ShanghaiTech University

CS101 Algorithms and Data Structures
Fall 2024

Homework 11

Due date: December 22, 2024, at 23:59

1. Please write your solutions in English.
2. Submit your solutions to Gradescope.
3. Set your FULL name to your Chinese name and your STUDENT ID correctly in Gradescope account settings.
4. If you want to submit a handwritten version, scan it clearly. `CamScanner` is recommended.
5. We recommend you to write in LaTeX.
6. When submitting, match your solutions to the problems correctly.
7. No late submission will be accepted.
8. Violations to any of the above may result in zero points.

Demand of the Algorithm Design: (MUST READ!)

All of your algorithms should need the three-part solution, this will help us to score your algorithm. You should include **main idea, proof of correctness and run-time analysis.** The detail is as below:

1. The **main idea** of your algorithm. You should correctly convey the idea of the algorithm in this part. It does not need to give all the details of your solutions or explain why they are correct. When grading these problems, we will emphasize how you define your sub-problems, whether your Bellman equation is correct, and the correctness of your complexity analysis:
   (a) **Define your subproblems clearly.** Your definition should include the variables you choose for each subproblem and a brief description of your subproblem in terms of the chosen variables.
   (b) Your **Bellman equation** should be a recurrence relation whose **base case** is well-defined.
   (c) You can **briefly explain each term in the equation** if necessary, which might improve the readability of your solution and help us grade it.
2. A **proof of correctness**. You must prove that your algorithm work correctly, no matter what input is chosen. For iterative or recursive algorithms, often a useful approach is to find an invariant. A loop invariant needs to satisfy three properties: (1) it must be true before the first iteration of the loop; (2) if it is true before the ith iteration of the loop, it must be true before the $i + 1$st iteration of the loop; (3) if it is true after the last iteration of the loop, it must follow that the output of your algorithm is correct. You need to prove each of these three properties holds. Most importantly, you must specify your invariant precisely and clearly. **However, sometimes the problem does not require you to prove the correctness of your algorithm. But remember to give your bellman Equation clearly.** If you invoke an algorithm that was proven correct in class, you don't need to re-prove its correctness.
3. The asymptotic **running time** of your algorithm, stated using $\Theta(\cdot)$ notation. And you should have your **running time analysis**, i.e., the justification for why your algorithm's running time is as you claimed. Often this can be stated in a few sentences (e.g.: "the loop performs $|E|$ iterations; in each iteration, we do $\Theta(1)$ Find and Union operations; each Find and Union operation takes $\Theta(\log |V|)$ time; so the total running time is $\Theta(|E| \log |V|)$"). Alternatively, this might involve showing a recurrence that characterizes the algorithm's running time and then solving the recurrence.
4. You only need to calculate the optimal value in each problem of this homework and you don't have to back-track to find the optimal solution.

1. **(0 points) (DP Example) Maximum Subarray Problem**

Given an array $A = \langle A_1, \ldots, A_n \rangle$ of n elements, please design a dynamic programming algorithm to find a contiguous subarray whose sum is maximum.

Solution:

Main Idea

Let $OPT(I)$ be the maximum sum of subarrays of A ending with A_i. Then we have the following equation:

$$OPT(i) = \begin{cases} A_1 & \text{if } i = 1 \\ \max\{A_i, A_i + OPT(i-1)\} & \text{otherwise} \end{cases}$$

The final answer is

$$\max_{i \in \{1,2,\ldots,n\}} OPT(i)$$

Proof of Correctness:

- The 1st term in max: only take A_i
- The 2nd term in max: take A_i together with the best subarray ending with A_{i-1}

Runtime Analysis:

The running time is $\Theta(n)$ since we have a single loop with range n to reduce subproblems.

2. (8 points) Odd number of coins changing

Given n coin denominations $\{c_1, c_2, \cdots, c_n\}$ and a target value V, you are going to make change for V. However, only odd number of coins is allowed.

