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 $x \geq 0$.

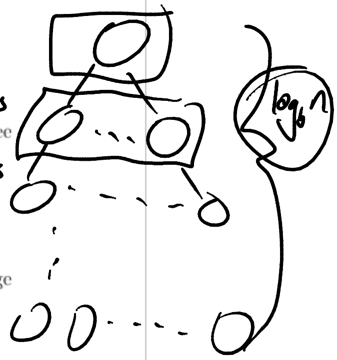
floors/ceilings don't matter

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

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

→ constant-time base cases!

$n^{\log_b a} = a^{\log_b n}$



- $f(n) \in O(g(n))$ (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**.

$\boxed{T(n) = aT\left(\frac{n}{b}\right) + f(n)}$

$a \geq 1$

$b > 1$

leaf: $f(n) \in O(n^c)$ $\rightarrow$ for $c < \log_b a$

balanced: $f(n) \in \Theta(n^c (\log n)^k)$, $c = \log_b a$

root: $f(n) \in \Omega(n^c)$, $c > \log_b a$

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

leaves **root** **balanced** $\boxed{\textbf{does not apply}}$

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

$\in \Theta(n^2)$

a, b, $f(n)$

$\log_b a = \log_3 9 = 2$

$n\sqrt{n} = n \cdot n^{0.5} = n^{1.5}$ n^2

$\boxed{\textbf{leaves}}$ **root** **balanced** **does not apply**

$\log_b a = \log_2 30$ $4 < \log_2 30 < 5$

$n^n \in \Omega(n^c)$ for $c = 5$

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

a, b, $f(n)$

$\in \Theta(n^n)$

$af\left(\frac{n}{b}\right) \leq kf(n)$ for $k < 1$ and suff. large n.

$30f\left(\frac{n}{2}\right) \leq kf(n)$

$30(n/2)^{(n/2)} \overset{?}{\leq} kn^n$ $k = 1/2$

$\left(\frac{n}{2}\right)^{n/2} \leq n^{n-1}$ for $n \geq 2$.

leaves $\boxed{\textbf{root}}$ **balanced** **does not apply**

$aT\left(\frac{n}{b}\right) + f(n)$

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

$1 = n^0$

$T\left(\frac{n}{100/99}\right)$, b

$\log_b a = \log_{\frac{100}{99}} 1 = \boxed{0}$

leaves **root** $\boxed{\textbf{balanced}}$ **does not apply**

$f(n) = n^0 \in \Theta(n^0) = \Theta(n^{\log_b a})$

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) = ____________________    (when n = _____________)`

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. Draw a recursion tree for `CEDI(A)` labeled by the amount of time taken by each recursive call to `CEDI` and the total time for each "level" of calls, both in terms of n for an arbitrary value of n that is **a power of** 4 **greater than 1** (i.e., $n = 4^k$ for $k \geq 1$).

5. Give and briefly justify—based on your tree—a good O-bound on the runtime of this algorithm in terms of n.

6. Briefly explain why your bound from the previous part is **not** a Θ-bound.

7. Briefly explain why we cannot use the Master Theorem to give a Θ-bound on the runtime of this algorithm.

8. If we consider only values of n that are powers of 4, we **can** apply the Master Theorem. Indicate the key parameters of the Master Theorem in this case and use it to re-justify your O-bound.

4 Doctoring the Master Theorem

Answer the questions below about this recurrence:

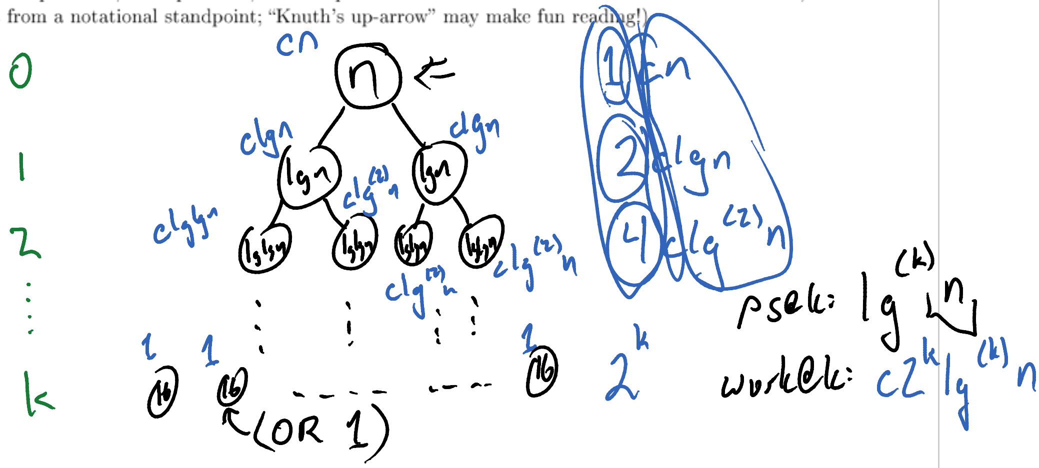
$$T(n) = \begin{cases} 2T(\lfloor \lg n \rfloor) + cn & \text{when } n > 16 \\ 1 & \text{when } n \le 16 \end{cases}$$

