College of Computer & Information Science
Northeastern University
Fall 2017

CS5800
MS Algorithms
7 October 2017

Sample Solutions to Problem set 2

1. (3 × 2 = 6 points) Recurrences

Solve the following recurrences and obtain tight asymptotic bounds on $T(n)$. You may ignore the floor operation while solving these recurrences.

(a) $T(n) = 9T(\lfloor n/3 \rfloor) + 7n$, $T(1) = 1$.

Answer: Using Master Theorem: $f(n) = 7n$, $a = 9, b = 3$, so $n^{\log_b a} = n^2$. Clearly, $f(n) = O(n^{2-\varepsilon}$ for $\varepsilon = 1$. So $T(n) = \Theta(n^2)$.

We can also solve it by the recursion tree approach. We find that the contributions in each level of the recursion tree increase geometrically, so the recurrence is determined by the contribution of the last level, which is $\Theta(n^{\log_3 9}) = \Theta(n^2)$.

(b) $T(n) = 9T(\lfloor n/3 \rfloor) + 5n^2$, $T(1) = 1$.

Answer: Using Master Theorem. Since $5n^2 = \Theta\left(n^{\log_3(9)}\right)$ it follows that $T(n) = \Theta(n^2 \log n)$.

We can also solve it by the recursion tree approach. We find that the contributions in each level of the recursion tree are asymptotically the same, which is $\Theta(n^2)$. Since the number of levels is $\log_3 n$, we obtain a value of $\Theta(n^2 \log n)$ for the recurrence.

(c) $T(n) = T(\lfloor n/3 \rfloor) + T(\lfloor 2n/3 \rfloor) + 6n$, $T(1) = 1$.

Answer: We use the recursion tree approach. First level = $6n$. Second level is also $6n$. And so on. The number of levels in the tree is at least $\log_3 n$ and at most $\log_{3/2} n$, so $\Theta(\log n)$. Hence the total contribution is $\Theta(n \log n)$.

2. (2 + 3 = 5 points) More on recurrences

Let $T(n)$ be defined by the following recurrence:

$$T(n) = \begin{cases} 1, & \text{n=1.} \\ 3T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + n^2, & n > 1; \end{cases}$$

(a) Show that $T(n)$ is a monotonically nondecreasing function.

Answer: We will prove this by induction (easy to verify this for $T(1)$ and $T(2)$). We have:

$$T(n) = 3T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + n^2$$

Since $\left\lfloor \frac{n}{2} \right\rfloor < n$ we can use our induction hypothesis. We get:

$$3T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + n^2 \geq 3T\left(\left\lfloor \frac{n-1}{2} \right\rfloor\right) + (n-1)^2 = T(n-1)$$

(b) Find a simple function $f(n)$ such that $T(n) = \Theta(f(n))$. Prove your claim without using the Master theorem. For full credit, prove your claim without ignoring the floor operation. Verify that your result agrees with the Master theorem.

Answer: We give two proofs.

First Proof. For $n = 2^k$ we have:

$$\frac{T(n)}{n^2} = \frac{T(2^k)}{(2^k)^2} = \frac{4^{k+1} - 3^{k+1}}{2^{2k}} = 4 - 3 \cdot \left(\frac{3}{4}\right)^k \underset{k\to\infty}{\Rightarrow} 4$$

We also have:

$$\frac{T\left(\frac{n}{2}\right)}{n^2} = \frac{T(2^{k-1})}{(2^k)^2} = \frac{4^k - 3^k}{2^{2k}} = 1 - \left(\frac{3}{4}\right)^k \underset{k\to\infty}{\longrightarrow} 1$$

For n which is not a power of 2. we can find k such that $2^{k-1} < n < 2^k$. Since $T(n)$ is a monotone function we get for n large enough:

$$T(n) \le T(2^k) \le 5 \cdot (2^k)^2 = 5 \cdot 2 \cdot 2^{k-1} \le 10n^2$$

$$T(n) \ge T(2^{k-1}) \ge \frac{1}{2} \cdot 2^{2k} \ge \frac{1}{2} \cdot n^2$$

It follows that $T(n) = \Theta(n^2)$.

Second Proof. We first note that for all n we have $T(n) \ge n^2$. So this yields $T(n) = \Omega(n^2)$. We next show that there exist constant $c, n_0 > 0$ such that $T(n) \le cn^2$ for all $n \ge n_0$.

