ShanghaiTech University

CS240 Algorithm Design and Analysis

Fall 2025

Midterm

Time: 13:00-14:40, Apr. 1, 2025

1. This exam is closed-book, but you may bring one A4-size cheat sheet. Put all the study materials and electronic devices into your bag and put your bag in the front, back, or sides of the classroom.
2. For short-answer questions, please write your responses in the designated answer box.
3. You can write your answers only in English.
4. Two blank pieces of paper are attached, which you can use as scratch paper. Raise your hand if you need more paper.

1 True or False (20 pts)

Determine whether the following statements are correct. Please fill in the corresponding box with the answer, T for True and F for False.

1	2	3	4	5
6	7	8	9	10

1. It is possible to implement matrix multiplication with a time complexity of $o(n^3)$.
2. Consider the recurrence relation: $T(n) = 4T(n/2) + O(n^2)$. The time complexity of this recurrence relation is $O(n^2 \log n)$.
3. Given two recurrence relation $T(n) = T(0.01n) + T(0.02n) + \Theta(n)$ and $S(n) = S(0.99n) + \Theta(1)$ where $T(0) = S(0) = 0$ and $T(1) = S(1) = 1$, then $T(n) = O(S(n))$.
4. To prove the correctness of a greedy algorithm, we must prove that every optimal solution contains our greedy choice.
5. Greedy algorithms are always applicable to solve the shortest path problem.
6. The core idea of a greedy algorithm is to choose the locally optimal solution at each step to move toward solving the problem.
7. Compared with top-down dynamic programming, bottom-up may skip some unnecessary subproblems.
8. In dynamic programming algorithms, subproblems must be independent of each other.
9. If all edge capacities are integers, then the Ford–Fulkerson algorithm might produce a final flow with fractional values on some edges.
10. The value of an $s - t$ cut is always an upper bound on the value of any feasible $s - t$ flow (where s and t are source and sink respectively in a network).

Solution:

1. T
2. T
3. F
4. F
5. F
6. T
7. F
8. F
9. F
10. T

2 Divide and Conquer (20 pts)

a. Given a sorted array of distinct integers $A[1, ..., n]$, we want to find out whether there is an index i for which $A[i] = i$. We can solve this problem using a divide-and-conquer algorithm that runs in time $O(\log n)$. Give a detail explanation of the algorithm and analyze its time complexity. (8 pts)

b. Given an unsorted array of distinct integers $B[1, ..., n]$, find the k-th smallest element in the array where k $\in [1, n]$. We can solve this problem using a divide-and-conquer algorithm that runs in time $O(n)$. Give a detail explanation of the algorithm and analyze its time complexity. (12 pts)

Solution:

Part (a):

1. **Algorithm:** We can use binary seacrh to solve this problem. We start by checking the middle element of the array. If $A[\frac{n}{2}] = \frac{n}{2}$, we have found the desired index. Otherwise, we can recursively search the left half if $A[\frac{n}{2}] > \frac{n}{2}$ or the right half if $A[\frac{n}{2}] < \frac{n}{2}$ when the array is ascending, vice versa.

2. **Running Time:** As we check by half, we need at most $O(\log n)$ steps to find the index i. $T(n) = T(n/2) + O(1)$

Part (b): First, you can answer this question by referring to the slides **03 Divide and Conquer2** pages 4-12 or **Introduction to Algorithms** chapter 9.2-9.3.

1. **Algorithm:** We can use the quicksort-like algorithm to solve this problem.

   - Choose a pivot as the median of median(you can find it with the worest time complexity $O(n)$) with index p.
   - Partition the array into two parts: $B[1, ..., p-1]$, where all elements are guaranteed to be less than $B[p]$ and $B[p+1, ..., n]$, where all elements are guaranteed to be greater than $B[p]$.
   - Let r be the rank of the pivot in the array. If $r = \mathrm{k}$, return $B[p]$.
   - Otherwise, recursively search the left half if $r > \mathrm{k}$ or the right half if $r < \mathrm{k}$(search $\mathrm{k} - r^{\mathrm{th}}$).
   - The algorithm terminates when $r = \mathrm{k}$.

2. **Running Time:** Since we choose the pivot as median of median, we reduce the array size at least 3/10 each time. Therefore, the running time is

$$T(n) = T(n/5) + T(7n/10) + O(n) = O(n)$$

In addition, if we do not employ the median of median strategy but only use quick select with random pivot selection, it could potentially result in a worst-case time complexity of $O(n^2)$.

By heap-sort-like select, you can also solve this question with $O(n + k \log n)$.

