Design and Analysis of Algorithms
Georgia Tech, Spring 2014

Homework 2

Instructor: Chris Peikert

1. *Median and order statistics.*

   (a) In class we saw an order statistic algorithm that broke up its input into groups of 5, partitioned around the median-of-medians, and recursed. This gave a runtime recurrence of

   $$T(n) \le T(n/5) + T(7n/10) + O(n),$$

   which solved to $T(n) = O(n)$. What recurrences and solutions would we get if we instead broke up the input into groups of 3? Groups of 7?

   Solution: With groups of 3, the runtime of the finding the median of medians would be $T(n/3)$, and $(1/2)\cdot(2n/3) = n/3$ elements can be safely eliminated with each call. Thus, the runtime of the recursive call would be $T(2n/3)$. There is also $O(n)$ additional work (partitioning, finding the medians of the groups of 3) that needs to be done. So the worst-case runtime recurrence is $T(n) = T(n/3) + T(2n/3) + O(n)$, which solves to $T(n) = O(n \log n)$ by a recursion tree.

   With groups of 7, the runtime of the finding the median of medians would be $T(n/7)$, and $(1/2)\cdot(4n/7) = 2n/7$ elements can be safely eliminated with each call. Thus, the runtime of the recursive call would be $T(5n/7)$. There is also $O(n)$ additional work that needs to be done. So the worst-case runtime recurrence is $T(n) = T(n/7) + T(5n/7) + O(n)$, which solves to $T(n) = O(n)$ by a recursion tree.

   (b) Do CLRS Exercise 9.3-5.

   Solution: The algorithm takes a (sub)array A of length n and an index k. If $k = \lfloor n/2 \rfloor$, then the median element is desired, so we just call the given algorithm and return its answer. Otherwise, use the given algorithm to find the median, and partition around the median into left and right subarrays A_1 and A_2, where A_1 contains (in some order) all $\lfloor n/2 \rfloor - 1$ elements smaller than the median, and A_2 contains all $n - \lfloor n/2 \rfloor - 1$ of the elements larger than the median. If $k < \lfloor n/2 \rfloor$, recurse on A_1 and k; otherwise, recurse on A_2 and $k - \lfloor n/2 \rfloor$.

   This algorithm exactly follows the "template" for selection algorithms shown in class, so it is correct. For runtime analysis, since the algorithm calls the given linear-time median algorithm plus partition, it does $O(n)$ work plus the recursive call on an array of half the size. Thus yields a worst-case runtime $T(n) = T(n/2) + O(n)$, which solves to $T(n) = O(n)$ as desired.

   (c) Do CLRS Exercise 9.3-6. Note that the desired runtime of $O(n \log k)$ is better than the $\Theta(nk)$ runtime you'd get by making k separate calls to a $\Theta(n)$-time order statistic algorithm. *Hint:* If it helps, you may assume that k is an exact power of 2.

   Solution: The algorithm takes a (sub)array $A[i \ldots j]$ and a desired quantile number k, which we assume is a power of two. It works as follows: find the median element m of the subarray and partition the subarray around it. Then if $k = 2$, simply return m. Otherwise, recurse on the left subarray $A[i \ldots (i + j)/2]$ with quantile number $k/2$, and on the right subarray

$A[(i+j)/2+1\dots j]$ with quantile number $k/2$. Output the $k/2-1$ quantiles returned by the first call, followed by m, followed by the $k/2-1$ quantiles returned by the second call. (This is a total of $k-1$ quantiles, as required.)

The algorithm is correct in the base case $k=2$, because it simply returns the median. For $k>2$, it is correct because the left/right halves of the array contain those elements smaller/larger than the median, and the recursive calls return the correct quantiles of those subarrays by induction.

The worst-case runtime of the algorithm (on a subarray of size n with quantile number k) satisfies the recurrence $T(n,k)=2T(n/2,k/2)+O(n)$, because each recursive call is on $n/2$ elements with quantile number $k/2$. The base case is $T(n,2)=O(n)$ for any n, because we can find the median and partition in $O(n)$ time. A recurrence tree shows that every level of the tree corresponds to $c\cdot n$ work, and there are $\log_2 k$ levels, for a solution of $T(n,k)=O(n\log k)$.

(d) *Extra credit:* Do CLRS Exercise 9.3-7.

