Homework 5

Due: Feb 21 2025

Graded Questions

1. Solve the following recurrences by giving tight Θ-notation bounds in terms of n for sufficiently large n. Assume that $T(\cdot)$ represents the running time of an algorithm, i.e., $T(n)$ is positive and non-decreasing, and for small constants c independent of n, $T(c)$ is also constant. **You need to give the answer and the analysis.** (10 points)

   (a)
   $$T(n) = 9T\left(\frac{n}{5}\right) + n\log n$$

   We observe that $n^{\log_b a} = n^{\log_5 9}$ and $f(n) = n\log n = O(n^{\log_5 9-\epsilon})$ for any $0 < \epsilon < \log_5 9 - 1$. Thus, invoking the Master Theorem:
   $$T(n) = \Theta(n^{\log_5 9}) \approx \Theta(n^{1.365}).$$

   (b)
   $$T(n) = \sqrt{2025}T\left(\frac{n}{3}\right) + n^{\sqrt{2025}}$$

   We have $n^{\log_b a} = n^{\log_3 \sqrt{2025}} \approx n^{3.465} = O(n^{3.5})$ and $f(n) = n^{\sqrt{2025}} = \Omega(n^{3.5+\epsilon})$ for any $0 < \epsilon < \sqrt{2025} - 3.5$. Thus, from the Master Theorem:
   $$T(n) = \Theta(n^{\sqrt{2025}}).$$

   (c)
   $$T(n) = 9T\left(\frac{n}{3}\right) + n^2\log n$$

   We observe that $f(n) = n^2\log n$ and $n^{\log_b a} = n^{\log_3 9} = n^2$. Applying the generalized Master Theorem, we get:
   $$T(n) = \Theta(n^2\log^2 n).$$

   (d)
   $$T(n) = 10T\left(\frac{n}{2}\right) + 2^n$$

   We have $n^{\log_b a} = n^{\log_2 10}$ and $f(n) = 2^n = \Omega(n^{\log_2 10+\epsilon})$ for any $\epsilon > 0$. From the Master Theorem:
   $$T(n) = \Theta(2^n).$$

(e)

$$T(n) = 3T\left(\frac{n}{4}\right) + n\log^2 n$$

We observe that $n^{\log_b a} = n^{\log_4 3} \approx n^{0.793}$ and $f(n) = n\log^2 n = \Omega(n^{0.793+\epsilon})$ for some $0 < \epsilon < 1 - 0.793$. Thus, invoking the Master Theorem:

$$T(n) = \Theta(n\log^2 n).$$

Rubrics (10 points):

- 2 points for each question: 1 point for correct answer; 1 point for analysis.

2. Given a square matrix M of size $n \times n$, where each row and each column is sorted in increasing order, design a divide-and-conquer algorithm to find a given value k in M. Assume that n is a power of 2. You need to:

   (a) Describe your algorithm. It **must** be a divide-and-conquer algorithm. No proof of correctness is needed. (5 points)

   (b) Give your algorithm's recurrence relation for runtime complexity. Briefly explain it. (5 points)

   (c) Solve the recurrence relation using the Master Theorem. (5 points)

   (a) The algorithm follows a divide-and-conquer approach:

   - Partition M into four submatrices:

$$\begin{aligned}
&M_{11}: \text{Top-left submatrix} \quad (0 \le r < n/2, 0 \le c < n/2)\\
&M_{12}: \text{Top-right submatrix} \quad (0 \le r < n/2, n/2 \le c < n)\\
&M_{21}: \text{Bottom-left submatrix} \quad (n/2 \le r < n, 0 \le c < n/2)\\
&M_{22}: \text{Bottom-right submatrix} \quad (n/2 \le r < n, n/2 \le c < n)
\end{aligned}$$

   - Compare k with the middle element $M[n/2][n/2]$:
     - If $k = M[n/2][n/2]$, return its index.
     - If $k < M[n/2][n/2]$, search in M_{11}, M_{12}, M_{21}.
     - If $k > M[n/2][n/2]$, search in M_{12}, M_{21}, M_{22}.
   - The search continues recursively in the appropriate submatrices.
   - The algorithm is parameterized by (lowRow, highRow, lowCol, highCol) to avoid copying submatrices.

   (b) The recurrence relation for the runtime complexity is:

$$T(n) = \begin{cases} C_1, & \text{if } n = 1 \\ 3T(n/2) + C_2, & \text{if } n > 1 \end{cases}$$

   where C_1 and C_2 are constants.