This question can be practiced on LeetCode. Follow the link https://leetcode.cn/problems/kth-largest-element-in-an-array.

3 Scheduling to minimize average completion time (20 pts)

Suppose you are given a set $S = \{a_1, a_2, \ldots, a_n\}$ of tasks, where task a_i requires p_i units of processing time to complete, once it has started. You have one computer on which to run these tasks, and the computer can run only one task at a time. Let c_i be the *completion time* of task a_i, that is, the time at which task a_i completes processing. Your goal is to minimize the average completion time, that is, to minimize

$$\frac{1}{n}\sum_{i=1}^{n} c_i.$$

For example, suppose there are two tasks, a_1 and a_2, with $p_1 = 3$ and $p_2 = 5$, and consider the schedule in which a_2 runs first, followed by a_1. Then $c_2 = 5$, $c_1 = 8$, and the average completion time is $(5 + 8)/2 = 6.5$. If task a_1 runs first, however, then $c_1 = 3$, $c_2 = 8$, and the average completion time is $(3 + 8)/2 = 5.5$.

a. Give an algorithm that schedules the tasks so as to minimize the average completion time. Each task must run non-preemptively, that is, once task $a_i, i \in [1, n]$, starts, it must run continuously for p_i units of time. Prove that your algorithm minimizes the average completion time, and state the running time of your algorithm. (8pts)

b. Suppose now that the tasks are not all available at once. That is, each task cannot start until its *release time* r_i,$i \in [1, n]$. Suppose also that we allow *preemption*, so that a task can be suspended and restarted at a later time. For example, a task a_i with processing time $p_i = 6$ and release time $r_i = 1$ might start running at time 1 and be preempted at time 4. It might then resume at time 10 but be preempted at time 11, and it might finally resume at time 13 and complete at time 15. Task a_i has run for a total of 6 time units, but its running time has been divided into three pieces. In this scenario, a_i's completion time is 15. Give an algorithm that schedules the tasks so as to minimize the average completion time in this new scenario. Prove that your algorithm minimizes the average completion time, and state the running time of your algorithm. (12pts)

Solution:

Answer (a):

- **Optimal Strategy:** Using the **Shortest Job First (SJF)** scheduling algorithm. This means that we should always schedule the task with the smallest processing time.
- **Proof:**
  - The algorithm schedules tasks in increasing order of their processing times.
  - Suppose tasks are ordered such that $p_1 \leq p_2 \leq \cdots \leq p_n$.
  - The completion time c_i of task a_i where $i \in [1, n]$ if the tasks are scheduled in this order, is the cumulative processing time of all previous tasks plus the processing time of a_i:

$$c_1 = p_1, \quad c_2 = p_1 + p_2, \quad \ldots, \quad c_n = \sum_{i=1}^{n} p_i$$

  - The average completion time is:

$$\frac{1}{n}\sum_{i=1}^{n} c_i = \frac{1}{n}\left(p_1 + (p_1 + p_2) + \cdots + \sum_{i=1}^{n} p_i\right)$$

    This formula shows that scheduling in increasing order of processing times minimizes the total sum of completion times. If we schedule a_j before a_i, the completion time of a_j will be higher than if we had scheduled a_i first. This would lead to a higher average completion time because the larger task delays the completion of the smaller task.
- Running Time:
  - Sorting the tasks by processing time takes $O(n \log n)$.
  - Calculating the cumulative completion times takes $O(n)$.
  - Thus, the total running time of the algorithm is $O(n \log n)$.

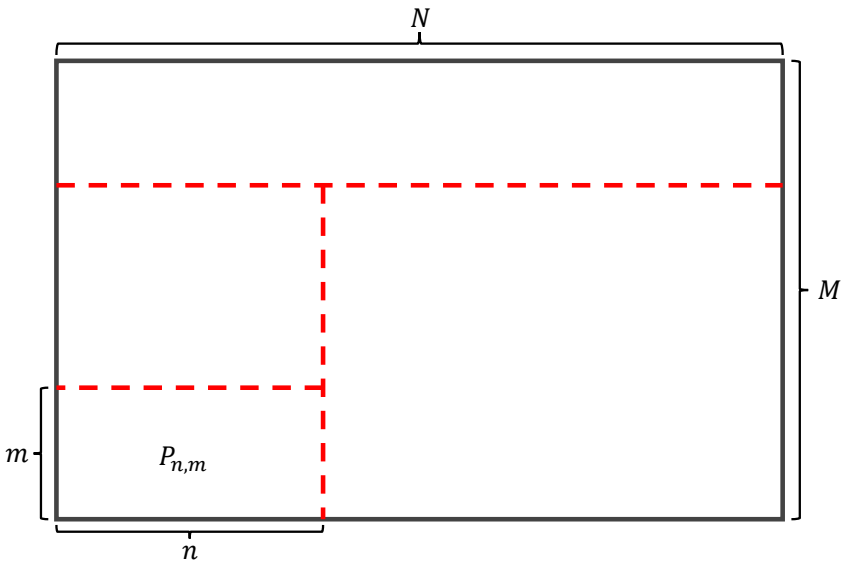
Answer (b):

