CS 170 Algorithms Fall 2014 David Wagner Sol 3

1. (10 pts.) Practice with recurrence relations

Solve the following recurrence relations. Express your answer using $\Theta(\cdot)$ notation. (For instance, if you were given the recurrence relation $T(n) = T(n-1)+3$, the solution would be $T(n) = \Theta(n)$. $T(n) = O(n^2)$ would receive no credit, even though it is a valid upper bound, as it is not the best upper bound.) You don't need to show your work or justify your answer for this problem.

(a) $F(n) = F(n-4)+1$.

Solution: Expanding the recurrence gives $\Theta(n/4) = \Theta(n)$ terms, each of which is 1. So $F(n) = \Theta(n)$.

(b) $G(n) = G(n/2)+3$.

Solution: By the master theorem with $a = 1$, $b = 2$, $c = \log_2 1 = 0$ and $d = 0$, we have $G(n) = \Theta(\log n)$.

(c) $H(n) = H(n/4)+1$.

Solution: By the master theorem with $a = 1$, $b = 4$, $c = \log_4 1 = 0$ and $d = 0$, we have $G(n) = \Theta(\log n)$.

(d) $I(n) = I(n/2)+n$.

Solution: By the master theorem with $a = 1$, $b = 2$, $c = \log_2 1 = 0$ and $d = 1$, we have $G(n) = \Theta(n)$.

(e) $J(n) = 2J(n/4)+6$.

Solution: By the master theorem with $a = 2$, $b = 4$, $c = \log_4 2 = 1/2$ and $d = 0$, we have $G(n) = \Theta(\sqrt{n})$.

(f) $K(n) = 2K(n/4)+n$.

Solution: By the master theorem with $a = 2$, $b = 4$, $c = \log_4 2 = 1/2$ and $d = 1$, we have $G(n) = \Theta(n)$.

(g) $L(n) = 2L(3n/4)+1$.

Solution: By the master theorem with $a = 2$, $b = 4/3$, $c = \log_{4/3} 2 > 1$ and $d = 1$, we have $G(n) = \Theta(n^{\log_{4/3} 2}) = \Theta(n^{2.41\cdots})$.

(h) $M(n) = 2M(n-1)+1$.

Solution: Expanding the recurrence gives $M(n) = \sum_{i=0}^{n-1} 2^i = \Theta(2^n)$.

2. (15 pts.) Procedural Terrain Generation

Recursive algorithms can be useful for generating objects with fractal structure, like realistic rocky terrain. We can store the shape of the ground as an $(n+1) \times (n+1)$ array of height values, so that $H[x][y]$ is the height at point (x,y). Say the heights at the corners $(0,0)$, $(0,n)$, $(n,0)$, and (n,n) have been given, and we need to fill in the rest of the grid.

Consider the following algorithm, more complex versions of which are the basis of the terrain generation procedures in many video games. The function Rand() returns a random number and takes constant time.

Algorithm MidpointDisplacement(n):
1. Call FillIn$(0,n,0,n)$.

Algorithm FillIn(ℓ,r,t,b):
1. If $r-\ell < 2$ or $b-t < 2$, return.
2. Let $x := \lfloor(\ell+r)/2\rfloor$.
3. Let $y := \lfloor(t+b)/2\rfloor$.

4. Set $H[x][y] := (H[\ell][t] + H[\ell][b] + H[r][t] + H[r][b])/4 + \text{Rand}()$.
5. Set $H[\ell][y] := (H[\ell][t] + H[\ell][b])/2$.
6. Set $H[x][t] := (H[\ell][t] + H[r][t])/2$.
7. Set $H[r][y] := (H[r][t] + H[r][b])/2$.
8. Set $H[x][b] := (H[\ell][b] + H[r][b])/2$.
9. Call FillIn(ℓ, x, t, y).
10. Call FillIn(ℓ, x, y, b).
11. Call FillIn(x, r, t, y).
12. Call FillIn(x, r, y, b).

(a) What is the asymptotic runtime of MidpointDisplacement(n), as a function of n?

