CS 341, Winter 2015
Andrew Arnold, Timothy Chan, Doug Stinson

Assignment 2 (due Tuesday, February 10, 4pm)

Please read `https://www.student.cs.uwaterloo.ca/~cs341/policies.html` for general instructions.

1. [*16 marks*]

   (a) Consider the following recurrence:

$$T(n) = \begin{cases} 9\,T(\lfloor n^{1/3} \rfloor) + \log^2 n & \text{if } n \geq 64 \\ 1 & \text{if } n < 64 \end{cases}$$

   Use the guess-and-check (substitution) method to prove the upper bound $T(n) = O(\log^2 n \log\log n)$.

   Answer: We guess that $T(n) \leq c \log^2 n \log\log n$ for $n \geq n_0 = 4$.

   Base case: If $4 \leq n < 64$, then $T(n) = 1$, and

$$\log^2 n \log\log n \geq \log^2 4 \log\log 4 = 2^2 \log 2 = 4.$$

   Thus $T(n) \leq cn$ provided $c \geq \frac{1}{4}$.

   Inductive step: Suppose $n \geq 64$ and $T(m) \leq c \log^2 m \log\log m$ for $4 \leq m < n$. Note that $\lfloor n^{1/3} \rfloor \geq \lfloor 64^{1/3} \rfloor = 4$, such that the induction hypothesis holds for $T(\lfloor n^{1/3} \rfloor)$. Thus

$$\begin{aligned} T(n) &= 9c \log^2 \lfloor n^{1/3} \rfloor \log\log \lfloor n^{1/3} \rfloor + \log^2 n, \\ &\leq 9c \log^2 (n^{1/3}) \log\log (n^{1/3}) + \log^2 n, \\ &= 9c (\tfrac{1}{3} \log n)^2 \log (\tfrac{1}{3} \log n) + \log^2 n, \\ &= c \log^2 n (\log\log n + \log \tfrac{1}{3}) + \log^2 n, \\ &= c \log^2 n (\log\log n - \log 3 + \tfrac{1}{c}). \end{aligned}$$

   This is less than $c \log^2 n \log\log n$ provided $\frac{1}{c} \leq \log 3$, i.e., if $c \geq 1/(\log 3)$. As $1/(\log 3) < 1/(\log 2) = 1$, we can take $c = 1$.

   We have shown $T(n) < \log^2 n \log\log n$ for all $n \geq 4$. Thus $T(n) \in \mathcal{O}(\log^2 n \log\log n)$.

   (b) Consider the following recurrence:

$$T(n) = \begin{cases} T(\lfloor n/2 \rfloor) + T(\lfloor n/4 \rfloor) + 1 & \text{if } n \geq 4 \\ 0 & \text{if } n < 4 \end{cases}$$

   Use the guess-and-check (substitution) method to prove the upper bound $T(n) = O(n^{0.7})$. [*Hint*: use $T(n) \leq cn^{0.7} - c'$ as the "guess", where c and c' are suitable constants.]

   Answer: We guess $T(n) \leq cn^{0.7} - c'$ for $n \geq 1$.

Base case: If $1 \le n \le 4$, then $T(n) = 0 \le cn^{0.7} - c'$, provided $c \ge c'$.

Inductive step: Suppose $n > 4$ and $T(m) < cm^{0.7} - c'$ for $1 \le m < n$. Note $1 \le \lfloor n/2 \rfloor, \lfloor n/4 \rfloor < n$, thus the induction hypothesis applies to $T(\lfloor n/2 \rfloor)$ and $T(\lfloor n/4 \rfloor)$, and

$$\begin{aligned} T(n) &\le \big(c\lfloor n/2 \rfloor^{0.7} - c'\big) + \big(c\lfloor n/4 \rfloor^{0.7} - c'\big) + 1, \\ &\le c(n/2)^{0.7} + c(n/4)^{0.7} - (2c' - 1), \\ &= c((\tfrac{1}{2})^{0.7} + (\tfrac{1}{4})^{0.7})n^{0.7} - (2c' - 1), \\ &\le cn^{0.7} - (2c' - 1) \quad \text{as } (\tfrac{1}{2})^{0.7} + (\tfrac{1}{4})^{0.7} < 0.99451, \\ &\le cn^{0.7} - c', \quad \text{provided } 2c' - 1 \ge c'. \end{aligned}$$

Thus it suffices that $c' \ge 1$ and $c \ge c'$. E.g., we can take $c = c' = 1$. We have shown $T(n) \le n^{0.7} - 1$ for $n \ge 1$, and so $T(n) \in \mathcal{O}(n^{0.7})$.