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.

Inside of $T(\lfloor \lg n \rfloor)$ is not of the form $\frac{n}{b}$ for $b > 1$.

3. Use the Master Theorem to give a Θ-bound on this related recurrence $T_2(n) = 2T_2(n/4) + cn$.

$\frac{cn}{2} = 2c\frac{n}{4} \le \frac{1}{2}cn$

$af\left(\frac{n}{b}\right) \le kf(n)$

Assume $T(1) = 1$. (So reasonable base case.)

$T(n) = aT\left(\frac{n}{b}\right) + f(n)$

$a = 2$ $\qquad \log_b a = \log_4 2 = \frac{1}{2}$

$b = 4$

$f(n) = cn$ $\qquad f(n) \in \Omega(n^1)$ and $1 > \frac{1}{2}$

(3) $(f(n)) = \boxed{\Theta(n)}$

4. 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 $\lg n$?

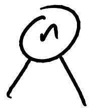
$n/4 \in \omega(\lg n)$ $\qquad T(n) = 2T(\lfloor \lg n \rfloor) + cn$

basically: $T_2(n) \ge T(n)$. non-dec: $T(n_1) \ge T(n_2)$ if $n_1 \ge n_2$.

So the Θ bound on T_2 is an O-bound on T. $\qquad T(n) \in O(n)$

5. Find a good Θ-bound on the original recurrence (building from your work in the previous part).

$T(n) = 2T(\lfloor \lg n \rfloor) + \boxed{cn}$ $\qquad T(n) \in O(n)$



$cn \in \Omega(n)$ $\longrightarrow$ $T(n) \in \Omega(n)$

so $T(n) \in \Theta(n)$

6. 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?

0

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.

LEFT INSTANCE CONTAINS ALL JOBS W/ FINISH TIMES $f_i < s_j$
& RIGHT CONTAINS JOBS W/ START TIMES $s_i > f_j$

3. Now, complete the divide-and-conquer algorithm `LazyIS` **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, `length(ByStart) = length(ByFinish)`.

```
LazyISP(ByStart, ByFinish):
  If length(ByStart) = 0:
    Return ___0___
  LazyISPHelper(ByStart, ByFinish, 1, n)

            ByStart      ByFinish
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) = ___0___:
    Return ___0___
  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:
      ___k conflicts w/ j's times___ (i.e., k is in j's conflict set)
    For each job k in Options:
      Let result be
        LazyISPHelper(ByStart, ByFinish,
          ___minIndex___,
          ___index of the job in ByFinish w/ the largest finish time < k's start time___)   (LEFT)
        + LazyISPHelper(ByStart, ByFinish,
          ___index of the job in ByStart w/ the smallest start time > f_k___,
          ___maxIndex___) +   (RIGHT)
        ___1___
      If result < bestSoFar:
        bestSoFar = result
    Return bestSoFar
```

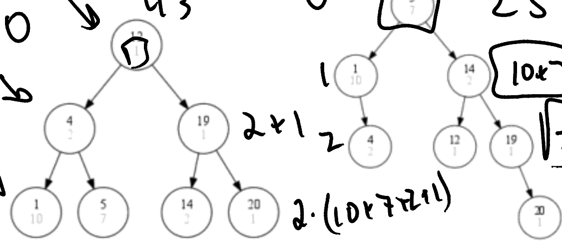
6 The Giving Tree

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 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.

1. 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?

(1,10) (4,2) (5,7) (12,1)
LEFT

(19,1) (20,1)
RIGHT

2. 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.