Solution: As usual we may assume that n is a power of 2 to simplify our calculations without affecting the asymptotic runtime. For any $k \geq 2$ which is a power of 2, if FillIn() is called with $r - \ell = b - t = k$, it makes four recursive calls with $r - \ell = b - t = k/2$. If $k \leq 1$ the recursion terminates, and the non-recursive part of FillIn() takes constant time. So writing $T(k)$ for the runtime of FillIn(ℓ, r, t, b) for any ℓ, r, t, b with $r - \ell = b - t = k$, we have

$$T(k) = 4T(k/2) + \Theta(1).$$

By the master theorem, $T(k) = \Theta(k^2)$. Since MidpointDisplacement(n) calls FillIn() with $r - \ell = b - t = n$, its asymptotic runtime is $\Theta(n^2)$.

(b) Is there an algorithm which computes the same values for H as MidpointDisplacement() but which is asymptotically faster? Why or why not?

Solution: No. Any such algorithm has to fill in all of the array H except its corners, and H has $\Theta(n^2)$ entries. So the algorithm must take $\Omega(n^2)$ time.

3. (20 pts.) Dominated?

You are given a collection of n points (x_i, y_i) in two dimensions. Design an $O(n \lg n)$ time algorithm to check whether there exists a pair of points $(x_i, y_i), (x_j, y_j)$ such that $x_i \leq x_j$ and $y_i \leq y_j$ and $i \neq j$.

Solution: Let us say that a point (x, y) *dominates* a point (a, b) if $a \leq x$ and $b \leq y$. Consider the following algorithm:

Main idea: We will sort the points by x-coordinate, then use divide-and-conquer. To check whether a point in the right half dominates a point in the left half, we need only compare their y-coordinates, so it suffices to find the point on the left with minimum y-coordinate and the point on the right with maximum y-coordinate. In fact, if we sort the points just right, it suffices to look at the last point on the right and the first point on the left (the proof is below).

Pseudocode:

Algorithm Dominated$(A[0 \ldots n-1])$:
1. Sort A in lexicographic order (by increasing x-coordinate, breaking ties with the y-coordinate).
2. Call Helper(A), and return its result.

Algorithm Helper$(A[0 \ldots m-1])$:
1. If $m \leq 1$, return false.
2. Let $h := \lfloor m/2 \rfloor$.
3. Call Helper$(A[0 \ldots h-1])$, and return true if it does.
4. Call Helper$(A[h \ldots m-1])$, and return true if it does.
5. Return whether $A[h]$ dominates $A[h-1]$.

Proof of correctness: We will use proof by strong induction on n. The algorithm is clearly correct when $n \leq 1$, since we do not have two distinct points. Now take any $n \geq 2$, and suppose that the algorithm is correct for all lists of points with fewer than n elements. Then given n points, the algorithm computes $h = \lfloor n/2 \rfloor$. Define $L = A[0 \ldots h-1]$ and $R = A[h \ldots m-1]$. Since $n \geq 2$, both L and R have strictly fewer than n elements, and so the recursive calls in lines 3 and 4 will behave correctly by hypothesis. So if one point in L dominates another, or one point in R dominates another, we will return true. Otherwise, no point in either of these subarrays dominates another in the same subarray. Then $A[h-1]$ has the smallest y-coordinate of any point in L, since by lexicographic order it has the largest x-coordinate and so would dominate any point in L with smaller y-coordinate. By an analogous argument, $A[h]$ has the largest y-coordinate of any point in R. So if any point in R dominates a point in L, then $A[h]$ dominates $A[h-1]$, and the algorithm returns true. Since no point in L can dominate a point in R, the algorithm returns true if and only if a point in A dominates another, and so the algorithm is correct for all lists of size n. By induction, the algorithm is correct for all inputs.

Running time: The sorting step of the algorithm can be done in $O(n \log n)$ time, e.g. with Mergesort. The non-recursive part of the Helper function clearly takes constant time, so if $T(m)$ is the runtime of Helper() on an array of length m, we have $T(m) = 2T(m/2) + O(1)$ (making our usual assumption that m is a power of two). By the master theorem with $a = 2$, $b = 2$, $c = \log_2 2 = 1$, and $d = 0$, we have $T(m) = O(m)$. So the overall algorithm takes $O(n \log n)$ time.

