CS 161 W19: Recitation 2 Solutions

January 2019

Exercise 0

Recall from lecture that the algorithm SELECT finds the k^{th} smallest element in an array of n elements in $O(n)$ time. In class, we split the array into groups of 5 elements, and used the "median of medians" as the pivot, as illustrated in the following pseudocode.

Algorithm 1: SELECT(A, k)

```
if len(A) ≤ 50 then
    A = MergeSort(A)
    return A[k-1]
pivot = GetPivot(A)
L, R = Partition(A, pivot)
if len(L) == k − 1 then
    return pivot
else if len(L) > k − 1 then
    return Select(L, k)
else
    return Select(R, k-len(L)-1)
```

Algorithm 2: GETPIVOT(A)

```
Split A into groups of 5 elements each (n/5 groups)
for i ∈ [0, n/5) do
    m_i = Select(group i, 3)
M = [m_1, m_2, ..., m_{n/5}]
return Select(M, n/10)
```

In this problem, we consider what happens to our algorithm if we modify GETPIVOT to use groups of a different size. (To simplify your argument, feel free to ignore rounding issues and floors/ceilings.)

a. Suppose that we instead divide A into $n/3$ groups of size 3, find the median of each group, and then let p be the median of these medians. Derive an upper bound on the number of elements of A that are greater than p. (See the end of the lecture 4 slides, or CLRS Ch. 9 for how this is done with groups of size 5—the proof was skipped in lecture.)

b. Derive a recurrence relation $T(n)$ for the worst-case runtime of SELECT using the modified GETPIVOT with groups of size 3.

c. Using the substitution/"guess and check" method, solve the recurrence from part b to compute the big-O runtime of this version of SELECT. (Hint: The runtime may not be $O(n)$.)

d. Repeat parts a–c where we instead modify GetPivot to divide A into $n/7$ groups of size 7.

e. Why do you think we used groups of size 5 in lecture?

f. Advanced (Take Home) - What happens if the group size is not a constant? For instance, what if we modify GetPivot to divide A into $\sqrt{n}$ groups of size $\sqrt{n}$?

Solution 0

a. First we will lower bound the number of elements of A less than or equal to p, lackadaisically ignoring floors and ceilings. Since p is the median of medians, at least half of the $n/3$ groups have medians less than or equal to p. In each of these groups, there is also one element less than that group's median, so 2 elements of each such group are less than or equal to p. Thus the number of elements less than or equal to p is at least $\frac{1}{2} \times \frac{n}{3} \times 2 = \frac{n}{3}$. Consequently, the number of elements greater than p is at most $n - \frac{n}{3} = \frac{2n}{3}$. (Note that this upper bound also holds for the number of elements less than p, by symmetry.)

b. Our algorithm does the following: (1) compute the median of $n/3$ groups of size 3, (2) compute the median of the $n/3$ medians, (3) partition the elements by comparing to the median of medians, and (4) recurse on a sub-problem of size at most $2n/3$ (by part a). Thus we can say that $T(n) \leq \frac{n}{3}T(3) + T(\frac{n}{3}) + n + T(\frac{2n}{3})$. We then note that $T(3)$ is a constant; in fact, any problem of size at most 50 hits the base case of our Select algorithm, and runs in constant time (let's arbitrarily say at most 100 operations). Thus we obtain

$$\begin{aligned} T(n) &\leq 100 && \text{for } n \leq 50 \\ T(n) &\leq T(\frac{n}{3}) + T(\frac{2n}{3}) + 35n && \text{otherwise.} \end{aligned}$$

c. To hazard a guess at the big-O runtime, let's unwind this recurrence relation by one step:

$$\begin{aligned} T(n) &\leq T(\frac{n}{3}) + T(\frac{2n}{3}) + 35n \\ &\leq \left(T(\frac{n}{9}) + T(\frac{2n}{9}) + 35\frac{n}{3}\right) + \left(T(\frac{2n}{9}) + T(\frac{4n}{9}) + 35\frac{2n}{3}\right) + 35n \\ &= \left(T(\frac{n}{9}) + T(\frac{2n}{9})\right) + \left(T(\frac{2n}{9}) + T(\frac{4n}{9})\right) + 35n + 35n. \end{aligned}$$

Based on this sample size of 2, it looks like maybe we have to do an additional $35n$ operations for every step we unwind. How many times do we have to unwind? Each time we unwind the relation, the largest problem size shrinks by a constant factor $\frac{3}{2} = 1.5$. But the number of times we have to divide n by a constant c to reach 1 is precisely $\log_c(n)$! So, let's guess that we have to unwind about $O(\log n)$ times, and thus let's guess that our runtime is $O(n \log n)$.

