CS101 Algorithms and Data Structures
Fall 2023
Homework 11

Due date: 23:59, January 7th, 2024

1. Please write your solutions in English.
2. Submit your solutions to Gradescope.
3. If you want to submit a handwritten version, scan it clearly.
4. When submitting, match your solutions to the problems correctly.
5. No late submission will be accepted.
6. Violations to any of the above may result in zero credits.
7. You are recommended to finish the algorithm design part of this homework with LaTeX.

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

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

Notes: (MUST READ!)

1. Problems in this homework require you to design **dynamic programming** algorithms. When grading these problems, we will put more emphasis on how you define your subproblems, whether your Bellman equation is correct and correctness of your complexity analysis.
2. **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.
3. Your **Bellman equation** should be a recurrence relation whose **base case** is well-defined. You can breifly **explain each term in the equation** if necessary, which might improve the readability of your solution and help TAs grade it.
4. Analyse the **runtime complexity** of your algorithm in terms of $\Theta(\cdot)$ notation.
5. 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.

(a) (0') Define the subproblems: $OPT(i)$ = the maximum sum of subarrays of A ending with A_i. Give your Bellman equation to solve the subproblems.

Solution:

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

Explanation: (NOT Required)

- 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}

(b) (0') What is the answer to this question in terms of OPT?

Solution:

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

(c) (0') What is the runtime complexity of your algorithm? (answer in $\Theta(\cdot)$)

Solution: $\Theta(n)$

2. (8 points) Multiple Choices

Each question has **one or more** correct answer(s). Select all the correct answer(s). For each question, you will get 0 points if you select one or more wrong answers, but you will get 1 point if you select a non-empty subset of the correct answers.

Write your answers in the following table.

(a)	(b)	(c)	(d)
AD	C	ABD	BCD

(a) (2') Which of the followings statements about Floyd-Warshall algorithm is/are true?

- **A. The Floyd-Warshall algorithm has a time complexity of $\Theta(|V|^3)$.**
- B. For two graphs $G_1 = (V_1, E_1), G_2 = (V_2, E_2)$ such that $|V_1| = |V_2|$ but $|E_1| > |E_2|$, the Floyd-Warshall algorithm costs more time on G_1 than G_2.
- C. Floyd-Warshall algorithm can't solve single-source shortest path problem, while Dijkstra can.
- **D. We can modify Floyd-Warshall algorithm to detect whether two vertices are strongly connected in a directed graph.**

Solution:

B. The run time is not affected by the number of edges.

C. It solves single-source shortest path problems for all sources.

D. u, v are strongly connected if and only if $d_{u,v} \neq \infty$ and $d_{v,u} \neq \infty$.

(b) (2') (**Single Choice**) Consider a weighted directed graph with n vertices $v_1, v_2, \ldots, v_n$, where n is even. Let $d_{i,j}^{(k)}$ be the shortest distance of v_i, v_j only allowing intermediate visits to $v_1, \ldots, v_k$ in the Floyd-Warshall algorithm. After running at least x iterations of the outermost loop k, it is ensured that the shortest path between $v_{n/2}$ and v_n is found in the matrix $d^{(x)}$ (in other words, $d_{n/2,n}^{(x)} = d_{n/2,n}^{(n)}$). Then $x =$

- A. $\frac{n}{2} - 1$
- B. $\frac{n}{2}$
- **C.** $n - 1$
- D. n

Solution: To ensure the shortest path between $v_{n/2}$ and v_n is found, we must consider the intermediate visits of every other vertices.

(c) (2') Which of the following statements about problems related to dynamic programming (DP) is/are true?

- **A. The greedy algorithm that solves the interval scheduling problem fails on the weighted interval scheduling problem.**
- **B. If there are n houses and m colors, the house coloring problem can be solved in $\Theta(nm)$ time complexity.**

C. Given an $n \times n$ matrix, maximum rectangle problem can be solved by calling Kadane's algorithm (solving maximum subarray problem) $\Theta(n)$ times.

D. The segmented least squares algorithm can be optimized to $\Theta(n^2)$ run time if we precompute the SSE e_{ij} for all $i \leq j$ in $\Theta(n^2)$ time.

Solution:

B. This choice is controversial. You can get full points if you choose ABD or AD.

Define the subproblems: $OPT(i, j)$ = minimal cost to paint the first i houses and the i-th house is painted with the j-th color.

$$OPT(i, j) = cost(i, j) + \min_{k \neq j}\{OPT(i-1, k)\}$$

The naive implementation is actually $\Theta(nm^2)$, because there are $\Theta(nm)$ subproblems and each subproblem needs to be computed in $\Theta(m)$ time.