2. [*6 marks*] Consider the following pseudocode:

```
strange(A, ℓ, r):
1.  if r − ℓ ≤ 5 then return
2.  m = (r − ℓ)/2
3.  strange(A, ℓ, ℓ + m)
4.  strange(A, ℓ + m, r)
5.  for i = ℓ to r do
6.      for j = r to ℓ do
7.          print A[j]
8.  strange(A, ℓ + (m/2), r − (m/2))
```

Analyze the time complexity as a function of $n = r - \ell$ by giving a recurrence $T(n)$ for this pseudocode and solving the recurrence with the method of your choice. Give a tight (Θ) bound. (You may assume $r - \ell$ is a power of 2.)

Answer:

The cost of strange is dominated by the three recursive calls with the parameter n reduced by a factor $\frac{1}{2}$, and the nested for loops on lines 5 and 6, which in total take $\Theta(n^2)$ iterations. Thus the cost may be expressed as the recurrence

$$T(n) = \begin{cases} 3T(n/2) + \Theta(n^2) & \text{if } n > 5 \\ \Theta(1) & \text{if } n \le 5 \end{cases}$$

As $\log_2 3 \approx 1.585$, we have that $\frac{n^2}{n^{\log_2 3 + 0.01}}$ is strictly increasing for $n > 0$, and thus $T(n) \in \Theta(n^2)$ by case 3 of the Master Theorem.

Alternative solution: If you interpreted the for loop on line 6 as never iterating (as $r \ge \ell$), then lines 5 and 6 take $\Theta(n)$ iterations as opposed to $\Theta(n^2)$. We then get the recurrence

$$T(n) = \begin{cases} 3T(n/2) + \Theta(n) & \text{if } n > 5 \\ \Theta(1) & \text{if } n \le 5 \end{cases}$$

As $n \in \mathcal{O}(n^{\log_2 3 - 0.01})$, this is case 1 of the Master Theorem, which gives $T(n) \in \Theta(n^{\log_2 3}) \subset \mathcal{O}(n^{1.59})$.

3. [*14 marks*] Given a set P of n points in two dimensions where each point is labelled either "red" or "blue", we want to count the number of pairs (r, b) where r is a red point in P and b is a blue point in P, such that r dominates b. Here, we say that r dominates b if r has larger x-coordinate and larger y-coordinate than b. Design and analyze a divide-and-conquer algorithm that solves this problem in $\mathcal{O}(n \log n)$ time. [Hint: divide by the median x-coordinate of the n points, like in the closest pair algorithm from class (although the details will be different from the closest pair algorithm).]

Answer:

Idea. Divide the point set into the left and right halves by the vertical line at the median x-coordinate. Recursively compute the count c_L for the left subset and c_R for the right subset. We still need to count the number c_{RL} of pairs (r, b) where red point r dominates blue point b with r in the right subset and b in the left subset. To compute c_{RL}, we scan the points in increasing y-order and keep track of the number of left blue points $n_{\text{blue,left}}$ seen so far; whenever we encounter a left blue point, we increment $n_{\text{blue,left}}$; whenever we encounter a right red point, it dominates $n_{\text{blue,left}}$ left blue points and so we add $n_{\text{blue,left}}$ to the counter c_{RL}. The final answer is $c_L + c_R + c_{RL}$.

We will pre-sort the input points by x-coordinates. Then dividing by the median x-coordinate becomes trivial. For the scan in increasing y-order to compute c_{RL}, one possible approach is to pre-sort by y-coordinates as well (as in the textbook's description of Shamos' closest pair algorithm). In the pseudocode below, we suggest another approach: require the algorithm to not only return the count but also rearrange the points in y-sorted order. From the y-sorted orders of the left subset and of the right subset produced after the recursive calls, we can obtain the y-sorted order of the whole set by a linear scan (similar to the merging subroutine from mergesort).

Pseudocode. (For simplicity, we assume that the x-coordinates are all distinct and the y-coordinates are all distinct.)

```
DominanceCount(p_0, ..., p_{n-1}):
// Input:   points p_0, ..., p_{n-1} sorted in increasing x-order
// Output:  (c, ⟨t_0, ..., t_n⟩) where
            c is the number of red-blue dominating pairs, and
            t_0, ..., t_{n-1} is the points p_0, ..., p_{n-1} rearranged in increasing y-order
            with t_n = (∞, ∞)
 0. if n = 0 then return (0, ⟨(∞, ∞)⟩)
 1. if n = 1 then return (0, ⟨p_0, (∞, ∞)⟩)
 2. (c_L, ⟨q_0, ..., q_{⌊n/2⌋}⟩) = DominanceCount(p_0, ..., p_{⌊n/2⌋-1})
 3. (c_R, ⟨s_0, ..., s_{⌈n/2⌉}⟩) = DominanceCount(p_{⌊n/2⌋}, ..., p_{n-1})
 4. i = j = n_blue,left = c_RL = 0
 5. for k = 0 to n − 1 do {
 6.     if q_i.y ≤ s_j.y then {
 7.         t_k = q_i; increment i
 8.         if t_k is blue then increment n_blue,left
        }
 9.     else {
10.         t_k = s_j; increment j
11.         if t_k is red then c_RL = c_RL + n_blue,left
        }
    }
12. return (c_L + c_R + c_RL, ⟨t_0, ..., t_{n-1}, (∞, ∞)⟩)
```