Inductive hypothesis: We want to prove that $T(n) \leq cn \log n$ for all $n \geq 2$, for an appropriate constant c to be chosen later. (Note that we choose to start with $n = 2$ simply because $\log 1 = 0$ and thus our hypothesis can't possibly hold for $n = 1$.)

Base case: For $2 \leq n \leq 50$, we have $T(n) \leq 100$ and $n \log n \geq 2$, so the hypothesis holds as long as $c \geq 50$.

Inductive step: Assume that $T(n) \leq cn \log n$ for all $n < k$, and consider $T(k)$. We have

$$\begin{aligned} T(k) &\leq T(\frac{k}{3}) + T(\frac{2k}{3}) + 35k \\ &\leq c\frac{k}{3}\log\frac{k}{3} + c\frac{2k}{3}\log\frac{2k}{3} + 35k \\ &= c\frac{k}{3}(\log k - \log 3) + c\frac{2k}{3}(\log k - \log 3 + 1) + 35k \\ &= ck\log k - ck\log 3 + c\frac{2k}{3} + 35k \leq ck\log k - 0.9ck + 35k. \end{aligned}$$

Therefore $T(k) \leq ck \log k$ provided $0.9ck \geq 35k$, which is certainly satisfied for $c \geq 50$. (If we were writing up this proof for homework, we would rewrite it pretending we'd chosen $c = 50$ from the beginning.)

Therefore $T(n) \leq 50n \log n$ for all $n \geq 2$, and thus $T(n) = O(n \log n)$, as desired.

d. As in part a, let p be the median of medians, this time with group size 7. At least half of the group medians are less than or equal to p, and in each group there are 4 elements less than or equal to that group's median. Thus at least $\frac{1}{2} \times \frac{n}{7} \times 4 = \frac{2n}{7}$ elements of A are less than or equal to p, and therefore at most $n - \frac{2n}{7} = \frac{5n}{7}$ elements are greater than p.

Using the same logic as in part b, we see that $T(n) \leq \frac{n}{7}T(7) + T(\frac{n}{7}) + n + T(\frac{5n}{7})$. Since our base case (when len(A) ≤ 50) is the same as before, we obtain

$$\begin{aligned} T(n) &\leq 100 && \text{for } n \leq 50 \\ T(n) &\leq T(\frac{n}{7}) + T(\frac{5n}{7}) + 16n && \text{otherwise.} \end{aligned}$$

To hazard a guess at the big-O runtime, note that at the top level we do $16n$ operations worth of work, while at the next level (unwinding the recurrence relation once) we will only do $16\frac{n}{7} + 16\frac{5n}{7} = 16\frac{6n}{7} < 16n$ operations. Since it seems like the amount of work decreases below the top level, let's guess that the top level dominates, resulting in an $O(n)$ runtime.

Inductive hypothesis: We want to prove that $T(n) \leq cn$ for all $n \geq 1$, for an appropriate c to be chosen later.

Base case: For $1 \leq n \leq 50$, we have $T(n) \leq 100$, so the hypothesis holds as long as $c \geq 100$.

Inductive step: Assume $T(n) \leq cn$ for all $n < k$, and consider $T(k)$. We have

$$\begin{aligned} T(k) &\leq T(\frac{k}{7}) + T(\frac{5k}{7}) + 16k \\ &\leq c\frac{k}{7} + c\frac{5k}{7} + 16k \end{aligned}$$

This shows that $T(k) \leq ck$ as long as $c\frac{k}{7} \geq 16k$, i.e., as long as $c \geq 112$. (If we were writing up this proof for homework, we would rewrite it pretending we'd chosen $c = 112$ from the beginning.)

Therefore $T(n) \leq 112n$ for all $n \geq 1$, and thus $T(n) = O(n)$, as desired.

e. 5 is the smallest group size that allows us to obtain a runtime of $O(n)$ using this technique. Smaller group sizes are easier to work with, and if the group size is too big the constants also get large even though the runtime remains linear (finding the median of larger groups takes longer).

