam problems and—in some cases—may be **very** similar in form or content to the real exam. However, this is **not** a real exam. Therefore, you should not expect that it will fit the predicted exam timeframe or that the questions will be of the appropriate level of specificity or difficulty for an exam. (That is: the real exam may be shorter or longer and more or less vague!)

All of that said, you would benefit tremendously from working hard on this practice exam!

2 Canopticon

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. $T(n) = 2T(n-1) + 1$

leaves root balanced does not apply

2. $T(n) = 9T(\lfloor n/3 \rfloor) + n\sqrt{n}$

leaves root balanced does not apply

3. $T(n) = 30T(\lfloor n/2 \rfloor) + n^n$

leaves root balanced does not apply

4. $T(n) = T(\lfloor 99n/100 \rfloor) + 1$

leaves root balanced does not apply

3 Vain-y Dividi Vici

REPEATED FROM PREVIOUS PRACTICE EXAM: Consider the following recursive algorithm called on an array of integers of length n . (Note: in this particular problem, it is not relevant, but generally if we refer to “fourths” of an array A with length n that is not divisible by 4, the “fourths” of A won’t be exactly length $\frac{n}{4}$, but each will have length either $\lceil \frac{n}{4} \rceil$ or $\lfloor \frac{n}{4} \rfloor$. Typically, this has no effect on the asymptotic analysis.)

CEDI(A):

If the length of A is odd OR half of the length of A is odd:

Return the first element of A

Else:

Note: If we reach here, the length of A is divisible by 4

Let A1 be the 1st fourth of A,

A2 be the 2nd fourth of A,

A3 be the 3rd fourth of A, and

A4 be the 4th fourth of A.

Return CEDI(A2) + CEDI(A4)

1. Give a recurrence $T(n)$ describing the runtime of this algorithm. Be careful to clearly specify both any recursive case(s) and base case(s) and what conditions on the input identify them. To clarify, we’ve started a solution below, but you will need more than the case we have started.

$T(n) = \text{-----}$ (when $n = \text{-----}$)

2. Would a good Ω -bound on the runtime of this algorithm in terms of n be **best** described as a best-case bound, a worst-case bound, or neither? Choose **one** and briefly justify your answer.

3. Give and briefly justify a **good** Ω -bound on the runtime of this algorithm in terms of n .

(Continued on the next page.)

4 Doctoring the Master Theorem

Answer the questions below about this recurrence:

$$T(n) = \begin{cases} 2T(\lfloor \lg n \rfloor) + cn & \text{when } n > 16 \\ 1 & \text{when } n \leq 16 \end{cases}$$

Throughout this problem, you may ignore floors and ceilings.

1. Draw the top three levels of the recursion tree for this recurrence, labeling as much as you can of: the level numbers, the problem size of each node (inside the node), the work at each node (next to the node), the total work per level, and general forms for the work per level and problem size of each node at a level k .

Then, draw a gap and draw in the leaf level, indicating the problem size at that level and, if you can, the level number in terms of n .

(Practically speaking, you should be able to get through labeling: level numbers, problem size at each node, work per node, work per level, and the problem size at the leaf level. The others are difficult, at least from a notational standpoint; “Knuth’s up-arrow” may make fun reading!)

2. Explain why we cannot use the Master Theorem on this recurrence to derive a Θ -bound on the algorithm’s performance.

- Use the Master Theorem to give a Θ -bound on this related recurrence $T_2(n) = 2T_2(n/4) + cn$.
- What can we conclude about asymptotic bounds on $T(n)$ from this Θ -bound on $T_2(n)$? Briefly justify your answer. *Hint:* for sufficiently large n , what do you know about the relationship between $n/4$ and $\lg n$?
- Find a good Θ -bound on the original recurrence (building from your work in the previous part).
- Want more practice? Modify this recurrence (e.g., changing additional work $f(n)$ in $T(n)$ from cn to 1 or n^2). Try a bunch of different recurrences, drawing trees for them and looking for upper- and lower-bounds on their performance.

5 The High Price of Plausible Deniability

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.

1. What is the minimum number of jobs you can perform in an instance containing **no** jobs?
2. Consider an arbitrary job j in some instance with one or more jobs. If we select j to be in a solution, it divides the instance into two **separate** smaller instances. Describe these two instances.
3. Now, complete the divide-and-conquer algorithm **LazyISP on the next page** that takes two arrays **ByStart** and **ByFinish** as input. **ByStart** is sorted by increasing start time while **ByFinish** is sorted by increasing finish time. Each array entry is an object with four fields: **start** is the job's start time, **finish** is its finish time, **sIndex** is the job's index in **ByStart**, and **fIndex** is its index in **ByFinish**. So, **ByStart[1].start** is the first job's start time, **ByStart[1].finish** is its finish time, **ByStart[1].sIndex = 1** since it's in **ByStart[i]**, and **ByFinish[ByStart[1].fIndex]** is the same job stored in the **ByFinish** array.