Alternate Solution #1: In case you didn't realize that $A[h]$ and $A[h-1]$ are the points you need to compare, an alternate solution is to do linear-time sweeps to find the point in L with least y-coordinate and the point in R with greatest y-coordinate, and check them for dominance. Then you'll get the recurrence $T(m) = 2T(m/2) + O(m)$, which solves to $T(m) = O(m \log m)$, so the total running time will be $O(n \log n)$. This is a fine solution, too.

Alternate Solution #2: Observing that the algorithm above only ever compares adjacent points in the sorted version of A, we can eliminate the recursion and get the following simpler algorithm:

Algorithm Dominated($A[0 \ldots n-1]$):
1. Sort A in lexicographic order.
2. For $i := 1, \ldots, n-1$:
3. If $A[i]$ dominates $A[i-1]$, return true.
4. Return false.

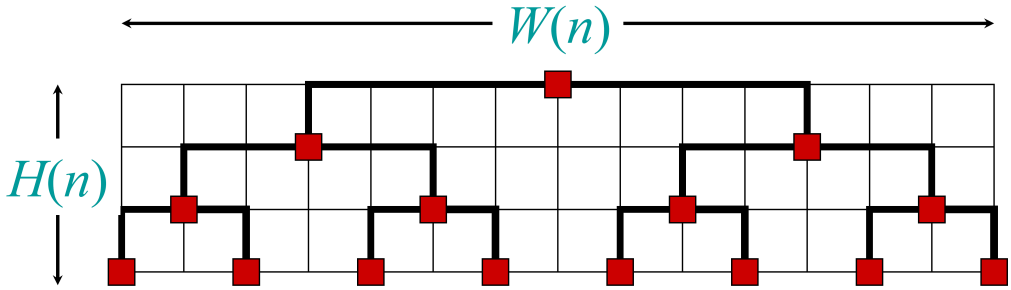
To see why this works, suppose that one point in A dominates another. Then after sorting there must be some $i < j$ such that $A[j]$ dominates $A[i]$, since by lexicographic order if $A[i]$ dominates $A[j]$ with $i < j$ then $A[i] = A[j]$. Let k be the largest index below j such that $A[j]$ dominates $A[k]$ (at least one such index exists, namely i). Now if $A[k+1]$ dominates $A[k]$, since $k+1 \leq j \leq n-1$ the algorithm will detect this and return true. Otherwise, we must have $k+1 < j$ since $A[j]$ does dominate $A[k]$. But since $A[k+1]$ doesn't dominate $A[k]$ and comes after it in the lexicographic order, it must have a smaller y-coordinate than $A[k]$. So $A[j]$ dominates $A[k+1]$, contradicting the fact that k is the *largest* index below j such that $A[j]$ dominates $A[k]$. Therefore if one point in A dominates another, our algorithm will return true. Since the algorithm will clearly return false if no point in A dominates another, it is correct.

4. (25 pts.) Chip design

Charlene the chip designer approaches you with the following problem. Charlene wants to lay out a complete binary tree with n leaves on the chip, where n is a power of two. (Imagine that the leaves contain n inputs, and the goal is to compute the logical AND of these signals, using two-input AND gates.) The nodes and wires are required to follow a rectangular grid. She wants to minimize the total area required for the tree.

In the following parts, show your work and justify your answers.

(a) Charlene's first thought is to lay out the wires like this:



Let $W(n)$ denote the width of this arrangement, in number of grid-squares, and $H(n)$ the height of this arrangement. For instance, in the above picture we have $n = 8$, $H(8) = 3$, and $W(8) = 14$.

Find an explicit formula for $H(n)$.

Solution: The circuit for $2n$ leaves consists of two copies of the circuit for n leaves, placed side-by-side with two spaces in between, and one additional AND gate above them to combine their results. So we have $H(2^0) = 0$ and for $k \geq 1$, $H(2^k) = H(2^{k-1}) + 1$. So $H(2^k) = k$ by a quick induction, and therefore $H(n) = \log_2 n$.