Fun fact 1: Although we saw above that a group size of 3 results in a runtime of $O(n \log n)$, it is not known whether the runtime is $\Omega(n \log n)$. It *is* possible to slightly modify the SELECT

algorithm to work with group sizes of 3 or 4 in linear time. [Ke Chen and Adrian Dumitrescu, "Select with Groups of 3 or 4", WADS'15]

Fun fact 2: The paper which originally introduced the SELECT algorithm used groups of size 21! [Manuel Blum et al., "Time bounds for selection", Journal of Computer and System Sciences 1973]

f. Let the group size be g and let p be the median of medians. At least half of the n/g group medians are less than or equal to p, and in each group there are $\frac{g+1}{2}$ elements less than or equal to that group's median. Thus at most $n - \frac{1}{2} \times \frac{n}{g} \times \frac{g+1}{2} = n\frac{3g-1}{4g} \leq \frac{3n}{4}$ elements of A are greater than p.

Analogous to the previous parts, we can write a recurrence relation for the runtime of this algorithm as $T(n) \leq \frac{n}{g}T(g) + T(\frac{n}{g}) + n + T(\frac{3n}{4})$. However, at this point we have a problem, because g and therefore $T(g)$ are super-constant. For instance, if $g = \sqrt{n}$, we get

$$T(n) \leq \left(\sqrt{n}+1\right) T(\sqrt{n}) + T(\frac{3n}{4}) + n.$$

This definitely can't result in a linear big-O bound. Consider $T'(n) = \sqrt{n}T(\sqrt{n}) + n$. Clearly $T'(n)$ is strictly less than our bound for $T(n)$ (let's define it to have the same base cases). However, $T'(n)$ does n operations on each level and has $\log\log n$ levels, so $T'(n) = \Theta(n \log\log n)$. From this we can deduce that $T(n)$ will give us a super-linear big-O bound. (In fact $T(n) = O\left(n \log^2 n\right)$.)

Exercise 1

You are given a collection of n differently-sized light bulbs that have to be fit into n flashlights in a dark room. You are guaranteed that there is exactly one appropriately-sized light bulb for each flashlight and vice versa; however, there is no way to compare two bulbs together or two flashlights together as you are in the dark and can barely see! (You are, however, able to see where the flashlights and light bulbs are.) You can try to fit a light bulb into a chosen flashlight, from which you can determine whether the light bulb's base is too large, too small, or is an exact fit for the flashlight. If the bulb fits exactly, it will flash once, in which case you have a correct match. (Note that the flashing light does not allow you to visually compare bulbs/flashlights to other bulbs/flashlights.)

Suggest a (possibly randomized) algorithm to match each light bulb to its matching flashlight. Your algorithm should run strictly faster than quadratic time in expectation. Give an upper bound on the worst-case runtime, then prove your algorithm's correctness and expected runtime.

Solution 1

Consider the following procedure for matching light bulbs with their corresponding flashlights. If the cardinalities of L and F are equal to 1, then we know that $\ell \in L$ matches $f \in F$, so we can match and return. Otherwise, we run the following recursive procedure:

Match(L, F):

- Select a lightbulb $\ell \in L$ uniformly at random
- For every flashlight $f \in F$: test whether ℓ is too small, too big, or just right (call the matching flashlight f^*)
- For every other lightbulb ℓ': test whether ℓ' is too big or too small to fit into f^*
- F_{big}, $F_{\text{small}} \leftarrow$ flashlights too big and too small for ℓ, respectively

- L_{big}, $L_{\text{small}} \leftarrow$ lightbulbs too big and too small for f^*, respectively
- Match(L_{big}, F_{big})
- Match(L_{small}, F_{small})

Correctness: Consider a given call to Match. For the lightbulb we pick at random, ℓ, we go through all the flashlights in F, so we are guaranteed to find its unique matching flashlight, f^*. Note that if a bulb is too big to fit in f^*, then it must fit in a flashlight that was too big for ℓ; likewise, if a bulb is too small to fit in f^*, then it must fit in a flashlight that was too small for ℓ. Thus, we can partition the bulbs and flashlights simultaneously, such that we only have to compare "small" bulbs to "small" flashlights and "big" bulbs to "big" flashlights. Thus, our recursive calls will correctly match the remaining bulbs to their corresponding flashlights.

Runtime: Note that at each level, we perform a linear amount of work: we go through each flashlight and each bulb and then partition the bulbs and flashlights accordingly. Then, we recurse on the big and small groups. Thus, our runtime will be

$$T(n) \leq T(|L_{\text{big}}|) + T(|L_{\text{small}}|) + cn$$

for some constant c. Note that because we pick ℓ uniformly at random, and the bulbs/flashlights are distinct sizes, this recurrence is exactly the same as the Quicksort recurrence. Thus, our algorithm has expected $O(n \log n)$ runtime and worst-case $O(n^2)$ runtime. (Note: Using randomness allows us to get an expected runtime of $O(n \log n)$ on EVERY input. Otherwise, there might be "bad" inputs which run in $\Omega(n^2)$ time.)