We can apply a modified version of the **Shortest Job First (SJF)** algorithm, called **Shortest Remaining Time First (SRTF)**, which takes into account the remaining processing time of each task and allows preemption. Additionally, we need to consider the release times of tasks. A task can only be scheduled once its release time has passed.

- **Algorithm:**
  1. Sort the tasks by their release times, so the first task starts as soon as it becomes available.
  2. Maintain a priority queue (or heap) to keep track of the tasks that are available to run, where tasks with the smallest remaining time are given the highest priority.
  3. At each time unit:
     - Check if any new tasks have become available (i.e., tasks whose release times have passed).
     - If no task is currently running, select the task with the shortest remaining time from the available tasks and start it.
     - If a task is running and a new task arrives with a shorter remaining time, preempt the current task and start the new task.
  4. Continue this process until all tasks are completed.
- **Proof:**
  - Suppose the completion time of task a_i is c_i where $i \in [1, n]$ and the average completion time of SRJF is AVG_{OPT}.
    Suppose there exists a better solution OPT'. Then there will be a time point t_i, where task a_i was running in OPT, exchanged to another task a_j. However, under the consumption of SRJF, a_i has the least left time to complete at time point t_i, the compile time for the rest task will increase together in OPT', leading to a larger average completion time, which is a conradiction.
    Thus, $\text{AVG}_{\text{OPT'}} \geq \text{AVG}_{\text{OPT}}$ and our greedy algorithm is the optimal.
  - The SRTF algorithm minimizes the remaining processing time at every step, ensuring that the task with the shortest remaining time is always executed first. This minimizes the total waiting time and, hence, the average completion time.
- **Running Time:**
  - Sorting the tasks by release times takes $O(n \log n)$.
  - Each operation of adding a task to or removing a task from the priority queue takes $O(\log n)$.
  - The total time complexity is $O(n \log n)$ due to the sorting and heap operations.

Thus, the time complexity of the algorithm for part (b) is $O(n \log n)$.

4 Uncharted (20 pts)



On an ordinary afternoon, the renowned explorer Nathan Drake found you and presented you with a rectangular stone tablet inscribed with ancient characters. Rather than donating the tablet to the government, he thought to maximize his profit by selling it on the black market. The stone tablet has a length of N and a width of M. It can be cut, but each cut can only be made horizontally or vertically, dividing the rectangular tablet into two separate rectangular pieces. Suppose the selling price of a stone tablet with a length of n and a width of m is given by $P_{n,m}$, and the cost of making a cut along the direction of length L is L. Your task is to design an algorithm of dynamic programming to determine the optimal cutting strategy that maximizes profit.

Importantly, you need to write down the **Initial Sate**, **State Transition Equation** and analyze the **Time Complexity** of the algorithm. It's worthy noticing that:

- Whenever you cut a rectangular piece of stone tablet, you are required to cut the piece all the way across.
- You should assume that all cuts are made on integer boundaries and that the width of the saw blade is negligible. Thus any pieces you create have integer dimensions.
- You should not make any assumptions about the market prices. For example, a smaller rectangle might have a higher market price than a larger rectangle.

Solution:

State Definition

We define the state $dp[i][j]$ to represent the maximum profit obtainable from a rectangular stone tablet of size $i \times j$, where $1 \leq i \leq N$ and $1 \leq j \leq M$.

State Transition Equation

For each $i \times j$ rectangle, we consider all possible ways to cut it. Specifically, we can choose to make either a horizontal or vertical cut. For each cutting option, we calculate the sum of the profits of the two resulting sub-rectangles and subtract the cost of the cut.