The proof is by induction on n. For $n = 1$, we have $T(1) = 1 \le c$ (we will set c at the end).

Suppose the claim is true for $n \le m$, where m is a positive integer. We now consider $T(m+1)$ and obtain the following:

$$\begin{aligned} T(n) &= 3T(\lfloor n/2 \rfloor) + n^2 \\ &\le 3c(\lfloor n/2 \rfloor)^2 + n^2 \\ &\le 3c(n/2)^2 + n^2 \\ &= n^2(3c/4 + 1) \\ &\le cn^2, \end{aligned}$$

if we set $c \ge 3c/4 + 1$, which is true if $c \ge 4$. So we have $T(n) \le 4n^2$ for all $n \ge 1$. Thus, $T(n) = O(n^2)$, completing the proof.

3. (2 + 2 = 4 points) Computing a sorted prefix of a list

(a) Prove or disprove: There exists a comparison algorithm that gets as input an array A of length n, and returns a sorted array with the $\lfloor \frac{n}{4} \rfloor$ smallest elements of A, using $O(n)$ comparison operations.

Answer: The claim is incorrect. The existence of such an algorithm would allow us to sort $\lfloor \frac{n}{4} \rfloor$ elements using $O(n)$ comparisons. This is impossible since we know that sorting $\lfloor \frac{n}{4} \rfloor$ requires $\Omega\left(\lfloor \frac{n}{4} \rfloor \cdot \log_2\left(\lfloor \frac{n}{4} \rfloor\right)\right) = \Omega(n \log_2 n)$

(b) Prove or disprove: There exists a comparison algorithm that gets as input an array A of length n, and returns a sorted array with the $\lfloor\sqrt{n}\rfloor$ smallest elements of A, using $O(n)$ comparison operations.

Answer: The claim is correct.
Use the selection algorithm to find the $\lfloor\sqrt{n}\rfloor$ smallest element of A using $O(n)$ comparisons.
Copy the $\lfloor\sqrt{n}\rfloor$ elements smaller than the $\lfloor\sqrt{n}\rfloor$ smallest element into a new array.
Sort the new array.

(We are assuming here that the time taken to compute the square-root of an integer is $O(1)$.)

4. (5 points) Selection from two sorted lists

Describe a comparison algorithm with time complexity $\Theta(\log n)$, that selects an element of a given rank k in the union of two given sorted arrays.

The input for the algorithm consists of an integer k, $1 \le k \le 2n$ and two sorted arrays of length n:

$$a = (a_1, a_2, ..., a_n) \text{ such that } a_1 < a_2 < ... < a_n$$

$$b = (b_1, b_2, ..., b_n) \text{ such that } b_1 < b_2 < ... < b_n$$

The output of the algorithm is the element of rank k in the union of the sets of elements of the arrays - $a \cup b$. You may assume that no two elements are equal.

Answer: We will, in fact, solve a more general problem. Let the two sorted lists be $A[1..m]$ and $B[1..n]$. Without loss of generality, assume that $n \le m$. The worst-case running time of our algorithm will turn out to be $O(\log n)$.

The basic idea is to reduce the size of array B (i.e., the smaller array) by half in each iteration by performing only a consant number of operations. This is very similar to binary search. For the median problem, one can even reduce the size of both arrays by half. Thus, the number of iterations will be $O(\log n)$ and so will the worst-case running time be. In the following pseudocode, A and B are the two given arrays, p and r are two indices of array B, $p \le r$, and k is the rank that we are seeking. The algorithm returns the element of rank k from the two sorted lists A and $B[p..r]$. In order to find the median, we simply call $TwoLists(A, B, 1, n, n)$.

```
TwoLists($A, B, p, r, k$)

1. **if** $k < r - p + 1$ then   // No point in searching $B[p + k..r]$
2.     then $r = p + k - 1$
3. **if** $p = r$ then   // $B$ has only one element
4.     **if** $A[k-1] > B[p]$ then **return** $A[k-1]$
5.     **else if** $A[k-1] < B[p]$ then **return** $\min\{A[k], B[p]\}$
6. **else**   // $B$ has more than one element
7.     $\ell = \lfloor (r - p + 1)/2 \rfloor$   // $\ell$ is half the size of $B$
8.     **if** $A[k - \ell] > B[p + \ell - 1]$   // compare the middle element of $B$ with appropriate element of $A$
9.     then **return** $TwoLists(A, B, p + \ell, r, k - \ell)$   // discard first half of $B$
10.    **else return** $TwoLists(A, B, p, p + \ell - 1, k)$   // discard second half of $B$
```

The worst-case running time of the algorithm is given by the recurrence $T(n) = T(n/2) + \Theta(1)$, which yields $T(n) = \Theta(\log n)$ (like binary search).