   (c) Solving the recurrence relation using the Master Theorem:

   - We have $a = 3$, $b = 2$, and $f(n) = O(1)$.
   - The number of leaves in the recurrence tree is $n^{\log_2 3}$.
   - Since $f(n) = O(n^0)$ and $\log_2 3 > 0$, the recurrence falls under Case 1 of the Master Theorem.

- Thus, the time complexity is:

$$T(n) = \Theta(n^{\log_2 3})$$

where $\log_2 3 \approx 1.585$.

Rubrics (15 points):

- (a) 5 points for a correct and detailed description of the divide-and-conquer algorithm, including how the matrix is partitioned and how the search is conducted.
- (b) 5 points for correctly formulating the recurrence relation and providing a clear explanation of its derivation.
- (c) 5 points for correctly applying the Master Theorem and solving the recurrence to obtain the final time complexity.

(a) Alternative Algorithm (Linear Time):
Start at the top-right corner of the matrix (or bottom-left).

- Initialize `row = 0` and `col = n - 1`.
- While `row < n` and `col >= 0`:
  - If `M[row][col] == k`, return `(row, col)`.
  - If `M[row][col] < k`, increment `row`. (Eliminate current row)
  - If `M[row][col] > k`, decrement `col`. (Eliminate current column)
- If not found, return "Not Found".

(b) Recurrence Relation:
In the worst case, each comparison eliminates a row or a column. Worst-case runtime (representing reducing rows + columns by 1 each step):

$$T(n) = T(n-1) + O(1)$$

(c) Time Complexity (Direct Analysis):

$$T(n) = T(n-1) + c = (T(n-2) + c) + c = T(n-2) + 2c = ... = T(0) + nc$$

for some c.
Therefore, the time complexity is

$$T(n) = O(n).$$

Rubrics (15 points):

- (a) 5 points for a correct description of the linear-time algorithm, including the starting point, looping condition, comparisons, and actions.
- (b) 5 points for correctly formulating recurrence relations $T(n)$.
- (c) 5 points for stating the $O(n)$ time complexity and providing a direct analysis (unrolling) of the $T(n) = T(n-1) + O(1)$ recurrence.

3. Solve Kleinberg and Tardos, Chapter 5, Exercise 3. You need to:

(a) Describe your algorithm. It **must** be a divide-and-conquer algorithm. No proof of correctness is needed. (7 points)

(b) Give your algorithm's recurrence relation for runtime complexity. Briefly explain it. (3 points)

(c) Solve the recurrence relation using the Master Theorem. (5 points)

We solve the problem by employing a divide-and-conquer strategy that resembles the majority vote algorithm. The idea is to recursively reduce the problem size by finding a potential majority candidate in each half of the input, and then merging these candidates to decide on a possible overall majority.

Algorithm Description:

(a) **Base Case:** If the input set contains only one card, return that card as the candidate.

(b) **Divide:** Partition the n cards into two groups S_1 and S_2 of (roughly) equal size.

(c) **Conquer:** Recursively, determine a candidate for the majority equivalence class in each half. Denote these candidates by c_1 (from S_1) and c_2 (from S_2). Note that it is possible for a recursive call to return "no candidate" if the half does not have a majority.

(d) **Combine:**

- If both c_1 and c_2 are present, use the equivalence tester to check whether they belong to the same equivalence class.
  - If they are equivalent, choose either as the candidate for the entire set.
  - If they are not equivalent, it implies that any majority candidate must appear in both halves. In this case, we postpone the decision and later verify by counting the occurrences.
- If only one candidate is available (say c_1), adopt it as the overall candidate.
- If neither half provides a candidate, conclude that no majority exists.

(e) **Verification:** Perform a final pass over all n cards. For each card, use the equivalence tester to check if it is equivalent to the candidate c. Count the total number k of cards equivalent to c.

- If $k > \frac{n}{2}$, then c is indeed the majority candidate.
- Otherwise, report that no majority equivalence class exists.

Recurrence Relation: At each recursion level, we process two subproblems of size $\frac{n}{2}$ and then spend $O(n)$ time in the combine (verification) step. Thus, the recurrence is:

$$T(n) = 2T\left(\frac{n}{2}\right) + O(n)$$

This relation reflects the cost of recursively determining candidates for both halves and then performing a linear scan to verify the candidate in the merged set.