Please design a **dynamic programming** algorithm that find the fewest odd number of coins needed to make change for V (or report impossible).

Hints: *For this problem, you can formulate your subproblems as:*

$$F(v) = \text{fewest odd number of coins to make change for } v.$$

$$G(v) = \text{fewest even number of coins to make change for } v.$$

Solution: Main Idea:

$F(v)$ = fewest odd number of coins to make change for v, $G(v)$ = fewest even number of coins to make change for v.

$$F(v) = \begin{cases} \infty & \text{if } v < 0 \\ \min\limits_{1 \le i \le n} \{1 + G(v - c_i)\} & \text{otherwise} \end{cases}$$

$$G(v) = \begin{cases} \infty & \text{if } v < 0 \\ 0 & \text{if } v = 0 \\ \min\limits_{1 \le i \le n} \{1 + F(v - c_i)\} & \text{otherwise} \end{cases}$$

The answer should be $F(v)$.

Proof of Correctness:

- The base case is $G(0) = 0$. Define $F(v), G(v) = \infty$ for all $v < 0$ for convenience.
- $F(v)$ should be updated by $G(v - c_i)$ and $G(v)$ should be updated by $F(v - c_i)$ alternately.

Runtime Analysis:

The running time is $\Theta(nV)$ since for every value of $v \in [1, n]$, we have to enumerate all c_i, which need two nested loop with range V and n.

3. (10 points) Happiness Planning

In the magical world of Hogwarts, Professor Slughorn likes to plan his potions classes with precision and flair.

For the next m months, starting with no Galleons, Slughorn will brew potions and earn x Galleons per month. During the i-th month ($1 \leq i \leq m$), there will be a singular chance to spend c_i Galleons to acquire happiness worth h_i. However, it's a fair bargain for Slughorn to buy happiness, that is, the amount of c_i will not be too big.

Borrowing is not permitted. Galleons earned in the i-th month can only be used in a subsequent j-th month ($j > i$).

Since potion masters don't dabble with Muggle technology, please help him design a dynamic programming algorithm to assist Slughorn in discovering the maximum achievable sum of happiness.

Hints: *Still consider the backpack problem! What should you design the Bellman equation with limitation?*

Solution:

Main Idea:

Define $F(i, j)$ be the minimum cost required to achieve happiness j. In the i-th month, we iterate through $F(i, j)$ and check that whether $F(i-1, j-h_i) + c_i \leq (i-1)x$. If so, we can update $F(i, j)$ since we can afford that:

$$F(i,j) = \begin{cases} \infty & \text{if } j < 0 \\ 0 & \text{if } i = 0 \textit{ or } j = 0 \\ \min\{F(i-1,j), F(i-1,j-h_i)+c_i\} & F(i-1,j-h_i)+c_i \leq (i-1)x \\ F(i-1,j) & \text{otherwise} \end{cases}$$

Proof of Correctness:

Obviously this is a modification of the backpack problem.

Runtime Analysis:

The running time is $\Theta(n \sum c_i)$ since for every day, we need to enumerate over $\sum c_i$ to give a transform over the previous status.

4. (10 points) Greedy doesn't work

Tom and Jerry are playing an interesting game, where there are n cards in a line. All cards are faced-up and the number on every card is between 2-9. Tom and Jerry take turns. In anyone's turn, they can take one card from either the right end or the left end of the line. The goal for each player is to maximize the sum of the cards they have collected.

(a) (3') Tom decides to use a greedy strategy: "On my turn, I will take the larger of the two cards available to me." Show a small counterexample ($n \le 5$) that Tom will lose if he plays this greedy strategy. Assuming Tom goes first and Jerry plays optimally. Tom could have won if he also played optimally.

Solution: One possible arrangement is $[2, 2, 9, 3]$. Tom first greedily takes the 3 from the right end, and then Jerry snatches the 9, so Jerry gets 11 and Tom a miserly 5. If Tom had started by craftily taking the 2 from the left end, he'd guaranteed that he would get 11 and poor Jerry would be stuck with 5.