(b) Find a recurrence relation for $W(n)$.

Solution: From the description of the layout used above, we have $W(n) = 2W(n/2) + 2$.

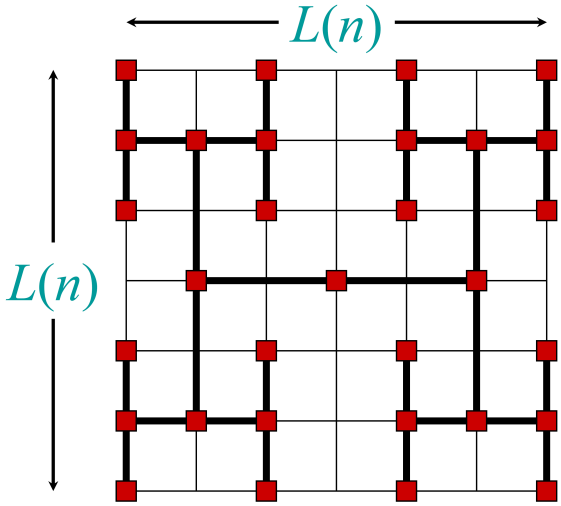
(c) Solve the recurrence relation for $W(n)$, to obtain an asymptotic expression for $W(n)$. In other words, find a function $f(n)$ such that $W(n) \in \Theta(f(n))$.

Solution: By the master theorem with $a = 2$, $b = 2$, $c = \log_2 2 = 1$ and $d = 0$, we have $W(n) = \Theta(n)$.

(d) The total area of her scheme is given by $A(n) = H(n) \times W(n)$. Find an asymptotic expression for $A(n)$. In other words, find a function $f(n)$ such that $A(n) \in \Theta(f(n))$.

Solution: We have $A(n) = \Theta(\log n) \cdot \Theta(n) = \Theta(n \log n)$.

(e) A flash of inspiration strikes you in the shower, and you have another idea for a possible chip layout, like this:



Assume that n is an even power of 2 (e.g., 1, 4, 16, 64, etc.). Then this arrangement will be a square, so let $L(n)$ denote the length of the side of the square, as a function of n. For instance, in the picture above we have $n = 16$ and $L(n) = 6$.

Find a recurrence relation for $L(n)$.

Solution: The circuit for $4n$ leaves consists of four copies of the circuit for n leaves, arranged in a square with two spaces separating them (and three new AND gates to combine their outputs). So we have $L(n) = 2L(n/4) + 2$.

Alternate Solution: The circuit for $4n$ leaves consists of the circuit for n leaves where each leaf has been expanded into an "H" of width and height 2. By a quick induction the circuit for n leaves has the leaves aligned in $\sqrt{n}$ rows of $\sqrt{n}$ leaves each. So we have $L(n) = L(n/4) + 2\sqrt{n/4} = L(n/4) + \sqrt{n}$.

(f) Solve the recurrence relation for $L(n)$, to obtain an asymptotic expression for $L(n)$ (up to a multiplicative factor). In other words, find a function $f(n)$ such that $L(n) \in \Theta(f(n))$.

Solution: By the master theorem with $a = 2$, $b = 4$, $c = \log_4 2 = 1/2$, and $d = 0$, we have $L(n) = \Theta(\sqrt{n})$.

Alternate Solution: By the master theorem with $a = 1$, $b = 4$, $c = \log_4 1 = 0$, and $d = 1/2$, we have $L(n) = \Theta(\sqrt{n})$.

(g) The total area of your scheme is given by $A'(n) = L(n)^2$. Find an asymptotic expression for $A'(n)$. In other words, find a function $f(n)$ such that $A'(n) \in \Theta(f(n))$.

Solution: We have $A'(n) = \Theta(\sqrt{n})^2 = \Theta(n)$.

(h) Which is better, for large n? Charlene's layout depicted in part (a), or your layout from part (e)?

Solution: The layout in part (e) is better, since its area grows only linearly with the number of leaves desired (which is clearly optimal!), while the area of the layout in part (a) grows faster than linearly.