Solving the Recurrence using the Master Theorem: Here, we have:

$$a = 2, \quad b = 2, \quad \text{and} \quad f(n) = O(n)$$

We compute:

$$n^{\log_b a} = n^{\log_2 2} = n.$$

Since $f(n) = \Theta(n)$, which is of the same order as $n^{\log_b a}$, we are in Case 2 of the Master Theorem. Therefore, the recurrence solves to:

$$T(n) = \Theta(n \log n).$$

This shows that the algorithm requires $O(n \log n)$ invocations of the equivalence tester.

Rubrics (15 points):

- (a) 7 points for a correct and detailed description of the divide-and-conquer algorithm, including the base case, how the set of cards is divided, the combination of candidates from the two halves, and the verification step.

- (b) 3 points for correctly formulating the recurrence relation $T(n) = 2T(n/2) + O(n)$ and providing a clear explanation of its derivation.
- (c) 5 points for correctly applying the Master Theorem to solve the recurrence and obtaining the final time complexity $\Theta(n \log n)$.

4. Emily has received a set of marbles as her birthday gift. She is trying to create a staircase shape with her marbles. A staircase shape contains k marbles in the kth row. Given n as the number of marbles, help her to figure out the number of rows of the largest staircase she can make. (Time complexity $< O(n)$)

   For example, a staircase of size 4 looks like:

```
*
* *
* * *
* * * *
```

   (a) Describe your algorithm. It **must** be a divide-and-conquer algorithm. (5 points)

   (b) Give your algorithm's recurrence relation for runtime complexity. Briefly explain it. (5 points)

   (c) Solve the recurrence relation and state the overall time complexity. (5 points)

Let $g(n)$ be the function indicating the number of marbles in a staircase of size n. Then:

$g(n) = n(n+1)/2$

The goal is to find the largest number $1 \leq k \leq n$ such that: $g(k) \leq n < g(k+1)$

(a) **Algorithm Description:**

We use a binary search algorithm to find k.

- Initialize `lbound = 1` and `ubound = n`.
- While `lbound <= ubound`:
  - `k = lbound + (ubound - lbound) / 2` (Prevent potential overflow)
  - If $g(k) = n$, then k is the solution; return k.
  - If $g(k) > n$, set `ubound = k - 1`.
  - If $g(k) < n$, set `lbound = k + 1`.
- Return `ubound`. (After the loop, `ubound` is the largest k such that $g(k) \leq n$)

(b) **Recurrence Relation:**

The algorithm uses binary search. Each step involves a constant amount of work (calculating $g(k)$ and comparisons) and then reduces the problem size by half. Therefore, the recurrence relation is:

$$T(n) = T(n/2) + O(1)$$

where $T(n)$ is the time complexity for an input of size n. The $O(1)$ represents the constant time operations in each step.

(c) **Solving the Recurrence and Time Complexity:**

We can solve the recurrence $T(n) = T(n/2) + O(1)$ using the Master Theorem. Here, $a = 1$, $b = 2$, and $f(n) = O(1)$. We have $n^{\log_b a} = n^{\log_2 1} = n^0 = 1$. Since $f(n) = \Theta(1)$, we are in Case 2 of the Master Theorem. Therefore:

$T(n) = \Theta(\log n)$

The overall time complexity of the algorithm is $O(\log n)$.

Rubrics (15 points):

- (a) 5 points: Correct description of the binary search algorithm, including initialization, loop condition, calculation of the middle element, comparison logic, and update rules for the bounds. Correctly returning `ubound`.
- (b) 5 points: Correctly formulating the recurrence relation $T(n) = T(n/2) + O(1)$ and providing a brief explanation of its derivation (halving the search space).
- (c) 5 points: Correctly applying the Master Theorem to solve the recurrence and stating the final time complexity as $O(\log n)$ or $\Theta(\log n)$.

Ungraded Questions

1. Solve Kleinberg and Tardos, Chapter 5, Exercise 5.

   We describe a divide-and-conquer algorithm that computes the visible lines (i.e., the upper envelope) of a set $L = \{L_1, L_2, \ldots, L_n\}$ of lines sorted by increasing slope.

   Step 1. Base Case: When the input set contains only one line, return that line as visible.

   Step 2. Divide: Divide L into two halves:

   $$L_{\text{left}} = \{L_1, L_2, \ldots, L_{\lfloor n/2 \rfloor}\} \quad \text{and} \quad L_{\text{right}} = \{L_{\lceil n/2 \rceil}, \ldots, L_n\}.$$

   Recursively compute the sorted sequence of visible lines for each half. In addition, for each half, compute the sorted set of intersection points between consecutive visible lines. For the left half, let

   $$A = \{a_1, a_2, \ldots, a_{m-1}\},$$

   where a_j is the x-coordinate of the intersection of the jth and $(j+1)$th visible lines in L_{left}; similarly, for the right half, let

   $$B = \{b_1, b_2, \ldots, b_{r-1}\},$$

   where b_j is the intersection of consecutive visible lines in L_{right}.