(b) (7') Jerry decides to use dynamic programming to find an algorithm to maximize his score, assuming he is playing against Tom and Tom is using the greedy strategy from part (a). Help Jerry to develop the dynamic programming solution. The solution you proposed should not worse than $O(n^2)$ in time complexity.

Solution:

Main Idea:

Let $A[1..n]$ denote the n cards in the line. Jerry defines $v(i, j)$ to be the highest score he can achieve if it's his turn and the line contains cards $A[i..j]$. Jerry can simplify your expression by expressing $v(i, j)$ as a function of $l(i, j)$ and $r(i, j)$, where $l(i, j)$ is defined as the highest score Jerry can achieve if it's his turn and the line contains cards $A[i..j]$, if he takes $A[i]$; also $r(i, j)$ is defined to be the highest score Jerry can achieve if it's his turn and the line contains cards $A[i..j]$, if he takes $A[j]$.

Then the recursive formula should be:

$$v(i, j) = \max\{l(i, j), r(i, j)\}$$

where

$$l(i, j) = \begin{cases} A[i] + v(i + 1, j - 1) & \text{if } A[j] > A[i + 1] \\ A[i] + v(i + 2, j) & \text{otherwise} \end{cases}$$

$$r(i, j) = \begin{cases} A[j] + v(i + 1, j - 1) & \text{if } A[i] \ge A[j - 1] \\ A[j] + v(i, j - 2) & \text{otherwise} \end{cases}$$

Proof of Correctness:

Let's consider the condition of $l(i, j)$. If Jerry takes the left one, then Tom will use a greedy strategy to consider the card sequence $A[i + 1..j]$. If $A[j] > A[i + 1]$, Tom will take $A[j]$ and then Jerry will consider the sequence $A[i+1..j-1]$. If $A[j] \leq A[i+1]$, Tom will take $A[i+1]$ and then Jerry will consider the sequence $A[i+2..j]$. The case for $r(i,j)$ is the same.

Time Complexity:

There are $n(n+1)/2$ sub-problems and each one can be solved in $\Theta(1)$ time. So the time complexity is $\Theta(n^2)$

5. (10 points) Pairwise DNA Sequence Alignment

Given two DNA sequences: a query sequence $Q = \langle Q_1, \dots, Q_m \rangle$ of m nucleotides and a subject sequence $S = \langle S_1, \dots, S_n \rangle$ of n nucleotides. There are 4 types of nucleotides, namely Adenine (A), Cytosine (C), Guanine (G) and Thymine (T). We provide a scoring matrix for nucleotides at the same index of these two sequences:

	A	C	G	T
A	1	-5	-1	-5
C	-5	1	-5	-1
G	-1	-5	1	-5
T	-5	-1	-5	1

where $\text{Score}(X, X)$ on the diagonal represents the score of a successful match for nucleotide X at the same index of both sequences, and $\text{Score}(X, Y)$ indicates the penalty for mismatching nucleotide X by nucleotide Y.

For example, assume we have $m = n$ here and there are two sequences $Q = \langle TGGTG \rangle$ and $S = \langle ATCGT \rangle$. Then the alignment score for these two sequences is

$$\begin{aligned}
\text{Score}(Q, S) &= \sum_i^n \text{Score}(Q_i, S_i) \\
&= \text{Score}(T, A) + \text{Score}(G, T) + \text{Score}(G, C) + \text{Score}(T, G) + \text{Score}(G, T) \\
&= (-5) + (-5) + (-5) + (-5) + (-5) \\
&= -25
\end{aligned}$$

However, if $m \neq n$, we must insert several gaps '$-$' to make two sequences the same length. In order to align two sequences for higher scores, we can also insert an arbitrary number of gaps '$-$' to each sequence at an arbitrary index. However, adding one gap will result in a gap penalty $\text{Penalty}(X, -) = \text{Penalty}(-, Y) = -2$.