Applications: The recursive H-tree layout is used in chip design, e.g., for communicating a clock signal to many parts of the chip.

5. (20 pts.) Pattern matching, with tolerance for noise

We are given binary strings s, t; s is m bits long, and t is n bits long, and $m < n$. We are also given an integer k. We want to find whether s occurs as a substring of t, but with $\leq k$ errors, and if so, find all such matches. In other words, we want to determine whether there exists an index i such that $s_0, s_1, \ldots, s_{m-1}$ agrees with $t_i, t_{i+1}, t_{i+2}, \ldots, t_{i+m-1}$ in all but k bits; and if yes, find all such indices i.

(a) Describe an $O(mn)$ time algorithm for this string matching problem. Just show the pseudocode; you don't need to give a proof of correctness or show the running time.

Solution: We try matching s against t at all possible shifts, counting how many errors there are at each one:

Algorithm Match($s[0 \ldots m-1], t[0 \ldots n-1], k$):
1. Set $M := \{\}$.
2. For $i := 0, \ldots, n-m$:
3. Set $e := 0$.
4. For $j := 0, \ldots, m-1$:
5. If $s[j] \neq t[i+j]$, set $e := e+1$.
6. If $e \leq k$, add i to M.
7. Return M.

The runtime of this algorithm is clearly $O((n-m) \cdot m) = O(nm)$.

(b) Let's work towards a faster algorithm. Suggest a way to choose polynomials $p(x), q(x)$ of degree $m-1, n-1$, respectively, with the following property: the coefficient of x^{m-1+i} in $p(x)q(x)$ is $m - 2d(i)$, where $d(i)$ is the number of bits that differ between $s_0, s_1, \ldots, s_{m-1}$ and $t_i, t_{i+1}, t_{i+2}, \ldots, t_{i+m-1}$.

Hint: use coefficients $+1$ and -1.

Solution: Define

$$p(x) = \sum_{i=0}^{m-1} (-1)^{s[m-1-i]} x^i$$

and

$$q(x) = \sum_{i=0}^{n-1} (-1)^{t[i]} x^i.$$

Multiplying, we get

$$p(x) \cdot q(x) = \sum_{i=0}^{m-1} \sum_{j=0}^{n-1} (-1)^{s[m-1-i]+t[j]} x^{i+j}.$$

Therefore the coefficient of $x^{m-1+\ell}$ in $p(x) \cdot q(x)$ is

$$\sum_{\substack{i+j=m-1+\ell \\ 0 \le i < m \\ 0 \le j < n}} (-1)^{s[m-1-i]+t[j]} = \sum_{i=0}^{m-1} (-1)^{s[m-1-i]+t[m-1-i+\ell]} = \sum_{i=0}^{m-1} (-1)^{s[i]+t[\ell+i]} = m - 2d(\ell),$$

using the fact that the sum of $m - d(\ell)$ $+1$'s and $d(\ell)$ -1's is $m - 2d(\ell)$. Also, we have used the fact that $(-1)^{y+z}$ is $+1$ if $y = z$ and -1 if $y \neq z$ (when y, z are bits).

Alternatively, you could have written your answer as follows. Define

$$p(x) = (-1)^{s[m-1]} + (-1)^{s[m-2]} x + (-1)^{s[m-3]} x^2 + \cdots + (-1)^{s[0]} x^{m-1}$$
$$q(x) = (-1)^{t[0]} + (-1)^{t[1]} x + (-1)^{t[2]} x^2 + \cdots + (-1)^{t[n-1]} x^{n-1}.$$

Multiplying, we find

$$\begin{aligned} p(x) \cdot q(x) = &(-1)^{s[m-1]+t[0]} + [(-1)^{s[m-2]+t[0]} + (-1)^{s[m-1]+t[1]}]x \\ &+ [(-1)^{s[m-3]+t[0]} + (-1)^{s[m-2]+t[1]} + (-1)^{s[m-1]+t[2]}]x^2 + \cdots \end{aligned}$$

(For instance, we can see that the coefficient of x^2 is the number of bits that differ between $s[m-3], s[m-2], s[m-1]$ and $t[0], t[1], t[2]$.) Now the coefficient of $x^{m-1+\ell}$ in $p(x) \cdot q(x)$ is

$$(-1)^{s[0]+t[\ell]} + (-1)^{s[1]+t[\ell+1]} + \cdots + (-1)^{s[m-1]+t[m+\ell-1]},$$

and this is $m - 2d(\ell)$, as explained above.