5. (5 points) Hidden surface removal in computer graphics

Chapter 5, Exercise 5, page 248.

Answer: We first label the lines in order of increasing slope, and then use divide and conquer. If $n \leq 3$, then we can easily solve it in constant time. The first and third lines will always be visible, the second will be visible if and only if it meets the first line to the left of where the third line meets the first line. We can compute the intersection points and compare to obtain the visible lines.

Let $m = \lceil n/2 \rceil$. We first recursively compute the sequence of visible lines among $L_1, \ldots, L_m$. Let the output of the recursion be $\mathcal{L} = \{L_{i_1}, \ldots, L_{i_p}\}$ in order of increasing slope. We also compute, in this recursive call, the sequence of points $a_1, \ldots, a_{p-1}$ where a_k is the intersection of line L_{i_k} with $L_{i_{k-1}}$. Notice that $a_1, \ldots, a_{p-1}$ will have increasing x-coordinates; for if two lines are both visible, the region in which the line of smaller slope is uppermost lies to the left of the region in which the line of larger slope is uppermost. Similarly, we recursively compute the sequence $\mathcal{L}' = \{L_{j_1}, \ldots, L_{j_q}\}$ of visible lines among $L_{m+1}, \ldots, L_n$, together with the sequence of intersection points $b_k = L_{j_k} \cap L_{j_{k-1}}$, for $k = 1, \ldots, q-1$.

To complete the algorithm, we need to design the conquer step. That is, we must show how to determine the visible lines in $\mathcal{L} \cup \mathcal{L}'$, together with the corresponding intersection points, in $O(n)$ time. (Note that $p + q \leq n$, so if we run in linear time, we are done.) We know that L_{i_1} and L_{j_q} are both visible since they have the minimum and maximum slopes, respectively, among all lines.

We merge the sorted lists $a_1, \ldots, a_{p-1}$ and $b_1, \ldots, b_{q-1}$ into a single list of points $c_1, c_2, \ldots, c_{p+q-2}$ ordered by increasing x-coordinates. This takes $O(n)$ time. For each k, we consider the line that is uppermost in $\mathcal{L}$ at x-coordinate of c_k, and the line that is uppermost in $\mathcal{L}'$ at x-coordinate of c_k. Let ℓ be the smallest index for which the uppermost line in $\mathcal{L}'$ lies above the uppermost line in $\mathcal{L}$ at x-coordinate c_ℓ. Let the two lines at this point be $L_{i_s} \in \mathcal{L}$ and $L_{j_t} \in \mathcal{L}'$. Let (x^*, y^*) denote the point at which L_{i_s} and L_{j_t} intersect. We thus see that x^* lies between the x-coordinates of $c_{\ell-1}$ and c_ℓ. This means that L_{i_s} is uppermost and immediately to the left of x^*, and L_{j_t} is uppermost immediately to the right of x^*. Consequently, the sequence of visible lines among $\mathcal{L} \cup \mathcal{L}'$ is $L_{i_1}, \ldots, L_{i_s}, L_{j_t}, \ldots, L_{j_q}$ and the sequence of intersection points is $a_{i_1}, \ldots, a_{i_s-1}, (x^*, y^*), b_{j_{t+1}}, \ldots, b_{j_{q-1}}$. This is exactly what we need to return to the next level of the recursion.

The running time of the conquer step is $O(n)$ since we are only doing a linear scan of the lists with comparisons.

6. (1 + 1 + 3 = 5 points) A fault-tolerant AND-gate

Assume we are given an infinite supply of two-input, one-output gates, most of which are AND gates and some of which are OR gates. Unfortunately the OR and AND gates have been mixed together and we can't tell them apart. For a given integer $k \geq 0$, we would like to construct a two-input, one-output combinational "k-AND" circuit from our supply of two-input, one output gates such that the following property holds: If at most k of the gates are OR gates then the circuit correctly implements AND. Assume for simplicity that k is a power of two.

For a given integer $k \geq 0$, we would like to design a k-AND circuit that uses the smallest number of gates.

(a) Design a 1-AND circuit with the smallest number of gates. Argue the correctness of your circuit.