Solution: Calculate the median m of S in $O(n)$ time. Then, create an array D of the absolute differences between each element of S and the median, using $D[i]=|S[i]-m|$, in $O(n)$ time. Note that the k smallest elements of D correspond to the k numbers in S that are closest to the median (which we want to find). So, we select the kth smallest element d of D in $O(n)$ time. Then pass through D, outputting all its elements that are within d of m; these elements are exactly those k numbers in S that are closest to the median. Each of the four main steps takes $O(n)$ time, so the total runtime is $O(n)$.

2. Do CLRS Exercise 9.3-8. *Hint:* think divide-and-conquer, and how you can rule out large subarrays of X and Y that cannot contain the desired median element.

Solution: The algorithm takes two (sorted) subarrays $X[i\dots j]$ and $Y[p\dots q]$ of equal size, starting from the whole arrays. It returns the median element of the combined arrays, as follows. As a base case, if the lengths of the X- and Y-subarrays are both one, return the sole element of the X-subarray. Otherwise, let mx be the median of the X-subarray and my be the median of the Y-subarray (which can both be found in $O(1)$ time since X,Y are sorted). If $mx=my$, return mx. If $mx<my$, recurse on the right half of the X-subarray and the left half of the Y-subarray. If $mx>my$, recurse on the left half of the X-subarray and the right half of the Y-subarray.

For correctness, the algorithm is clearly correct in the base case. When $mx=my$, there are exactly $n/2+n/2=n$ elements that are less than or equal to mx, so mx is the desired median of all $2n$ elements. When $mx<my$, there are less than n total elements smaller than mx, and less than n elements larger than my. So the desired median is between mx and my. Moreover, since we remove from consideration the *same* number of elements that are smaller and larger than the desired median, it is actually the *median* of the two subarrays we recurse upon, and the recursive call returns this median by induction. Similar reasoning applies in the case $mx>my$. Therefore, the algorithm is correct.

The runtime analysis is as follows: since mx,my can be found in $O(1)$ time, and the recursive call is on subarrays that are half the size of the input subarrays, the worst-case runtime recurrence is

$T(n) = T(n/2) + O(1)$, which solves to $T(n) = O(\log n)$ as desired.

3. *Doing your homework.* Imagine (!) that you have a homework assignment of N problems, where each problem i is worth a certain number of points $P_i > 0$, and will take some positive integer T_i minutes to solve. You have only T minutes to spend on the assignment altogether, and want to earn as many points as possible (there is no partial credit).

   (a) An obvious "greedy" strategy is to do the problems in non-increasing order of their "relative value" P_i/T_i, until you run out of time. Give a concrete example (i.e., specific values of T, and some P_is and T_is) showing that the greedy strategy does *not* necessarily maximize the number of points earned.

   Solution: We construct an example that can be arbitrarily bad. Consider two questions where $P_1 = 1 + \epsilon, T_1 = 1$ and $P_2 = T - 1, T_2 = T - 1 + \epsilon$ for some small $\epsilon > 0$, where $T \gg 2$ is the total time budget. With the greedy strategy, we will first do question 1 but then will not have enough time to do question 2. But doing question 2 alone will yield many more points.

   (b) In the remaining parts, you will devise an efficient algorithm that determines which homework problems to do, so as to maximize the number of points earned.

   Let $M(j, t)$ denote the most points you can obtain by doing some subset of problems 1 though j *only*, within a total time budget of t. Give the simplest recursive equation you can for $M(j, t)$ in terms of the given P_i and T_i values and other values of $M(\cdot, \cdot)$, along with the base cases. Justify your answers.

   Solution: An optimal subset of problems 1 through j either includes question j, or it doesn't. If we skip question j (which we must do if $t < T_j$), then the best we can do is to choose an optimal subset of questions 1 through $j - 1$ with our remaining time budget of t. If we do solve problem j (assuming $t \geq T_j$), then we earn P_j points and then the best we can do is to solve an optimal subset of problems 1 through $j - 1$ with our remaining time budget of $t - T_j$. A globally optimal subset of problems 1 through j must be the best between these two scenarios.

   By the above reasoning, we have the following recurrence:

$$M(j,t) = \begin{cases} \max\{M(j-1,t)\ ,\ P_j + M(j-1, t-T_j)\} & \text{if } t \geq T_j, \\ M(j-1,t) & \text{if } t < T_j. \end{cases}$$

   The base cases are $M(0, \star) = M(\star, 0) = 0$.