However, it can be optimized to $\Theta(nm)$. We can define auxiliary subproblems F and G:

- $F(i, j) = \min_{k<j}\{OPT(i, k)\} = \min\{F(i, j-1), OPT(i, j-1)\}$.
- $G(i, j) = \min_{k>j}\{OPT(i, k)\} = \min\{F(i, j+1), OPT(i, j+1)\}$.

Then $OPT(i, j) = cost(i, j) + \min\{F(i-1, j), G(i-1, j)\}$. So $F(i, j), G(i, j), OPT(i, j)$ can all be computed in $\Theta(1)$ time.

C. $\Theta(n^2)$ times.

(d) (2') Which of the following statements about different solutions of the weighted interval scheduling is/are true? There are n jobs. Define $p(j)$ = largest index $i < j$ such that job i is compatible with j, and define $OPT(j)$ = max weight of any subset of mutually compatible jobs for subproblem consisting only of jobs $1, 2, \ldots, j$.

A. If we use brute-force algorithm to compute $OPT(n)$, then the worst-case time complexity is $\Theta\left(\left(\frac{1+\sqrt{5}}{2}\right)^n\right)$.

B. If we use top-down dynamic programming (memorization) to compute $OPT(n)$, then we consume additional $O(n)$ function call stack space compared to bottom-up dynamic programming.

C. No matter whether we use top-down dynamic programming (memorization) or bottom-up dynamic programming to compute $OPT(n)$, the time complexity is $\Theta(n)$, excluding the time for sorting and computing $p(j)$.

D. After $OPT(j)$ for all $j \in [0, n]$ are computed, we can find one possible set of mutually compatible jobs that maximize the sum of weights in $O(n)$ time.

Solution:

A. $\Theta\left(2^n\right)$, because $T(n) = T(n-1) + T(n-p(n)) + \Theta(1)$, and in the worst case $T(n) = 2T(n-1) + \Theta(1)$

$T(n) + c \sim 2T(n-1) + 2c \sim 2^n(T(0) + c) = \Theta(2^n)$.

3. (6 points) Floyd Warshall Algorithm

(a) (6') Consider the following implementation of the Floyd-Warshall algorithm. Assume $w_{ij} = \infty$ where there is no edge between vertex i and vertex j, and assume $w_{ii} = 0$ for every vertex i. Add some codes in the blank lines to detect whether there is negative cycles in the graph. (You may not use all blank lines.)

```
bool detectNegCycle(int graph[][V]) {
    int dist[V][V], i, j, k;

    for (i = 0; i < V; i++)
        for (j = 0; j < V; j++)
            dist[i][j] = graph[i][j];

    for (k = 0; k < V; k++) {
        for (i = 0; i < V; i++) {
            for (j = 0; j < V; j++) {
                if (dist[i][k] + dist[k][j] < dist[i][j])
                        dist[i][j] = dist[i][k] + dist[k][j];
            }
        }
    }

    -------------------------------------
    -------------------------------------
    -------------------------------------
    -------------------------------------
    -------------------------------------

    return false;
}
```

Solution:

```
bool detectNegCycle(int graph[][V]) {
    int dist[V][V], i, j, k;

    for (i = 0; i < V; i++)
        for (j = 0; j < V; j++)
            dist[i][j] = graph[i][j];

    for (k = 0; k < V; k++) {
        for (i = 0; i < V; i++) {
            for (j = 0; j < V; j++) {
                if (dist[i][k] + dist[k][j] < dist[i][j])
                        dist[i][j] = dist[i][k] + dist[k][j];
            }
        }
```

```
    }

    for (int i = 0; i < V; i ++)_________
        if (dist [ i ][ i ] < 0)_________
            return true;_________________
    _____________________________________
    _____________________________________

    return false;
}
```

4. (7 points) Array Section: Maximal Power

Given a sequence of positive integers $A = \langle a_1, \cdots, a_n \rangle$, we want to divide it into several consecutive sections so that the sum of power of these sections is maximized.

The power of section $\langle a_i, \cdots, a_j \rangle$ is defined as $aX_{i,j}^2 + bX_{i,j} + c$, where

- a, b, c are given constants.
- $X_{i,j} = \sum_{k=i}^{j} a_k$ is the sum of values in the section.

Please design a **dynamic programming** algorithm that returns the maximal sum of power of the sections that you divided.

For example, if $a = -1, b = 10, c = -20$ and $A = \langle 2, 2, 3, 4 \rangle$, the maximal sum of power is 9, and the way to divide the sequence is: $\langle 2, 2 \rangle, \langle 3 \rangle, \langle 4 \rangle$.

