Homework 5

● Graded

1 Day, 22 Hours Late

Student

Nick Zhu

Total Points

98 / 100 pts

Question 1

Problem 1 — **35** / 35 pts

1.1 (a) — **5** / 5 pts

✓ **− 0 pts** Correct

1.2 (b) (i) — **10** / 10 pts

✓ **− 0 pts** Correct

1.3 (b) (ii) — **10** / 10 pts

✓ **− 0 pts** Correct

1.4 (b) (iii) — **10** / 10 pts

✓ **− 0 pts** Correct

Question 2

Problem 2 — **28** / 30 pts

2.1 (a) — **13** / 15 pts

✓ **− 2 pts** Base Case (1 wrong/missing)

2.2 (b) — **15** / 15 pts

✓ **− 0 pts** Correct

Question 3

Problem 3 — **35** / 35 pts

3.1 (a) — **10** / 10 pts

✓ **− 0 pts** Correct

3.2 (b) — **10** / 10 pts

✓ **− 0 pts** Correct

3.3 (c) — **15** / 15 pts

✓ **− 0 pts** Correct

- **Due Date: Thursday, March 6th, 23:59**
- **Late submission** will be accepted **without** penalty until **Saturday, March 8th, 23:59**.
- **No submissions will be accepted after March 8th, 23:59.**

Problem 1 (Optimizing Radix Sort for Sensor Data, 35 points)

A smart city infrastructure system collects sensor readings from n different locations every second. Each sensor generates an integer in the range of $[0, M)$ representing environmental data, such as temperature, air quality, or traffic flow. To efficiently process and analyze this data in real-time, your goal is to sort the sensor readings as quickly as possible using radix sort.

(a) Recall that the radix sort algorithm we learned in class utilizes counting sort as a subroutine, and achieves a worst case runtime of $\Theta(d(n+k))$. In short sentences, explain the meaning of d, n, and k in this expression.

(b) To optimize the runtime of radix sort (using counting sort as the subroutine), you must choose appropriate values for d and k. For each of the following scenarios, determine the best possible asymptotic (worst-case) runtime of radix sort in terms of n. Also, specify the values of d and k (as functions of n) that achieve this runtime, and briefly justify why your choice leads to an (asymptotically) optimal runtime.

Example: If $M = n$, the best asymptotic runtime you can get is $\Theta(n)$ by picking $d = 1$ and $k = n$. This is optimal because d needs to be at least 1, so the runtime is at least $\Omega(1(n+k)) = \Omega(n)$. (Your justification might be very different for the following cases.)

(i) $M = n^3$.
(ii) $M = 2^n$.
(iii) $M = n!$.

(*Hint for (ii) and (iii): First, express the runtime in terms of n and k. Then, analyze the asymptotic runtime separately for the cases $k \leq n$ and $k \geq n$. Finally, think about how these two expressions change with k.*)

Solution

(a)

- n is the number of sensor readings to be sorted.
- k is the range of each digit in the counting sort subroutine.
- d is the number of digits required to represent the numbers in the chosen base, which is determined by $d = \log_k M$, where M is the range of sensor readings.

(b)

(i) **Case:** $M = n^3$

Questions assigned to the following page: 1.3, 1.4, and 1.2

Choose $k = \Theta(n)$. Then,

$$d = \log_k(n^3) = \log_n(n^3) = 3.$$

Since each pass of counting sort takes $\Theta(n+k) = \Theta(n)$ time (because $k = \Theta(n)$), the total runtime is

$$\Theta(3 \cdot n) = \Theta(n).$$

This is optimal because we minimize the number of passes and the cost per pass remains linear in n.

(ii) **Case:** $M = 2^n$

Again, choose $k = \Theta(n)$. Then,

$$d = \log_k(2^n) = \frac{n}{\log_2 k} = \Theta\Big(\frac{n}{\log n}\Big).$$

Each pass takes $\Theta(n+k) = \Theta(n)$, so the overall runtime is

$$\Theta\Big(\frac{n}{\log n} \cdot n\Big) = \Theta\Big(\frac{n^2}{\log n}\Big).$$

This choice optimally balances the number of passes against the per-pass cost.

(iii) **Case:** $M = n!$

In class, we have already showed that $\log(n!) = \Theta(n \log n)$. Again, set $k = \Theta(n)$, so that

$$d = \log_k(n!) = \Theta\left(\frac{n \log n}{\log n}\right) = \Theta(n).$$

With each pass costing $\Theta(n+k) = \Theta(n)$, the total runtime becomes

$$\Theta(n \cdot n) = \Theta(n^2).$$

Choosing $k = \Theta(n)$ minimizes the combined cost of the number of passes and the per-pass work.