(c) Describe an $O(n \lg n)$ time algorithm for this string matching problem, taking advantage of the polynomials $p(x), q(x)$ from part (b).

Solution: Construction of the polynomials $p(x)$ and $q(x)$ above clearly can be done in $O(n)$ time, and each has degree $O(n)$. So we can multiply them in $O(n \log n)$ time with the FFT, and in linear time find all indices ℓ such that the coefficient of $x^{m-1+\ell}$ in the product is $m - 2k$ or larger. By part (b), every such index corresponds to a match between s and $t[\ell \ldots \ell + m - 1]$ with at most k errors. Therefore this algorithm is correct, and its total runtime is $O(n \log n)$.

(d) Now imagine that s, t are not binary strings, but DNA sequences: each position is either A, C, G, or T (rather than 0 or 1). As before, we want to check whether s matches any substring of t with $\le k$ errors (i.e., $s_0, s_1, \ldots, s_{m-1}$ agrees with $t_i, t_{i+1}, t_{i+2}, \ldots, t_{i+m-1}$ in all but k letters), and if so, output the location of all such matches. Describe an $O(n \lg n)$ time algorithm for this problem.

Hint: encode each letter into 4 bits.

Solution: Replace the letters by 1000, 0100, 0010, and 0001, so that s and t are converted to binary strings s' and t'. Then we can compute $p(x)$ and $q(x)$ and before, and return all indices ℓ such that the coefficient of $x^{4m-1+4\ell}$ in the product is $4m - 4k$ or larger. By part (b), every such index corresponds to a match between s' and $t'[4\ell \ldots 4\ell + 4m - 1]$ with at most $2k$ errors. This match in turn corresponds to

a match between s and $t[\ell \ldots \ell + m - 1]$ with at most k errors, since the codes for two different letters differ on exactly two bits (and we're only looking at shifts which are multiples of 4, so that the codes in s' and t' line up). Therefore this algorithm is correct, and since we only increase the sizes of s and t by a constant factor (namely 4), it runs in $O(n \log n)$ time.

6. (10 pts.) Triple sum

Design an efficient algorithm for the following problem. We are given an array $A[0..n-1]$ with n elements, where each element of A is an integer in the range $0 \leq A[i] \leq 10n$. We are also given an integer t. The algorithm must answer the following yes-or-no question: does there exist indices i, j, k such that $A[i] + A[j] + A[k] = t$?

Design an $O(n \lg n)$ time algorithm for this problem.

Hint: define a polynomial of degree $O(n)$ based upon A, then use FFT for fast polynomial multiplication.

Reminder: don't forget to include explanation, pseudocode, running time analysis, and proof of correctness.

Solution:

Main idea: Exponentiation converts addition to multiplication. So, define

$$p(x) = x^{A[0]} + x^{A[1]} + \cdots + x^{A[n-1]}.$$

Notice that $p(x)^3$ contains a sum of terms, where each term has the form $x^{A[i]} \cdot x^{A[j]} \cdot x^{A[k]} = x^{A[i]+A[j]+A[k]}$. Therefore, we just need to check whether $p(x)^3$ contains x^t as a term.

Pseudocode:

Algorithm TripleSum($A[0 \ldots n-1]$, t):
1. Set $p(x) := \sum_{i=0}^{n-1} x^{A[i]}$.
2. Set $q(x) := p(x) \cdot p(x) \cdot p(x)$, computed using the FFT.
3. Return whether the coefficient of x^t in q is nonzero.