(a) (3') Define the subproblems for $i \in [0, n]$: $OPT(i)$ = the maximal sum of power of if you only consider dividing the first i elements. Give your Bellman equation to solve the subproblems.

Solution:

$$OPT(i) = \begin{cases} 0 & \text{if } i = 0 \\ \max\limits_{j \in [1,i]} \{OPT(j-1) + aX_{j,i}^2 + bX_{j,i} + c\} & \text{otherwise} \end{cases}$$

Explanation: (NOT Required)

- The base case is $OPT(0) = 0$.
- For the i-th subproblem, if the last section is from j to i, the maximal sum of power before the last section is $OPT(j-1)$, and the power of this section is $aX_{j,i}^2 + bX_{j,i} + c$. Consider all the cases of j and select the optimal one.

(b) (1') What is the answer to this question in terms of OPT?

Solution: $OPT(n)$.

(c) (3') What is the runtime complexity of your algorithm? (answer in $\Theta(\cdot)$)
Hint: How will you compute $X_{j,i}$ in $\Theta(1)$ time? Please give your analysis of the preprocessing and computing complexity.

Solution: $\Theta(n^2)$ (You can't get points here if your design a worst-case $\omega(n^2)$ algorithm.)

We can preprocess the prefix sum $S_i = \sum_{i=1}^{n} a_i$ for $i \in [0, n]$. Compute it by $S_i = S_{i-1} + a_i$, which costs $\Theta(n)$ time totally.
And then compute $X_{j,i} = S_j - S_{i-1}$, which costs $\Theta(1)$ time.

5. (7 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).

(a) (5') Define the subproblems: $F(v)$ = fewest odd number of coins to make change for v and $G(v)$ = fewest even number of coins to make change for v. Give your Bellman equation to solve the subproblems.

Solution:

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

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

Explanation: (NOT Required)

- 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.

(b) (1') What is the answer to this question in terms of F, G?

Solution: $F(V)$.

(c) (1') What is the runtime complexity of your algorithm? (answer in $\Theta(\cdot)$)

Solution: $\Theta(nV)$ (You can't get points here if your design a worst-case $\omega(nV)$ algorithm.)

6. (6 points) Minimum Cost Refueling

You are planning to from city A to city B on a highway. The distance between A and B is d kilometers. The vehicle departs with f_0 units of fuel. Each unit of fuel makes the vehicle travel one kilometer.

There are n gas stations along the way. The i-th station is situated p_i kilometers away from city A. Note that $0 < p_1 < p_2 < \cdots < p_n < d$.

If the vehicle chooses to refuel at the i-th station, $f_i > 0$ units would be added to the fuel tank whose capacity is unlimited, which costs you \$$c_i$. You have a budget B, which means that the sum of costs on refueling is at most \$$B$.

Please design a **dynamic programming** algorithm that returns **the minimum cost of refueling** to make sure the vehicle reaches the destination if the vehicle can reach the target, or returns $\varnothing$ if your budget is not enough or the fuel is not enough to support you to reach city B.

(a) (3') Define the subproblems for $i \in [0, n], j \in [0, B]$: $OPT(i, j)$ = the maximum distance you can drive if you spend **at most** \$$j$ among the first i stations. Give your Bellman equation to solve the subproblems.

Solution:

$$OPT(i,j) = \begin{cases} f_0 & \text{if } i = 0 \\ \max \begin{Bmatrix} OPT(i-1,j-c_i)+f_i \\ OPT(i-1,j) \end{Bmatrix} & \text{if } i > 0, OPT(i-1, j-c_i) \geq p_i \\ OPT(i-1,j) & \text{otherwise} \end{cases}$$

Explanation: (NOT Required)

- The base case is $i = 0$ and the initial fuel is f_0.
- If there exists a way to refuel with no more than \$$(j - c_i)$ and the fuel is enough to reach the i-th station, $OPT(i, j)$ can be updated by $OPT(i - 1, j - c_i)$.
- Otherwise, $OPT(i, j) = OPT(i - 1, j)$.

(b) (2') What is the answer to this question in terms of OPT?

Solution: If $OPT(n, B) < d$, return $\varnothing$.
Otherwise, the answer is $\min\limits_{\substack{j\in[0,B] \\ OPT(n,j)\geq d}} j$.

(c) (1') What is the runtime complexity of your algorithm? (answer in $\Theta(\cdot)$)

Solution: $\Theta(nB)$ (You can't get points here if your design a worst-case $\omega(nB)$ algorithm.)