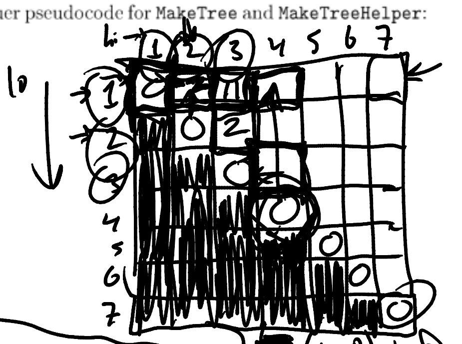
Any arb consecutive seg. w/in the orig problem may be a subproblem & it takes two params to specify all such subsegs.

Nodes[p ... end]
Nodes[1 ... p]

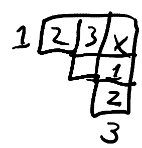
3. Now, finish this (radically inefficient!) divide-and-conquer pseudocode for `MakeTree` and `MakeTreeHelper`:

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

MakeTreeHelper(Nodes, lo, hi):
  If hi-lo+1 = 0:
    Return 0
  Else:
    Let bestSoFar be infinity
    For each index i such that lo <= i <= hi:
      Let leftCost be MakeTreeHelper(Nodes, lo, i-1)
      Let rightCost be MakeTreeHelper(Nodes, i+1, hi)
      Let additionalCost be:
        sum of all freqs of
        Nodes[lo..hi] minus Nodes[i].freq
      Let totalCost be leftCost + rightCost + additionalCost
      If totalCost < bestSoFar:
        bestSoFar = totalCost
    Return bestSoFar
  EndIf
```

BC



(1,10)
(4,2)
(5,7)
(12,1)
(14,2)
(19,1)
(20,1)



$(d+1)a - da$

$da + a - da$

a

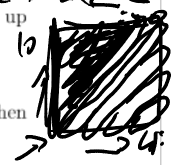
4. Give a recurrence relation $T(n)$ modelling the runtime of this algorithm (but you need not solve the recurrence!).

$T(0) = 1$

$T(n) = \sum_{i=1}^{n} \left(T(i-1) + T(n-i)\right) + \Theta(n) = \left(2\sum^{n-1} T(i)\right) + \Theta(n)$ Let $n = hi - lo + 1$

5. If you convert this algorithm into a dynamic programming approach, what order should you build up solutions?

Shortest problems first



6. Asymptotically, how many table entries will you need to store all the solutions you compute when solving a problem with n nodes?

$\Theta(n^2)$

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.)

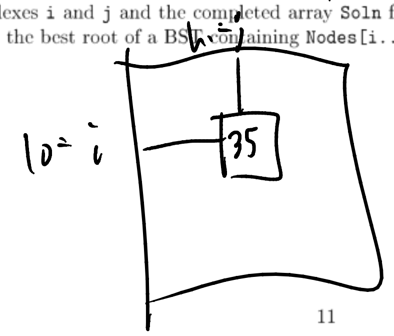
8. Actually rewrite MakeTree to use dynamic programming:

```
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 <= i, j <= n to infinity
```

```
For diag = 1 to n:
  For lo = diag downto 1:
    Soln[lo][diag] =
      min over the set of problems:
        For all lo ≤ i ≤ diag:
          SolnCheck(lo, i-1) +
          SolnCheck(i+1, diag) +
          (Sum over lo≤j≤diag of
            freqs of nodes j) -          Nodes[j].freq
          freq of node i                 Nodes[i].freq

