CS3230 Semester 2 2024/2025
Design and Analysis of Algorithms

Tutorial 07
Dynamic Programming
For Week 08

Document is last modified on: February 19, 2025

1 Lecture Review: Dynamic Programming

The key ideas to solve a problem with Dynamic Programming (DP) are as follows:

- Optimal substructure: Can we express the solution recursively?
  Break the original problem into its subproblems.
- Realizes that there are only a small (maybe polynomial) number of subproblems.
  The naive implementation of the recursive solution encounters many overlapping subproblems.
  The recursive algorithm may take exponential time (solving the same subproblem many times).

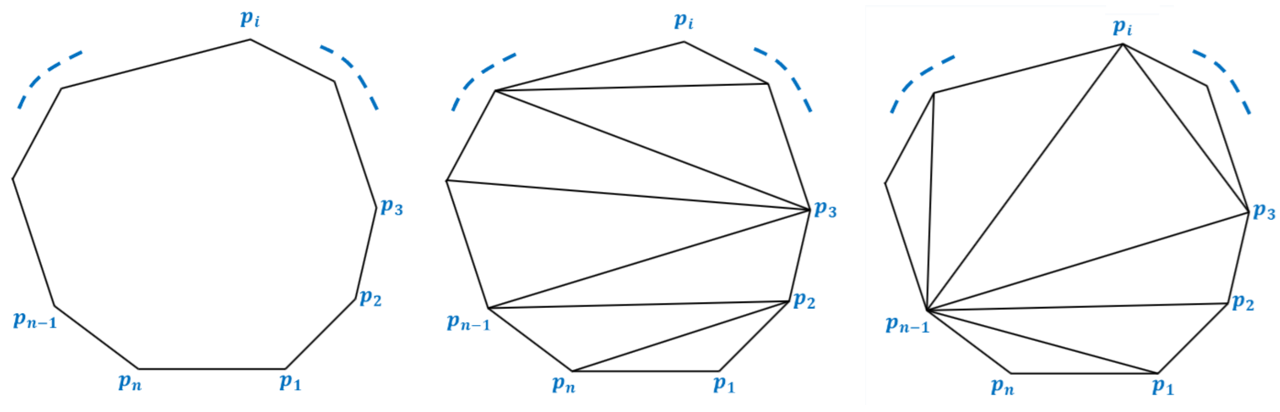
So we either:

- Top-down: Compute the recursive solution but memoize the solutions of the computed subproblems, so the next computation of the same subproblem can be done in $O(1)$.
- Bottom-up: Compute the recursive solution iteratively in a bottom-up fashion (also called tabulation), starting from the base cases and continue filling the next subproblems that we can compute next, gradually.

Both methods avoids wastage of computation and leads to an efficient implementation.

2 Tutorial 07 Questions

This tutorial is related to the **Convex Polygon Triangulation** problem: Given a convex polygon with n ($n \geq 2$) vertices (labeled with $1, 2, \ldots, n$), divide (or triangulate) the polygon into $n-2$ triangles. We can triangulate a convex polygon in many ways. The figure below shows 2 ways (middle and right pictures).



A triangle consisting of vertices (x, y, z) will have a weight of $W(x, y, z)$ – for the purpose of this problem, treat W as a black-box $O(1)$ function. Our objective is to minimize the sum of the weights of $n-2$ triangles in the optimal triangulation!

Q1). Let $TRI(x, y)$ be a function to triangulate a polygon with minimum weight sum, but we only consider the vertices in the range of $(x, x+1, x+2, \ldots, y)$. So our problem can be solved by calling $TRI(1, n)$. Your first task is to write a recursive formula of $TRI(x, y)$.

(a) Find the base case of $TRI(x, y)$

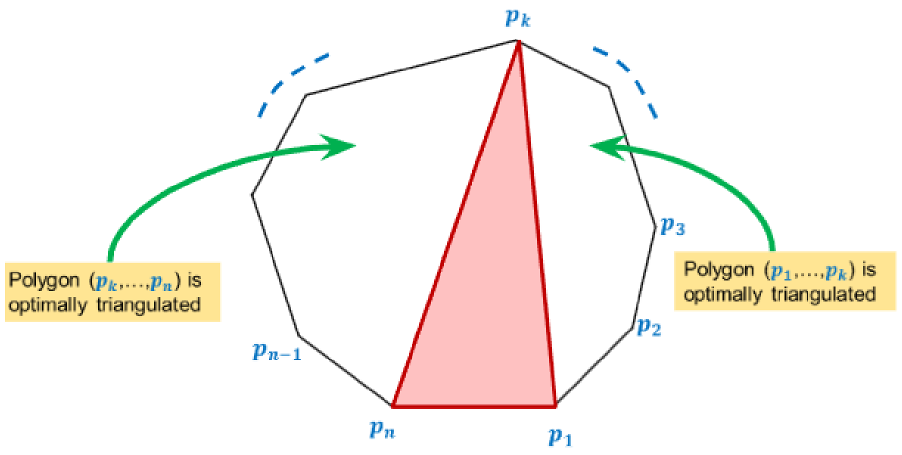
(b) Find the recursive case of $TRI(x, y)$

Hint: It calls $TRI(x', y')$ where $x < x'$ or $y' < y$.

$$TRI(x,y) = \begin{cases} 0, & \text{if } y - x = 1 \\ \min\limits_{k \in [x+1, y-1]} \left[TRI(x,k) + W(x,k,y) + TRI(k,y)\right], & \text{otherwise} \end{cases}$$

Base Case: Cannot triangulate a line (adjacent vertices x and y).
Recursive Case: Try all triangulations in any order in the recurrence:

- Subproblems $TRI(x, k)$ and $TRI(k, y)$
- Triangle (x, k, y) with weight $W(x, k, y)$



p_k
p_3
p_2
p_1
p_n
p_{n-1}
Polygon ($p_k,\ldots,p_n$) is optimally triangulated
Polygon ($p_1,\ldots,p_k$) is optimally triangulated

Q2). What is the time complexity of this recursive formula $TRI(1, n)$, if implemented verbatim.