Problem 2 (Rod Cutting without Repetition, 30 pts)

Recall that in the (vanilla) *rod cutting problem* we are given a rod of length n and a list of n prices (assume all prices are positive), where p_i is the price fetched by a piece of length i. The goal is to calculate the maximum revenue you can generate from cutting the rod.

In the *rod cutting without repetition* problem, your discerning customers desire to be unique, and hence refuse to buy a length of rod that has been bought before (weird assumption, I know. But let's say this customer is an enthusiastic rod collector, and aims to collect rods of different lengths ¯_(ツ)_/¯). Namely, even if you cut two pieces of length i, the total contribution to revenue from these pieces is just p_i (not $2p_i$ as in the vanilla version).

In this problem, you will design a dynamic programming algorithm for solving the rod cutting without repetition in polynomial time.

(a) Start by writing a recurrence relation for this problem (make sure to state clearly what the base cases are) and justifying its correctness (i.e. why does this problem exhibit optimal substructure). In class, our recurrence relation had just one parameter, the length of the rod. Here, you can introduce the function $T(n, i)$ with two parameters n and i, that is equal to the optimal revenue for the rod of length n when we cut it into pieces of length at most i. You can then write the recurrence for $T(n, i)$.

(b) Next, write pseudo-code for your algorithm (either memoized recursion or bottom-up) and (very briefly) justify its run-time.

Solution

Let $T(n, i)$ represents the optimal revenue for the rod of length n when we cut it into pieces of length at most i (with each piece length used at most once).

(a)

$$T(n,i) = \begin{cases} 0, & \text{if } n = 0, \\ -\infty, & \text{if } n > 0 \text{ and } i = 0, \\ \max\Big\{T(n, i-1),\ p_i + T(n-i, i-1)\Big\}, & \text{if } n \geq i, \\ T(n, i-1), & \text{if } n < i. \end{cases}$$

- **Base Cases:**
  - $T(0, i) = 0$ for all i, since a rod of length 0 yields zero revenue.
  - $T(n, 0) = -\infty$ for $n > 0$ because if no pieces are available, we cannot obtain any revenue.
- **Recurrence:** For $n \geq i$, we have two choices:
  1. *Do not use* the piece of length i. In this case, the maximum revenue is $T(n, i-1)$.
  2. *Use* the piece of length i, gaining revenue p_i, and then solve the subproblem for the remaining rod of length $n - i$ with pieces up to $i - 1$ (i.e., $T(n-i, i-1)$). Note that even if more than one piece of length i could fit, we can only count it once.

- If $n < i$, the piece of length i cannot be used, so we simply have $T(n, i) = T(n, i - 1)$.

Optimal Substructure: The decision to include or exclude a piece of a particular length i leads to subproblems involving a smaller rod length and a reduced set of available pieces. Since the optimal solution to the entire problem is composed of optimal solutions to these subproblems and the recurrence covers all possible cuttings, the problem exhibits optimal substructure.

(b)
We use a bottom-up dp solution that fills in a table $T[i][j]$ where $T[i][j]$ denotes the maximum revenue for a rod of length i using pieces of lengths at most j.

Pseudocode:

```
RodCuttingWithoutRepetition($n, p$)
1: Initialize a 2D array $T[0 \dots n][0 \dots n]$
2: for $j \leftarrow 0$ to $n$ do
3:     $T[0][j] \leftarrow 0$
4: end for
5: for $i \leftarrow 1$ to $n$ do
6:     $T[i][0] \leftarrow -\infty$
7: end for
8: for $j \leftarrow 1$ to $n$ do
9:     for $i \leftarrow 1$ to $n$ do
10:        if $i \geq j$ then
11:            $T[i][j] \leftarrow \max\{ T[i][j-1],\ p[j] + T[i-j][j-1] \}$
12:        else
13:            $T[i][j] \leftarrow T[i][j-1]$
14:        end if
15:    end for
16: end for
17: return $T[n][n]$
```

Runtime Justification:

- **Initialization:**
  - The first `for` loop (lines 2–4) initializes $T[0][j]$ for $j = 0$ to n, taking $O(n)$ time.
  - The second `for` loop (lines 5–7) initializes $T[i][0]$ for $i = 1$ to n, taking another $O(n)$ time.
- **Table Filling:**
  - The nested loops (lines 8–15) iterate over $j = 1$ to n and, for each j, over $i = 1$ to n. This results in $n \times n = n^2$ iterations.
  - Within each iteration, the operations (comparison, addition, and `max` computation) take constant time, i.e., $O(1)$.
- **Overall Runtime:**

- The initialization steps contribute $O(n)$ time each.
- The table filling step contributes $O(n^2)$ time.
- Thus, the total runtime is $O(n) + O(n) + O(n^2) = O(n^2)$.