Since they contain the same jobs, $\text{length}(\text{ByStart}) = \text{length}(\text{ByFinish})$.

```

LazyISP(ByStart, ByFinish):
  If length(ByStart) = 0:

    Return -----

  LazyISPHelper(ByStart, ByFinish, 1, n)

LazyISPHelper(StartTime, FinishTime, minIndex, maxIndex):

  Let CandidateJobs be the list of all jobs j such that
    j.start >= ByStart[minIndex].start and
    j.finish <= ByFinish[maxIndex].finish

  If length(CandidateJobs) = -----:

    Return -----

  Else:

    Choose an arbitrary job j in CandidateJobs

    bestSoFar = length(CandidateJobs)

    Construct a list Options of all jobs k in
      CandidateJobs with the following relationship to j,
      so that exactly one of Options must be chosen:

      -----

    For each job k in Options:

      Let result be

        LazyISPHelper(ByStart, ByFinish,
          -----,
          -----) +

        LazyISPHelper(ByStart, ByFinish,
          -----,
          -----) +

        -----

      If result < bestSoFar:

        bestSoFar = result

    Return bestSoFar

```

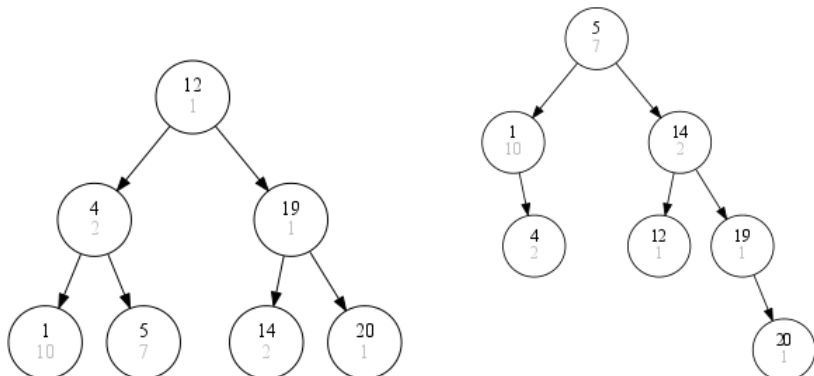

6 The Giving Tree

You're designing an "occasionally self-adjusting" binary search tree. Rather than simply being "balanced", it counts the number of accesses to each key in the tree over the course of a day and then rearranges itself overnight so that that same set of accesses would have the minimum possible total cost the next day.

Specifically: We count the cost of an access to a key as the key's depth in the tree. Given a list of keys and the number of accesses to each key, generate a binary search tree that minimizes the total cost of all accesses (i.e., the sum over all nodes in the tree of the product of the node's depth and its access count). Note: the keys do have associated values, but they'll go with their keys and so are irrelevant to this problem.

So, an instance might be these (key, count) pairs: (1, 10), (4, 2), (5, 7), (12, 1), (14, 2), (19, 1), (20, 1).

Either of these trees would be a solution (with keys in black and their access counts in grey):



However, the cost of the left tree is $(10 * 2) + (2 * 1) + (7 * 2) + (1 * 0) + (2 * 2) + (1 * 1) + (1 * 2) = 43$, while the cost of the right (optimal) tree is: $(10 * 1) + (2 * 2) + (7 * 0) + (1 * 2) + (2 * 1) + (1 * 2) + (1 * 4) = 23$.

Use this insight to develop a solution to the problem: **Some** node must be the root of the tree, and the choice of root divides the remaining problem into two separate pieces.

- Imagine in the example above that you select the 5th node as the root (i.e., the one with the key 14), what two subproblems does this create?

- You will create an algorithm `MakeTree(Nodes)` with helper `MakeTreeHelper(Nodes, lo, hi)` that takes a list of nodes—where `Nodes[k].key` is the node's key and `Nodes[k].freq` is its count of accesses—and the indexes of the left- (in `lo`) and right-most (in `hi`) nodes in the current subproblem and returns the total cost of accessing all the nodes in the optimal tree built from `Nodes[lo..hi]`.

Explain why we need **two** parameters to describe the subproblem rather than one.