(a) $O(n^2)$

too low

(b) $O(n^3)$

still too low

(c) $O(3^n)$

the answer; derivation below

Let $T(n)$ be the worst case running time of $TRI(1, n)$, so: Let $T(n)$ be the worst-case running time of $TRI(1, n)$.

$$T(2) = c, \quad \text{when } y - x = 1.$$

Expanding the recurrence for $T(n), T(n-1)$:

$$\begin{aligned} T(n) &= (T(2) + \boxed{\text{T(n-1)}} + c) + (T(3) + T(n-2) + c) \\ &\quad + \ldots + (T(n-2) + T(3) + c) + (\boxed{\text{T(n-1)}} + T(2) + \boxed{c}) \\ T(n-1) &= (T(2) + T(n-2) + c) + (T(3) + T(n-3) + c) \\ &\quad + \ldots + (T(n-2) + T(2) + c) \end{aligned}$$

Subtracting $T(n-1)$ from $T(n)$:

$$\begin{aligned} T(n) - T(n-1) &= 2T(n-1) + c \\ &\implies T(n) = 3T(n-1) + c \\ &\implies T(n) \approx 3^n \in O(3^n). \end{aligned}$$

Q3). Which one is the correct explanation regarding the findings from (Q2)?

(a) It has 3^n non-overlapping subproblems and each call runs in $\Theta(1)$

(b) It has n^2 non-overlapping subproblems and each call runs in $\Theta(\frac{3^n}{n^2})$

(c) It has n^2 subproblems but there are many overlaps

It has n^2 subproblems with significant overlap, making a Dynamic Programming solution necessary for efficiency.

Q4). Design a Dynamic Programming (DP) solution for **Convex Polygon Triangulation** problem.

(a) Using Top-Down DP

(b) Using Bottom-Up DP

(a) Using Top-Down DP

This is the easier route, as we just need to prepare a 2D *memo* table of size $n \times n$, set all cells to -1 or NULL or None (the number of distinct subproblems, and thus the space complexity, is $O(n^2)$).

In $TRI(x, y)$, if the subproblem/state (x, y) has been computed optimally, we return $memo[x][y]$.

Otherwise, we compute the optimal value for state (x, y) once, in $O(n)$, and store it in $memo[x][y]$, so future call of the same state (x, y) will be $O(1)$.

Notice that $TRI(x, y)$ requires the following combinations of $O(n)$ subproblems that will be handled recursively: $TRI(x, x+1)$ and $TRI(x+1, y)$, $TRI(x, x+2)$ and $TRI(x+2, y)$, $\ldots$, until $TRI(x, y-1)$ and $TRI(y-1, y)$. Visually, computing $TRI(x, y)$ requires all subproblems in that row of $TRI(x, ?)$ and all subproblems in that column of $TRI(?, y)$. PS: It is probably clearer to see this pattern using an animation (shown in tutorial).

Overall, there are $O(n^2)$ different subproblems and each sub-problem is only computed once in $O(n)$, so the total time complexity is $O(n^2 \times n) = O(n^3)$.

(b) Using Bottom-Up DP

This is the harder route, as we need to prepare the same 2D (DP, no longer called 'memo') table TRI of size $n \times n$ (space complexity is the same, $O(n^2)$), but we now need to figure out the 'correct filling order' (the topological order of the underlying recursion DAG) of this DP table. Here is one way:

Base case: For each $x \in [1..n-1]$, set $TRI[x][x+1] = 0$ (easy to figure out). And notice that this is ONE index away from the anti/counter diagonal of the $n \times n$ 2D DP table.

Recursive case: In bottom up, is to fill the table anti-diagonally. This is because each $TRI(x, y)$ requires all subproblems in that row of $TRI(x, ?)$ and all subproblems in that column of $TRI(?, y)$ to already been computed optimally before, and the only way this happens is if we process the 2D DP table anti-diagonally (not row-by-row left-to-right that we usually do for other DP problems involving 2D DP table):

```
for each $\Delta \in [2..n-1]$: // we try the other anti-diagonals, from 2 indices away and so on
    for each $x \in [1..n-1-\Delta]$: // we try all $x$
        $y = x + \Delta$ // this is the corresponding $y$
        $TRI[x][y] = \infty$
        for each $k \in [x+1..y-1]$: // we try all subproblems
            $TRI[x][y] = min(TRI[x][y], TRI[x][k] + w(x,k,y) + TRI[k][y])$
return $TRI[1][n]$
```

PS: It is probably clearer to see this pattern using an animation (shown in tutorial).

Clearly, the overall time complexity is $O(n^3)$ (from three nested loops), which is the same as Top-Down DP approach. While asymptotically equivalent, the Bottom-Up method can benefit from reduced recursion overhead.