Analysis. Let $T(n)$ be the running time of DominanceCount on an input with n points. Line 2 takes $T(n/2)$ time and line 3 takes $T(n/2)$ time (ignoring floors and ceilings). Lines 5–11 take $\Theta(n)$ time since there are n iterations, each taking constant time. Thus, we obtain the following recurrence

$$T(n) = \begin{cases} 2\,T(n/2) + \Theta(n) & \text{if } n > 1 \\ \Theta(1) & \text{if } n = 1 \end{cases}$$

This is identical to the recurrence from mergesort and thus solves to $T(n) = \Theta(n \log n)$.

Now, this assumes that the input has been pre-sorted by x-coordinate. Pre-sorting takes $\Theta(n \log n)$ time, e.g., by mergesort, and the total time remains $\Theta(n \log n)$.

4. [*14 marks*] Imagine a card game involving two parties: yourself and a dealer. In this game there are n cards $C_1, \ldots, C_n$, face down on a table, each with a symbol on its face-down side, call it symbol(C_i). You cannot look at the cards, but you can ask the dealer if two cards have the same symbol (and the dealer's reponse is "yes" if they are the same, and "no" if they are different). Your task is to tell whether or not more than $n/2$ cards have the same symbol. Furthermore, if the answer is yes, you should also output an index i of any card C_i whose symbol appears more than $n/2$ times (of course, i is not unique). Design a divide-and-conquer strategy to accomplish this task, that requires you ask the dealer at most $\mathcal{O}(n \log n)$ times (or better).

Answer:

Idea. Divide the n cards into two groups of $n_1 = \lfloor n/2 \rfloor$ and $n_2 = \lceil n/2 \rceil$ cards. Recursively find a card C_i whose symbol appears more than $n_1/2$ times in the first group (if it exists), and a card C_j whose symbol appears more than $n_2/2$ times in the second group (if it exists).

- **Observation**: If there is a card C_k whose symbol appears more than $n/2$ times, then symbol(C_k) must be equal to symbol(C_i) or symbol(C_j).
- **Proof**: Suppose that the conclusion is false. Since symbol(C_k) $\neq$ symbol(C_i), symbol(C_k) appears at most $n_1/2$ times in the first group (because only symbol(C_i) can appear more than $n_1/2$ times in the first group). Similarly, since symbol(C_k) $\neq$ symbol(C_i), symbol(C_k) appears and at most $n_2/2$ times in the second group. Thus, symbol(C_k) appears at most $n_1/2 + n_2/2 = n/2$ times in total: a contradiction.

Thus, we only have two possible candidates for a symbol that appear more than $n/2$ times—it's either symbol(C_i) or symbol(C_j). We can check these two possibilities by asking the dealer $O(n)$ questions.

Pseudocode.

```
CardGame(C_1, ..., C_n):
 1. if n = 1 then return ("yes", 1)
 2. (x, i) = CardGame(C_1, ..., C_⌊n/2⌋)
 3. (y, j') = CardGame(C_⌊n/2⌋+1, ..., C_n); j = ⌊n/2⌋ + j'
 4. count_1 = count_2 = 0
 5. for k = 1 to n do {
 6.     ask the dealer if symbol(C_k) = symbol(C_i)
 7.     if the dealer replies "yes" then increment count_1
 8.     ask the dealer if symbol(C_k) = symbol(C_j)
 9.     if the dealer replies "yes" then increment count_2
    }
10. if count_1 > n/2 then return ("yes", i)
11. if count_2 > n/2 then return ("yes", j)
12. return ("no", 1)
```

In the pseudocode above, if the answer is "yes", the returned answer is ("yes", k) for some card C_k whose symbol appears more than $n/2$ times. If the answer is "no", the returned answer is ("no", k) for some arbitrary index k.

Analysis. Let $T(n)$ be the number of questions posed to the dealer when running CardGame on n cards. Line 2 requires $T(n/2)$ questions and line 3 takes $T(n/2)$ questions (ignoring floors and ceilings). Lines 5–10 requires $2n$ questions. Thus, we obtain the following recurrence

$$T(n) = \begin{cases} 2\,T(n/2) + 2n & \text{if } n > 1 \\ 0 & \text{if } n = 1 \end{cases}$$

This is similar to the recurrence from mergesort and thus solves to $T(n) = \Theta(n \log n)$.