3. Now, finish this (radically inefficient!) divide-and-conquer pseudocode for `MakeTree` and `MakeTreeHelper`:

```

MakeTree(Nodes):
    MakeTreeHelper(Nodes, 1, length(Nodes))

MakeTreeHelper(Nodes, lo, hi):

    If _____:

        Return _____

    Else:

        Let bestSoFar be infinity

        For each index i such that lo <= i <= hi:

            Let leftCost be MakeTreeHelper(Nodes, _____, _____)

            Let rightCost be MakeTreeHelper(Nodes, _____, _____)

            Let additionalCost be:

                _____

                _____

            Let totalCost be leftCost + rightCost + additionalCost

            If totalCost < bestSoFar:

                bestSoFar = totalCost

        Return bestSoFar
    EndIf

```

4. Give a recurrence relation $T(n)$ modelling the runtime of this algorithm (but you need not solve the recurrence!).
5. If you convert this algorithm into a dynamic programming approach, what order should you build up solutions?
6. Asymptotically, how many table entries will you need to store all the solutions you compute when solving a problem with n nodes?
7. We'd never ask you this on a real exam because it's too tedious and time-consuming, but it is good practice: Solve the example problem by hand above using dynamic programming. (Notice that a **much** smaller example might be helpful in solving the next two more abstract problems.)

8. Actually rewrite `MakeTree` to use dynamic programming:

Note: We take `Nodes` to be of length `n`

`MakeTree(Nodes)`:

Let `Soln` be a two-dimensional array of length `n x n`,
where `Soln[i][j]` represents the minimum cost of a
BST containing all of `Nodes[i..j]`

Initialize all `Soln[i][j]` for $1 \leq i, j \leq n$ to infinity

9. Given indexes `i` and `j` and the completed array `Soln` from your algorithm, describe how you would find what the best root of a BST containing `Nodes[i..j]`.

7 I'm a k , You're $O(k)$

Suppose that we are given an unsorted array `A` with n distinct elements, and another sorted array `Positions` with k distinct elements chosen from the set $\{1, 2, \dots, n\}$.

In this question, we consider the problem of finding the `Positions[1]`, `Positions[2]`, ..., `Positions[k]` smallest elements of `A`. For instance, if `A` = [15, 3, 19, 12, 16, 21, 18, 10] and `Positions` = [3, 5, 8], then the solution is the array [12, 16, 21] because 12 is the third smallest element of `A`, 16 is the fifth smallest element of `A`, and 21 is the eighth smallest element of `A`.

1. Describe a divide-and-conquer algorithm to compute the solution in $O(n \lg k)$ expected time. You are encouraged to use `RandomizedQuickSelect` (`QuickSelect` with a randomly chosen pivot), which has expected runtime in $O(n)$ for an array of length n (and any order statistic). *Hint:* where might the $\lg k$ term in the running time come from?

2. Prove that your algorithm has an expected running time in $O(n \lg k)$ using recursion trees.

8 I Want the Truth

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 non-empty input arrays A passed to the algorithm: The `DeterministicSelect` algorithm picks the smallest element in the array as its pivot.

always true **never** true **sometimes** true

2. Evaluate this statement over the set of all unordered arrays of distinct integers of length $n > 1$: The divide-and-conquer inversion counting algorithm adds more than $n/4$ to the count of inversions on some step of the merge process.

always true **never** true **sometimes** true

3. Evaluate this statement over the legal instances of the closest pair of points problem with at least four points: Every point in the input is within the 2δ -wide strip around the dividing line on the top-level recursive call to the divide-and-conquer closest pair of points algorithm.

always true **never** true **sometimes** true

4. Evaluate this statement over the legal instances of the closest pair of points problem with at least four points: No more than one point in the input is within the 2δ -wide strip around the dividing line on the top-level recursive call to the divide-and-conquer closest pair of points algorithm.

always true **never** true **sometimes** true

5. Evaluate this statement over the instances of the weighted interval scheduling problem: Running the greedy algorithm for the interval scheduling problem on the instance (with the weights deleted) runs in $O(n \lg n)$ time.

always true **never** true **sometimes** true

6. Evaluate this statement over possible dynamic programming algorithms: The asymptotic runtime of the dynamic programming algorithm is lower-bounded by the asymptotic number of entries in the table used to actually store results.

always true **never** true **sometimes** true

7. Evaluate this statement over divide-and-conquer algorithms where memoization asymptotically improves their performance: Memoized (i.e., already calculated and stored) results are accessed $\omega(1)$ times.

always true **never** true **sometimes** true

This page intentionally left (almost) blank.
If you write answers here, you must **CLEARLY** indicate on this page what question they belong with **AND** on the problem's page that you have answers here.