Exercise 2

Suppose you're given n distinct ordered pairs of integers $(x_1, y_1), (x_2, y_2), ..., (x_n, y_n)$, where for all i, j, $x_i \neq x_j$ and $y_i \neq y_j$. Recall that two points uniquely define a line, $y = mx + b$, with slope m and intercept b. (Note that choosing m and b also uniquely defines a line.) We say that a set of points S is *collinear* if they all fall on the same line; that is, for all $(x_i, y_i) \in S$, $y_i = mx_i + b$ for fixed m and b. In this question, we want to find the maximum cardinality subset of the given points which are collinear. Assume that given two points, you can compute the corresponding m and b for the line passing through them in constant time, and you can compare two slopes or two intercepts in constant time.

a. Design an algorithm to find a maximum cardinality set of collinear points in $O(n^2 \log n)$ time. If there are several maximal sets, your algorithm can output any such set. Since we haven't covered hashing yet, your algorithm should not use any form of hash table.

b. It is not known whether we can solve the collinear points problem in better than $O(n^2)$ time. But suppose we know that our maximum-cardinality set of collinear points consists of exactly n/k points for some constant k. Design a randomzed algorithm that reports the points in some maximum-cardinality set in expected time $O(n)$. *(Hint: Your running time may also be expressed as $O(k^2 n)$.)* Prove the correctness and runtime of your algorithms.

c. Is your algorithm from part b guaranteed to terminate?

Solution 2

a. Consider the following procedure:

- For all pairs of points, compute their slope and intercept (m, b)

- Sort these pairs lexicographically
- Iterate through the pairs, and note where the longest run of identical (m, b) pairs occurs
- Return a list of the points in this run of pairs

We claim this procedure finds the maximum cardinality set of collinear points in $O(n^2 \log n)$ time.

Correctness: We know that a line is defined uniquely by its slope and intercept. Thus, if two pairs of points give the same slope and intercept, all of the points in the pairs are collinear. If we sort pairs by their resulting (m, b), we know that all paris of points with identical (m, b) values will be adjacent. Each set of k collinear points will have $\binom{k}{2}$ adjacent (m, b) pairs, so the largest set will correspond to the maximum cardinality set of collinear pionts.

Runtime: We know there are $\binom{n}{2} = O(n^2)$ pairs of points. For a given pair of points, we can compute the slope and intercept in $O(1)$ time. Moreover, because we can compare (m, b) pairs in $O(1)$ time, we can run any comparison-based sorting algorithm to sort the (m, b) pairs in $O(n^2 \log n^2) = O(n^2 \log n)$ time.

b. Consider repeating the following procedure until success:

- Sample two points uniformly at random
- Compute their (m, b)
- count $\leftarrow 2$
- For all other points (x_i, y_i): if $y_i = mx_i + b$: count $\leftarrow$ count+1
- if count $= n/k$: SUCCESS

We claim this procedure will find the maximum-cardinality set of collinear points with constant probability, so the expected number of repetitions needed is also a constant.

Correctness: We are guaranteed the maximum-cardinality set has n/k collinear points. Each iteration, we sample two points and find the corresponding line $y = mx + b$. Then, we check for every other point, if the point is on the line. We repeat this process until we find a set of points with n/k collinear points, so we will repeat this procedure until we find the maximum-cardinality set of collinear points.

Runtime: In each iteration, we sample two points in constant time and then go through every other point. Thus, each iteration will take $O(n)$ time. The question that we must ask is how many iterations do we need in expectation until we find the right line. We know that n/k points are on the line with the maximum number of points. Thus, if we sample two points from all the points uniformly at random, the probability of selecting two points on the line is

$$\begin{aligned}
\frac{n/k}{n} \cdot \frac{n/k - 1}{n - 1} &= \frac{1}{k} \cdot \frac{\frac{n-k}{k}}{n-1} \\
&= \frac{1}{k^2} \cdot \frac{n-k}{n-1} \\
&\leq \frac{1}{k^2} \\
&= O(k^{-2}).
\end{aligned}$$

We can view the number of iterations our algorithm takes as a geometric random variable—flipping a coin with probability $p = O(k^{-2})$ until we get a success. The expected value of a geometric

random variable is $\frac{1}{p} = O(k^2)$. Thus, our overall expected runtime is $O(k^2 n) = O(n)$ because k is a fixed constant.

c. The algorithm is not guaranteed to terminate but the probability of failure decreases exponentially.