Remark. There are more efficient algorithms that require only $\Theta(n)$ questions. One approach is based on pairing cards and leads to an improved recurrence $T(n) = T(n/2)+\Theta(n)$. Another approach finds a candidate solution directly with a clever linear scan (the candidate can then be checked with a second linear scan).

5. [*20 marks*] Let A be a 7×7 matrix where each entry is either 0 or 1.

   (a) [*4 marks*] Argue that each entry of the matrix A^k are less than 7^k and requires $O(k)$ bits each.

   Answer: We prove that each entry of A^k is a positive integer between 0 and 7^{k-1}. For $k = 1$ this is obviously correct. As an induction assumption, assume the statement is true for $k-1 \geq 0$; we prove that it is true for k. Note that $A^k = A \times A^{k-1}$. Let A have entries a_{ij}, let A^{k-1} have entries b_{ij} and let A^k have entries c_{ij}. From the definition of matrix multiplication we have

   $$c_{ij} = \sum_{h=1}^{7} a_{ih} b_{hj}.$$

   Clearly each $c_{ij} \geq 0$. The maximum value of c_{ij} occurs when every $a_{ih} = 1$ and every $b_{hj} = 7^{k-2}$. In this case, $c_{ij} = 7 \times 7^{k-2} = 7^{k-1}$.
   Since $0 \leq a_{ij} \leq 7^{k-1}$, the number of bits required to represent a_{ij} (for $k \geq 2$) is at most

   $$\log_2(7^{k-1} + 1) \leq \log_2 8^{k-1} = 3(k-1) < 3k \in O(k).$$

   (b) [*10 marks*] Design and analyze an algorithm that computes the matrix A^n for a given n with running time $\mathcal{O}(n^{\log_2 3})$ (measured in bit complexity).
   Note that because matrix entries are large integers, do not assume that multiplying two integers takes constant time. Do not assume that n is a power of 2. You may use Karatsuba's integer multiplication algorithm as a subroutine.

   Answer: Using Karatsuba, we assume that multiplication of two n-bit integers can be done in time $O(n^{\log_2 3})$. We use the following recursive formula for A^n:

   $$A^n = \begin{cases} A^{n/2} \times A^{n/2} & \text{if } n \geq 2 \text{ is even} \\ A \times (A^{(n-1)/2} \times A^{(n-1)/2}) & \text{if } n \geq 3 \text{ is odd.} \end{cases}$$

   Suppose first that n is even. When we compute A^n from $A^{n/2}$, each of the 49 entries is computed using 7 multiplications and 6 additions of $O(n)$-bit integers. Observe that $(cn)^{\log_2 3}) = c^{\log_2 3} n^{\log_2 3} \in O(n^{\log_2 3})$, so the time required to compute A^n from $A^{n/2}$ is therefore $O(n^{\log_2 3})$.
   If n is odd, the analysis is similar. We do have an extra multiplication by A after we compute A^{n-1} from $A^{(n-1)/2}$, but this just requires a constant number of additions of $O(n)$-bit integers and so it does not increase the complexity.
   The time $T(n)$ to compute A^n can therefore be expressed as a recurrence

   $$T(n) = T(\lfloor n/2 \rfloor) + O(n^{\log_2 3}).$$

   Applying the Master Theorem, we see that $T(n) \in O(n^{\log_2 3})$.

(c) [*6 marks*] Now suppose that A is a $b \times b$ matrix instead. Describe how to modify your analysis to show that A^n can be computed with running time $\mathcal{O}(n^{\log_2 3} b^{\log_2 7} \log^{\log_2 3} b)$ (measured in bit complexity). You may use Karatsuba's integer multiplication algorithm and Strassen's matrix multiplication algorithm as subroutines.

Answer: Similar to part (a), the maximum value of an entry in A^n is b^{n-1}, so each entry of A^n requires at most $n \log_2 b$ bits to store. We use the same recursive algorithm as in part (b). Suppose first that n is even. When we compute A^n from A using Strassen's algorithm, the number of multiplications that are performed is $O(b^{\log_2 7})$. Each of these multiplications can be done in time $O((n \log_2 b)^{\log_2 3})$. So the time required for this computation is $O(n^{\log_2 3}(\log_2 b)^{\log_2 3} b^{\log_2 7})$. A similar analysis applies for n odd. Thus the recurrence is

$$T(n) = T(\lfloor n/2 \rfloor) + O(n^{\log_2 3}(\log_2 b)^{\log_2 3} b^{\log_2 7}).$$

Applying the Master Theorem, we see that $T(n) \in O(n^{\log_2 3}(\log_2 b)^{\log_2 3} b^{\log_2 7})$.