- Horizontal Cut: For each possible horizontal cutting position k (where $1 \leq k < i$), we cut the rectangle into two sub-rectangles of sizes $k \times j$ and $(i-k) \times j$. The cost of the cut is j dollars (since the length of the cut is j inches). Therefore, the profit from a horizontal cut can be expressed as:

$$dp[k][j] + dp[i-k][j] - j$$

- Vertical Cut: For each possible vertical cutting position l (where $1 \leq l < j$), we cut the rectangle into two sub-rectangles of sizes $i \times j$ and $i \times (j-l)$. The cost of the cut is i dollars (since the length of the cut is i inches). Therefore, the profit from a vertical cut can be expressed as:

$$dp[i][l] + dp[i][j-l] - i$$

- No Cut: We can also choose not to cut the rectangle and sell the entire $i \times j$ rectangle as is. In this case, the profit is simply $P_{i,j}$.

Thus, the state transition equation can be written as:

$$dp[i][j] = \max(P_{i,j}, \max_{1 \leq k < i}(dp[k][j] + dp[i-k][j] - j), \max_{1 \leq l < j}(dp[i][l] + dp[i][j-l] - i))$$

Initial Conditions and Final Answer

For any i, j, we have $dp[i][j] = P_{i,j}$.

The maximum profit is $dp[N][M]$.

Complexity Analysis

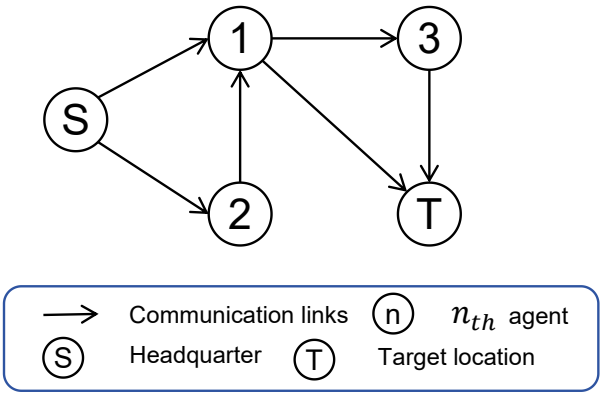
For each i and j, we need to consider $O(i+j)$ cutting options. Therefore, the overall time complexity is $O(N^2M + NM^2)$

5 Spy Network Routing (20 pts)

You are a spy master coordinating agents in a covert network (Directed Acyclic Graph). Agents communicate through secure relay stations (vertices). A message route is a simple directed path from s to t in the network.

- **Source (s):** Headquarters.
- **Sink (t):** A high-value target location.
- **Edges:** Directed communication links between stations.

Our mission is to find the max number of *secure message routes* from headquarters to the target and find these routes. A route is *secure* if every relay station in this route (except s and t) is not shared by other routes.



Part (a) How many *secure message routes* in the figure above? Please also specify which routes. If there are multiple combinations of solution, please write all possible solutions (e.g., Solution 1: {route$_1$, route$_2$, ...}, Solution 2: {route$_1$, route$_2$, ...}, ...). (8 pts)

Part (b) Given the spy network $G = (V, E)$, design an algorithm with network flow to find the maximum number of secure $s - t$ routes. Describe the algorithm (including two parts, which are how to find the number of such routes and how to find these routes). (12 pts)

Solution:

Part (a): There are 4 solutions, each with one secure route.
$\{s \to 1 \to 3 \to t\}$
$\{s \to 1 \to t\}$
$\{s \to 2 \to 1 \to 3 \to t\}$
$\{s \to 2 \to 1 \to t\}$

Part (b): To ensure relay stations (vertices) are not reused, transform the original directed graph $G = (V, E)$ into a flow network $G' = (V', E')$:

- **Vertex Splitting:** For each relay station $v \in V \setminus \{s, t\}$, split it into two nodes:
  - v_{in}: Entry checkpoint for v.
  - v_{out}: Exit checkpoint for v.

  Add an edge $v_{\text{in}} \to v_{\text{out}}$ with capacity 1 to restrict v to at most one path.
- **Edge Redirection:** For each original directed link $(u, v) \in E$:
  - If $u = s$: Connect $s \to v_{\text{in}}$ (capacity 1).
  - If $v = t$: Connect $u_{\text{out}} \to t$ (capacity 1).
  - Otherwise: Connect $u_{\text{out}} \to v_{\text{in}}$ (capacity 1).

The maximum number of secure routes corresponds to the maximum flow in G':

- Use **Ford–Fulkerson's algorithm** (or other algorithms for maximum flow) for unit-capacity networks.
- The maximum flow value equals the number of *secure message routes* $s - t$ paths in G.

To find *secure message routes*, decompose the flow into paths:

- Traverse the residual graph from s to t, collecting edges with flow 1.
- Collapse $v_{\text{in}} \to v_{\text{out}}$ into v to recover the original relay stations.
- Each path in G' corresponds to a secure route in G.