Questions assigned to the following page: 3.1, 3.3, and 3.2

Problem 3 (The Coin Game, 35 points)

Consider a row of n coins of values $v_1, v_2, \ldots, v_n$, where n is even. A turn-based game is being played between 2 players where they alternate turns. In each turn, a player selects either the first or last coin from the row, removes it from the row permanently, and receives the value of the coin. We want an algorithm that determines the maximum possible amount of money you can definitely win if you move first (assume your opponent will also play to maximize the amount they get). To this end, we will use dynamic programming. Define the subproblems $\mathsf{MaxGain}(i, j)$ (for $i \le j$) to be the maximum guaranteed payoff for the first player if given the subarray $[v_i, \ldots, v_j]$.

(a) Suppose the row of coins is $A = [1, 1, 1, 2, 2, 2]$. If you move first and try to maximize your gains, how much money can you definitely win regardless of the opponent's moves? Describe in words your optimal play strategy for this particular instance.

(b) State the base cases for $\mathsf{MaxGain}(i, j)$ and their values.

(c) Give the overall DP algorithm for $\mathsf{MaxGain}$ (just stating the recurrence is sufficient). Justify the correctness and runtime of your proposed algorithm.

Solution

(a)
I can guarantee a payoff of **5**.
I will start with the last coin (with value 2). This move forces the remaining coins into a configuration where, regardless of whether my opponent picks from the beginning or the end, I can always continue with moves that secure me a total gain of 5 (since I can always get one of the two remaining 2s).

(b) Base Cases for $\mathsf{MaxGain}(i, j)$:

- **Single Coin:** When $i = j$, there is only one coin available. Thus,

$$\mathsf{MaxGain}(i, i) = v_i.$$

- **Two Coins:** When $j = i + 1$, there are two coins available. Since the first player can choose the coin with the higher value,

$$\mathsf{MaxGain}(i, i + 1) = \max\{v_i, v_{i+1}\}.$$

(c)
1. Subproblem: Define $\mathsf{MaxGain}(i, j)$ as the maximum guaranteed payoff for the first player when the game is played on the subarray $[v_i, v_{i+1}, \ldots, v_j]$. This subproblem represents the situation in which only the coins from positions i to j remain, and both players play optimally from that point onward.

2. Guess: At any turn, the first player has two choices:

- Pick the coin v_i at the left end.

Question assigned to the following page: 3.3

- Pick the coin v_j at the right end.

3. Recurrence: When the first player makes a move, the opponent is left with a smaller subarray and will choose in a way that minimizes the first player's future gain. Hence, the recurrence is:

$$\mathsf{MaxGain}(i, j) = \max \begin{cases} v_i + \min\{\mathsf{MaxGain}(i+2, j),\ \mathsf{MaxGain}(i+1, j-1)\}, \\ v_j + \min\{\mathsf{MaxGain}(i+1, j-1),\ \mathsf{MaxGain}(i, j-2)\} \end{cases}$$

The two cases represent:

- If the first player picks v_i, the opponent will then choose either v_{i+1} or v_j, leaving the first player with subarrays $[v_{i+2}, \ldots, v_j]$ or $[v_{i+1}, \ldots, v_{j-1}]$. Since the opponent minimizes your gain, you receive v_i plus the minimum of the two resulting $\mathsf{MaxGain}$ values.
- Similarly, if the first player picks v_j, the gain is v_j plus the minimum between $\mathsf{MaxGain}(i+1, j-1)$ and $\mathsf{MaxGain}(i, j-2)$.

Base Cases:

- When $i = j$: $\mathsf{MaxGain}(i, i) = v_i$.
- When $j = i + 1$: $\mathsf{MaxGain}(i, i+1) = \max\{v_i, v_{i+1}\}$.

4. Implementation Approach: We use a bottom-up dynamic programming approach. We fill in a 2D DP table where each entry $\mathtt{DP}[i][j]$ stores $\mathsf{MaxGain}(i, j)$, starting from the smallest subarrays (base cases) and working up to the entire array $[v_1, \ldots, v_n]$.

Correctness and Runtime Justification:

- *Correctness:* The recurrence correctly models the decision process at each turn by considering both choices and accounting for the opponent's optimal (minimizing) response. Since the problem exhibits optimal substructure and overlapping subproblems, filling in the DP table in a bottom-up manner ensures that every subproblem is solved optimally before being used in a larger problem.
- *Runtime:* The number of distinct subproblems is $O(n^2)$ because there are $\Theta(n^2)$ pairs (i, j) with $1 \leq i \leq j \leq n$. Each subproblem is solved in $O(1)$ time (performing a constant number of arithmetic and comparison operations). Therefore, the overall time complexity is $O(n^2)$.