Return SolnCheck(1, n)
```

```
SolnCheck(Soln, i, j):
  If j-i+1 ≤ 0:
    return 0
  Else:
    Return Soln[i][j]
```

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].



REDO COMP + FIND THE "i" FOR WHICH THE CALC'D COST = Soln[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 $\boxed{\{1, 2, \ldots, n\}}$.

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 eigth smallest element of `A`.

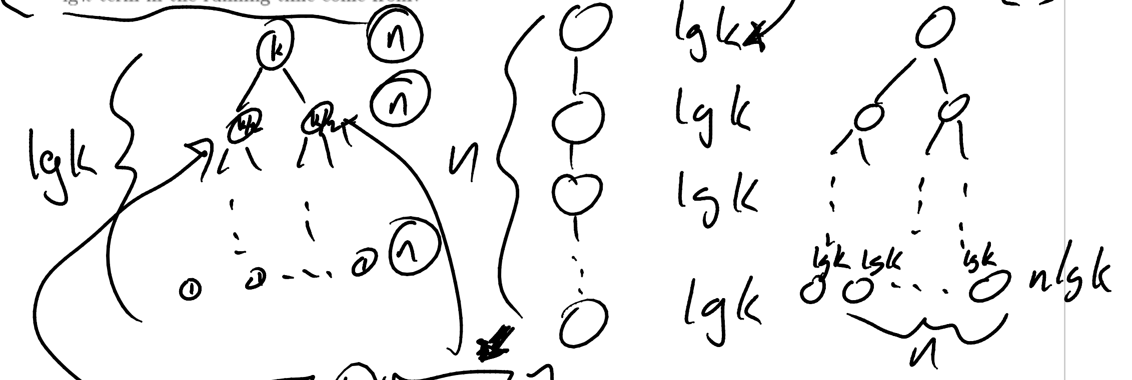
$L = [15, 3, \boxed{12}, 10]$

$G = [19, 21, 18]$

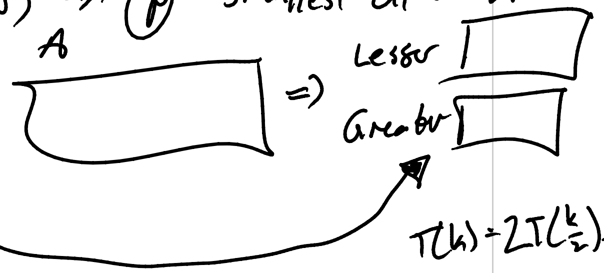
$LP = [3]$

$RP = [8]$

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?



RQS(A, p) ⇒ p^{th} smallest elt of A



$T(k) = 2T(\frac{k}{2}) + \Theta(n)$?!

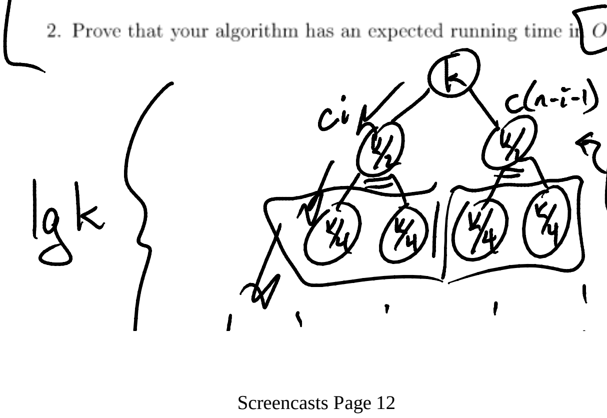
SelectAll(A, Positions): FA(A, Positions, 0)

FindAll(A, Positions, offset):
- If |Positions| = 0:
  - Return []
- Else
  - let mid = |Positions|/2
  - let midItem = RQS(A, Positions[mid] − offset)
  - Partition A around midItem into Lesser + Greater
  - Return FindAll($\boxed{\text{Lesser}}$, $\boxed{\text{Positions [1.. mid-1]}}$, offset) + [midItem] + FindAll($\boxed{\text{Greater}}$, Positions [mid+1..n], |Lesser| + 1 + offset)

O(n)

Θ(n)

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



cn

$ci + c(n-i-1) = ci + cn - ci - 1 = cn - 1 \leq cn$

$\boxed{cn}$

cn

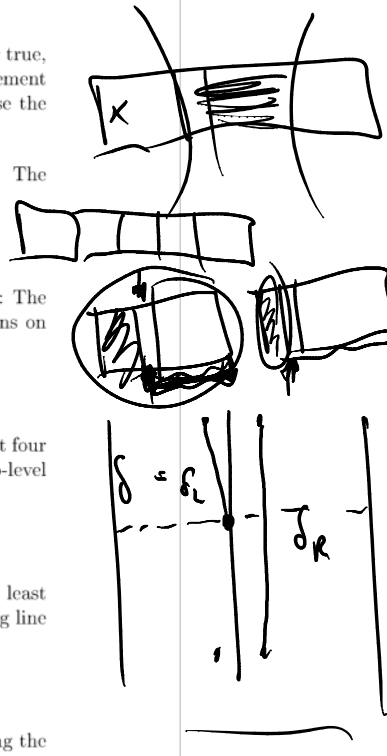
$cn \lg k \in O(n \lg k)$

cn

$cn \lg k \in O(n \lg k)$

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.

   Assume n jobs

   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.