Answer: For $k = 1$, we have the circuit given in Figure 1, where each triangle represents a gate. The circuit works correctly if at most one of the three gates is OR. If any of the gates on the first level is OR, then the other two gates are AND, allowing the AND computed by the first OR gate to be propagated correcty to the final AND gate. A more formal proof is given below in part (c).

(b) Using a 1-AND circuit as a black box, design a 2-AND circuit. Argue the correctness of your circuit.

Answer: For $k = 2$, one can build a circuit with three gates in the first level, two in the second level, and one in the third level. The middle gate in the first level feeds its output to both the gates in the second level. If only two of the six gates are OR, then no matter where the two OR gates are placed, there is a path from the two inputs to the output consisting entirely of AND gates, ensuring that the correct output is produced at the end.

The above circuit will work correctly, and can be generalized to yield a $\Theta(k^2)$ gate implementation for k-AND.

The above circuit, however, is not a "black box" construction using 1-AND circuits. For a black box construction, we can simply take the same solution as given in Figure 1, where each triangle now represents a 1-AND circuit. This circuit uses 9 gates, but will lead to better implementation than $\Theta(k^2)$ for k-AND, as we see below. The correctness of this 2-AND circuit follows from the proof given below for part (c), which is for all k.

(c) Generalizing the above approach, or using a different approach, design the best possible k-AND circuit you can and derive a Θ-bound (in terms of the parameter k) for the number of gates in your k-AND circuit. Argue the correctness of your circuit.

Answer: We construct a $\Theta(k^{\lg 3})$-gate k-AND circuit. The construction is recursive (divide-and-conquer). For $k = 1 = 2^0$, we have the circuit given in Figure 1, where each triangle represents a gate. The circuit for $k = 2^{i+1}$ is constructed out of 3 2^i-AND circuits, as depicted in Figure 1, where each triangle represents a 2^i-AND circuit.

In order the prove the correctness of the circuit, we first note that the output of the OR and AND gates match when both the inputs are identical (that is, when they are either 0 0 or 1 1). It follows that any 2-input circuit consisting of OR and AND gates only will yield as an output 0 (resp., 1) if both of its inputs are 0s (resp., 1s), regardless of the number of OR gates. Therefore, any circuit consisting of OR and AND gates only will yield the correct AND output when the two inputs are identical. So we need only consider the case when one of the inputs is a 0 and the other is a 1. In this case, the desired output is a 0.

The proof of correctness is by induction on i. For the induction base $k = 2^0 = 1$, we note that at most one of A, B, or C is an OR gate. If C is an OR gate, then the outputs of A and B yield 0 each, implying that the output of C is also a 0. If C is an AND gate, then one of A and B is an AND gate; therefore, at least one of the inputs to C is a 0, implying that the output of C is also 0, thus establishing the correctness of the circuit.

As the induction hypothesis, we assume that k-AND circuits for $k = 2^i$ are correct, for some $k \geq 0$. We now prove the correctness of the $2k$-AND circuit. Since there are at most $2k$

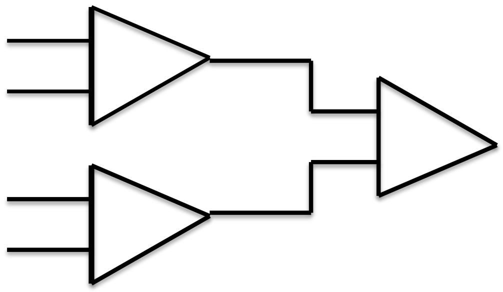

Figure 1: Recursive construction of a k-AND circuit

OR gates, only one of the 3 circuits X, Y, and Z has more than k OR gates. This implies that two of the 3 circuits correctly compute an AND. If X and Y are AND circuits, then the outputs of the 2 gates are 0s. Since the circuit Z consists of OR and AND gates only, the output of Z is 0 (see the observation above). On the other hand, if Z and one of X and Y are AND circuits, then at least one of the inputs into Z is a 0 and the output of Z is also a 0, thus establishing the correctness of the circuit.

We now compute the number of gates. This is given by the recurrence $N(k)$ as follows.

$$N(k) = \begin{cases} 3 & k = 1 \\ 3N(k/2) & k = 2^i, i > 0 \end{cases}$$

On solving the recurrence by simple iteration, or using the Master Theorem, we get $N(k) = 3k^{\lg 3}$.