   Step 3. Conquer (Merge): The key observation is that the visible portions of the lines (i.e., the upper envelope) form a boundary that is monotonic in x. In particular, since the lines are sorted by slope, if two visible lines intersect then the one with the smaller slope is uppermost to the left of the intersection point. To merge the two envelopes, we need to locate the x-coordinate at which the boundary for the left half meets the boundary for the right half.

   More specifically, parse the two sorted lists of visible lines to find the smallest index where a line from L_{left} lies below a line from L_{right}. Let

   $$L_{\text{up-left}}(s) = L_{i_s} \quad (\text{from } L_{\text{left}})$$

   and

   $$L_{\text{up-right}}(t) = L_{k_t} \quad (\text{from } L_{\text{right}})$$

   be the corresponding lines, and let (x^*, y^*) be their intersection point. This intersection point partitions the x-axis so that to the left of x^* the upper envelope is defined by the visible lines of L_{left}, while to the right of x^* it is defined by those of L_{right}.

   Thus, the merged set of visible lines for L is given by taking the visible lines from L_{left} up to L_{i_s} and the visible lines from L_{right} starting at L_{k_t}:

   $$\{L_{i_1}, L_{i_2}, \ldots, L_{i_{s-1}}, L_{i_s}, L_{k_t}, L_{k_{t+1}}, \ldots, L_{k_r}\}.$$

This merge step, which involves scanning through the sorted lists A and B, can be accomplished in $O(n)$ time.

Overall Runtime: Let $T(n)$ denote the time to compute the visible lines for a set of n lines. The algorithm divides the problem into two subproblems of size $n/2$ each, and the merge step takes $O(n)$ time. Thus, we have the recurrence:

$$T(n) = 2T\left(\frac{n}{2}\right) + O(n).$$

By the Master Theorem (with $a = 2$, $b = 2$, and $f(n) = O(n)$), we obtain

$$T(n) = O(n \log n).$$

This completes the description of an $O(n \log n)$ algorithm to compute the visible lines.

2. Assume that you have a blackbox that can multiply two integers in one call. Describe an algorithm that, when given an n-bit positive integer exponent a and an integer x, computes x^a using at most $O(n)$ calls to the blackbox.

   You need to:

   (a) Describe your algorithm.

   (b) Provide the recurrence relation for the number of blackbox calls and explain its derivation.

   (c) Solve the recurrence to show that the algorithm uses $O(n)$ calls to the blackbox.

   We can compute x^a efficiently using the technique known as exponentiation by squaring (or fast exponentiation). The idea is to reduce the computation of x^a to a smaller exponent in each step. In particular, we observe that:

   - If a is even, then
   $$x^a = \left(x^{\lfloor a/2 \rfloor}\right)^2.$$
   - If a is odd, then
   $$x^a = \left(x^{\lfloor a/2 \rfloor}\right)^2 \times x.$$

   Based on this observation, we can describe the algorithm as follows:

   Algorithm Description:

   (a) **Base Case:** If $a = 0$, return 1 (since $x^0 = 1$); if $a = 1$, return x.

   (b) **Recursive Step:**

   i. Compute $y = x^{\lfloor a/2 \rfloor}$ by a recursive call.

   ii. If a is even, then compute x^a as:
   $$x^a = y \times y.$$

   iii. If a is odd, then compute x^a as:
   $$x^a = y \times y \times x.$$

   In each recursive call, the exponent is effectively halved (i.e., its bit-length decreases by one). The multiplications (performed via the blackbox) used in the combination step are at most three per call (two multiplications when a is even and three when a is odd).

   Recurrence Relation: Let $T(n)$ denote the number of blackbox calls required when the exponent a is an n-bit number. Since each recursive call reduces the bit-length by 1 and uses at most 3 additional multiplications, we have:
   $$T(n) \leq T(n-1) + 3.$$

Solving the Recurrence: We can unroll the recurrence:

$$T(n) \leq T(n-1) + 3 \leq T(n-2) + 3 + 3 \leq \cdots \leq T(1) + 3(n-1).$$

Since $T(1)$ is a constant (say, at most 3), it follows that

$$T(n) = O(n).$$

Thus, the algorithm computes x^a using $O(n)$ calls to the blackbox.