Assume after inserting 4 gaps, we obtain $Q' = \langle -T - GGTG \rangle$ and $S' = \langle ATCG - T- \rangle$. Then the recomputed alignment score is:

$$\begin{aligned}
\text{Score}(Q', S') &= \text{Penalty}(-, A) + \text{Score}(T, T) + \text{Penalty}(-, C) + \text{Score}(G, G) \\
&\quad + \text{Penalty}(G, -) + \text{Score}(T, T) + \text{Penalty}(G, -) \\
&= (-2) + 1 + (-2) + 1 + (-2) + 1 + (-2) \\
&= -5
\end{aligned}$$

Notice that Q' and S' should share the same length after inserting gaps.

Given the scoring matrix and gap penalty, please come up with a dynamic programming algorithm to **maximize the pairwise alignment score** of a pair of DNA sequences by inserting gaps into these two sequences.

(a) (2') Define your subproblem for this question.

Solution: $OPT(i,j)$ = the maximum score of aligning the first i nucleotides of Q and the first j nucleotides of S.

(b) (6') Give your Bellman equation to solve the subproblems. (Proof of correctness is not required)

Solution:

$$OPT(i,j) = \begin{cases} j \cdot \text{Penalty}(-, Y) & \text{if } i = 0 \\ i \cdot \text{Penalty}(X, -) & \text{if } j = 0 \\ \max \begin{cases} OPT(i-1, j-1) + \text{Score}(S_i, Q_j) \\ OPT(i-1, j) + \text{Penalty}(S_i, -) \\ OPT(i, j-1) + \text{Penalty}(-, Q_j) \end{cases} & \text{otherwise} \end{cases}$$

Explanation: (NOT Required)

- The 1st term in max : align $S_{1\ldots i}$ with $Q_{1\ldots j}$ and match or mismatch S_i with Q_j
- The 2nd term in max : align $S_{1\ldots i-1}$ with $Q_{1\ldots j}$ and insert a gap to Q to align S_i
- The 3rd term in max : align $S_{1\ldots i}$ with $Q_{1\ldots j-1}$ and insert a gap to S to align Q_j

(c) (1') What is the answer to this question in terms of the subproblems?

Solution: $OPT(m,n)$

(d) (1') What is the runtime complexity of your algorithm?

Solution: $\Theta(mn)$

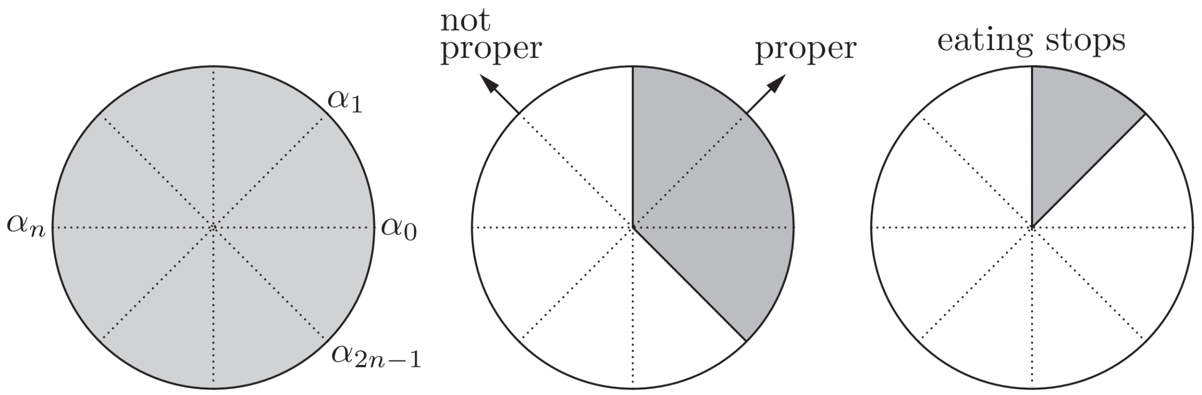
6. (4 points) Pizza Partitioning

Logan and his good friend Joel want to share a round pizza pie that has been cut into $2n$ equal sector slices along rays from the center at angles $\alpha_i = \frac{i\pi}{n}$ for $i \in \{0, 1, ..., 2n\}$, where $\alpha_0 = \alpha_{2n}$. Each slice i between angles α_i and α_{i+1} has a known integer tastiness t_i (which might be negative). To be "fair" to her friend, Logan decides to eat slices in the following way:

- They will each take turns choosing slices of pizza to eat: Logan starts as the chooser.
- If there is only one slice remaining, the chooser eats that slice, and eating stops.
- Otherwise the chooser does the following:
  - Angle α_i is proper if there is at least one uneaten slice on either side of the line passing through the center of the pizza at angle α_i.
  - The chooser picks any number $i \in \{1, ..., 2n\}$ where α_i is proper, and eats all uneaten slices counter-clockwise around the pizza from angle α_i to angle $\alpha_i + \pi$.

Logan wants to maximize the total tastiness of the slices she will eat. Describe an $O(n^3)$-time dynamic programming algorithm to find the maximum total tastiness Logan can guarantee herself via this selection process.

Hints: *As they eat pizza, the uneaten slices are always cyclically consecutive. That is, each time the chooser eats all uneaten slices within half side, There is no chance that two consecutively parts of pizza are left uneaten. You can choose consecutive cyclic subarrays as the subproblems.*



(a) (2') As a kickoff, let $v(i, j)$ be the tastiness of the j slices counter-clockwise from angle α_i. Write the formula over $v(i, j)$.

Solution:

$$v(i, j) = \sum_{k=0}^{j-1} t_{(i+k) \bmod 2n}$$

(b) (2') Define your subproblem for this question.

Solution: Define $OPT(i,j,p)$ be the maximum tastiness Logan can acheive with the j slices counter-clockwise from α_i remaining, when either: Logan is the chooser when $p=0$, or Joel is the chooser when $p=1$

(c) (4' (bonus)) Give your Bellman equation to solve the subproblems. (Proof of correctness is not required)

Solution:

$$OPT(i,j,0) = \begin{cases} t_i & \text{if } j=1 \\ \max\left\{ \max \left\{ \substack{v(i,j)+OPT(i+k,j-k,1) \\ v(i+k,j-k)+OPT(i,k,1)} \right\} \quad k \in \{1,...,j-1\} \right\} & \text{otherwise} \end{cases}$$

$$OPT(i,j,1) = \begin{cases} 0 & \text{if } j=1 \\ \max\left\{ \max \left\{ \substack{OPT(i+k,j-k,0) \\ OPT(i,k,0)} \right\} \quad k \in \{1,...,j-1\} \right\} & \text{otherwise} \end{cases}$$

(d) (2' (bonus)) What is the answer to this question in terms of the subproblems?

Solution:

$$\max_{i \in \{0,...,2n-1\}} \{OPT(i,n,1) + v((i+n) \bmod 2n, n)\}$$

(e) (2' (bonus)) Give an explanation and time complexity analysis of this algorithm.

Solution: As Logan tries to maximize the gain while Joel tries to minimize the gain, the chooser can choose any proper angle and then choose a side. So chooser can eat between k slices between α_i and α_{i+k} or $j-k$ slices between α_{i+k} and α_{i+j}.

In that case, Logan first chooses which has to eat. Max is the sum of tastiness on a half and tastiness achieved letting Joel choose on the other half. That is:

$$\max_{i \in \{0,...,2n-1\}} \{OPT(i,n,1) + v((i+n) \bmod 2n, n)\}$$

For time complexity, compute all $v(i,j)$ costs $O(n^3)$ time, and totally we have $\Theta(n^2)$ subproblems. For each subproblem, computation costs $\Theta(n)$ times since we need to enumerate k through $1-j$. Hence the total time complexity would be $\Theta(n^3)$.