   (c) Using the above recurrence, describe an algorithm that, given all the P_i, T_i values and a maximum possible time budget T, prepares a data structure that can be used to answer queries of the form: "if I have only t minutes (where $t \leq T$), what subset of all the problems will maximize the number of points I earn?" Also describe the algorithm that answers such queries using the data structure. Analyze the runtime and space requirements of your algorithms, and make them as small as you can.

Solution: A dynamic programming algorithm for the preparation phase is straightforward: fill in a table $M[j,t]$ according to the recurrence from the previous part, for each $j = 0, 1, \ldots, N$ and each $t = 0, 1, \ldots, T$. To make sure that each entry of the table is filled in before it is needed, we fill in all the values $M[j, \star]$ before moving on to $M[j+1, \star]$. By design of the recurrence, filling each table entry takes only constant time, so the total runtime is $O(NT)$.

To answer a query given t and the M table, we start at $M[N,t]$ and recompute (in $O(1)$ time) whether problem N is included in an optimal choice of problems, update the value of t to t' appropriately (depending on whether problem N was included or not). We then do the same for $M[N-1,t']$ etc., tracing back through the table to get a set of optimal questions. This takes only $O(N)$ time.

4. Recall that a *subsequence* of an array $A[1 \ldots n]$ is a sequence of elements $A[i_1], A[i_2], \ldots, A[i_k]$ where $1 \le i_1 < i_2 < \cdots i_k \le n$, and it is *non-decreasing* if $A[i_1] \le A[i_2] \le \cdots \le A[i_k]$. Give an $O(n^2)$-time dynamic programming algorithm to find the longest non-decreasing subsequence in a given array $A[1 \ldots n]$ of arbitrary real numbers. Justify its correctness and claimed runtime.

Solution: For shorthand, let NDS be short for "non-decreasing subsequence," and let LNDS be short for "longest NDS."

To help us devise a dynamic programming algorithm, let's first investigate the structure of LNDSs. Let $A[i_1], \ldots, A[i_k]$ for $k \ge 1$ be an LNDS of A. This subsequence ends with some array element $A[i_k]$, and if $k > 1$, the remaining elements $A[i_1], \ldots, A[i_{k-1}]$ form an NDS that ends with $A[i_{k-1}] \le A[i_k]$. We claim that this subsequence must be an *LNDS that ends with* $A[i_{k-1}]$. For if not, we could replace it with a longer NDS ending in $A[i_{k-1}]$, and then append $A[i_k] \ge A[i_{k-1}]$ to get a longer NDS (of the whole array A) than the one we started with, contradicting its assumed optimality.

The above observations lead us to the kinds of subproblems we should solve in a dynamic programming algorithm. Define $E[i]$ to be the length of an LNDS of A *that ends with* $A[i]$. Since an LNDS must end with some element of A, the maximum value of $E[i]$ over all indices i will be the length of an LNDS of A. Moreover, by the claim from the previous paragraph, we can write a nice recurrence for $E[i]$: since an LNDS ending with $A[i]$ must either be $A[i]$ alone, or be $A[i]$ preceded by an LNDS ending with some $A[j]$ where $j < i$ and $A[j] \le A[i]$, we have the recurrence

$$E[i] = 1 + \max_{\substack{j<i \\ A[j] \le A[i]}} E[j],$$

where the max expression is understood to be zero if no values of j satisfy the conditions.

We can easily implement a dynamic programming algorithm that calculates $E[1], E[2], \ldots, E[n]$ via the above recurrence. To calculate $E[i]$ we need to look at $i - 1 \le n$ previous values and do constant work for each, so the overall runtime is $O(n^2)$ (in fact it is $\Theta(n^2)$). To actually *find* an LNDS of A, we first find a maximal value of $E[i]$, which indicates that $A[i]$ is a final element of an LNDS. We then backtrack through E, successively finding an index $j < i$ such that $A[j] \le A[i]$ and $E[j]$ is maximized, and prepend $A[j]$ to our sequence. This takes $O(n^2)$ time as well.

Alternatively, if when filling in each $E[i]$ value we also remember a corresponding index $j < i$ that maximizes the above expression, then we can implement backtracking in just $O(n)$ time by just following those stored j indices.