Correctness: Observe that

$$q(x) = p(x)^3 = \left(\sum_{0 \leq i < n} x^{A[i]} \right)^3 = \sum_{0 \leq i,j,k < n} x^{A[i]} x^{A[j]} x^{A[k]} = \sum_{0 \leq i,j,k < n} x^{A[i]+A[j]+A[k]}.$$

Therefore, the coefficient of x^t in q is nonzero if and only if there exist indices i, j, k such that $A[i] + A[j] + A[k] = t$. So the algorithm is correct. (In fact, it does more: the coefficient of x^t tells us *how many* such triples (i, j, k) there are.)

Constructing $p(x)$ clearly takes $O(n)$ time. Since $0 \leq A[i] \leq 10n$, $p(x)$ is a polynomial of degree at most $10n = O(n)$. Therefore doing the two multiplications to compute $q(x)$ takes $O(n \log n)$ time with the FFT. Finally, looking up the coefficient of x^t takes constant time, so overall the algorithm takes $O(n \log n)$ time.

Comment: This problem promised you that each element of the array is in the range $0 \ldots 10n$. What if we didn't have any such promise? Then the FFT-based method above becomes inefficient (because the degree of the polynomial is as large as the largest element of A). It is easy to find a $O(n^2)$ time algorithm, but no faster algorithm is known. In particular, it is a famous open problem (called the 3SUM problem) whether this problem can be solved more efficiently than $O(n^2)$ time. This problem has been studied extensively, because it is closely connected to a number of problems in computational geometry.

7. (5 pts.) Optional bonus problem: More medians

(This is an *optional* bonus challenge problem. Only solve it if you want an extra challenge.)

We saw in class a randomized algorithm for computing the median, where the expected running time was $O(n)$. Design a algorithm for computing the median where the *worst-case* running time is $O(n)$.

Hint: It's all in finding a good pivot. If you divide the array into small groups of c elements, can you use that to help you a good pivot?

Solution: As was the case for our earlier algorithm, we actually solve the more general *selection* problem: given an array $A[0\ldots n-1]$ and an index k, find the kth-smallest element.

Main idea: We use the same selection algorithm as from class, but now deterministically select a pivot that will be guaranteed to be pretty good. To do that, we divide the array into $n/5$ groups of 5 elements, take the median of each group, and then use the median of those $n/5$ elements as our pivot. It is possible to show that this pivot is pretty good (somewhere between 30% to 70% of the array is smaller than it).

Pseudocode:

Algorithm Select($A[0\ldots n-1]$, k):
1. If $n = 1$, return $A[0]$.
2. Set $M := \{\}$.
3. For $i := 0, \ldots, \lceil n/5 \rceil - 1$:
4. Compute the median of $A[5i], \ldots, A[5i+4]$ recursively and add it to M.
5. Set $p :=$ Select($M, \lceil n/10 \rceil$).
6. Set $(i, j) :=$ Partition(A, p).
7. If $k < i$, return Select($A[0\ldots i-1]$, k);
8. Else if $i \leq k \leq j$, return p;
9. Else return Select($A[j+1\ldots n-1]$, $k-(j+1)$).

Here Partition(A,x) uses the partitioning algorithm we used previously in the expected linear-time selection algorithm. Given an array A and element x, it rearranges the elements of A into three groups: first elements which are less than x, then one or more copies of x, and finally the elements which are greater than x. It returns the indices of the leftmost and rightmost copy of x. Recall that if A has n elements, we can perform this operation in $O(n)$ time.

Correctness: Correctness follows immediately from our discussion of the randomized algorithm for selection seen in class. In particular, the selection algorithm shown in class is correct no matter how you choose the pivot (the choice of pivot only affects the runtime). The algorithm differs from the selection algorithm shown in class only in its choice of pivot, so it must be correct.

Or, we can prove it correct more directly, as follows. The algorithm above is clearly correct when $n = 1$. For any $n \geq 2$, suppose the algorithm is correct on all arrays with fewer than n elements (and any index k). Then on any array A of length n, we compute an array M with $\lceil n/5 \rceil < n$ elements, and so the recursive call on line 5 correctly finds the $\lceil n/10 \rceil$-th smallest element p of M. Partitioning puts p and its copies into their correct positions in A, so if the kth-smallest value is p we return it correctly on line 8. Otherwise we recurse into the left or right subarray as appropriate, and since these have strictly fewer than n elements the recursive call on line 7 or 9 will return the correct answer. So the algorithm is correct on arrays of size n, and by induction it is correct in general.

Running time: For convenience, assume n is a power of 10; this won't affect the asymptotic running time. The construction of M takes $O(n)$ time, since we compute $O(n)$ medians of a constant number of elements each. By the definition of p, there are at least $n/10$ elements in M smaller or equal to p. Also, for each such

element x in M, there are two more elements in A that are also smaller (the two smallest from the same group as x), since each element in M is a median of a distinct group of 5 elements (from lines 3–4). Therefore, there are at least $3n/10$ elements smaller than or equal to p in A. Therefore $j \geq 3n/10$, and a symmetric argument shows that $i \leq 7n/10$. Thus the recursive calls in lines 7 and 9 are on an array which has at most $7n/10$ elements. Letting $T(n)$ be the worst-case runtime of the algorithm on an array with n elements (over all possible values of k), this gives us the recurrence

$$T(n) \leq T(n/5) + T(7n/10) + O(n).$$

In particular,

$$T(n) \leq T(n/5) + T(7n/10) + cn.$$

for some constant c.

We'll use guess-and-check to solve this recurrence; in particular, we will show that the solution to this recurrence satisfies $T(n) = O(n)$. Take any $d \geq 10c$ such that $d \geq T(1)$. We prove by strong induction that $T(n) \leq dn$ for all n. The base case holds since we required $T(1) \leq d$. For any $n > 1$, suppose that $T(k) \leq dk$ for all $k < n$. Then we have $T(n) \leq T(n/5) + T(7n/10) + cn \leq dn/5 + 7dn/10 + cn \leq 9dn/10 + dn/10 = dn$. So by induction $T(n) \leq dn$ for all n, and thus $T(n) = O(n)$.

Comments: Why groups of 5? Why not divide the array into groups of 3? Because then we'd get the recurrence $T(n) \leq T(n/3) + T(2n/3) + O(n)$, which solves to $O(n \log n)$: no good.

Why not divide the array into groups of 4? Because odd numbers are convenient (what's the median of a group of 4 elements?).

Why not divide the array into groups of 7? That would work. 5 elements just happens to be the smallest odd number where the recurrence solves to $O(n)$.

Incidentally, notice how much simpler the randomized version of this algorithm was? The randomized version also performs better in practice. This example illustrates how randomization can sometimes make algorithms simpler and more efficient—a recurring theme in algorithms.

In fact, it is a famous open problem in complexity theory whether randomized algorithms are more powerful than deterministic algorithms: there are some problems for which we know an efficient randomized algorithm, but we do not know of any efficient deterministic algorithm. (Complexity theorists call this the $P \stackrel{?}{=} BPP$ question.) One example of such a problem is polynomial identity testing: given a prime p and a polynomial $q(x)$, determine whether $q(x) = 0 \pmod{p}$ for all x. We know of an efficient randomized algorithm for polynomial identity testing (just pick a few values of x at random and test whether $q(x) = 0 \pmod{p}$ holds for all of them), but no efficient deterministic algorithm is known. For instance, if we are given a prime p and polynomials $q_1(x), q_2(x), q_3(x)$, we do not know of any efficient deterministic algorithm to check whether $q_1(x) \cdot q_2(x) = q_3(x) \pmod{p}$ holds for all x. Until recently, another example was primality testing: given a number n, determine whether n is prime. For a long time, we knew efficient randomized algorithms for primality testing, but no efficient deterministic algorithm. However, in 2002 there was a major breakthrough: several computer scientists build a deterministic polynomial-time algorithm for primality testing, removing that example from the textbooks[1].

[1] That said, the deterministic algorithm runs in $O((\lg n)^{12})$ time and the randomized algorithm is $O((\lg n)^3)$ time, so the randomized algorithm is significantly more efficient in practice, even though they both run in polynomial time.