NUS | Computing

National University of Singapore

CS3230

Computer Science

T06 – Week 7

Dynamic Programming

CS3230 – Design and Analysis of Algorithms

Lecture Review

Key Ideas in Dynamic Programming (DP)

- **Optimal substructure**: Solve recursively by breaking into subproblems.
- **Few unique subproblems**: Avoid redundant recomputation.

Two Approaches:

- **Top-down (Memoization)**: Store computed results to reuse in $O(1)$.
- **Bottom-up (Tabulation)**: Solve iteratively from base cases.

Both methods improve efficiency by **avoiding redundant** work.

Convex Polygon Triangulation

Minimize the total weight of $n-2$ triangles in the optimal triangulation, considering:

- Given a convex polygon with $n \geq 2$ vertices labeled $1, 2, \ldots, n$
- Divide the polygon into $n-2$ triangles.
- A triangle (x, y, z) has weight $W(x, y, z)$ (an $O(1)$ black-box function).
- Multiple triangulations exist.

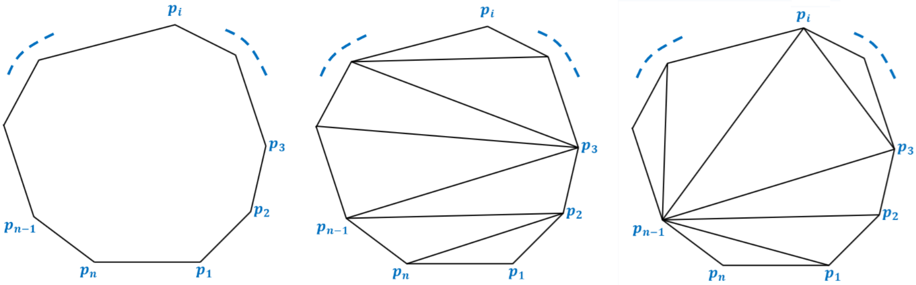

Figure 1: Two triangulation examples (middle, right).

Question 1

Let $TRI(x, y)$ be a function to triangulate a polygon with minimum weight sum (and returns this value), 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 $y' < y$.

Answer

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

a. **Base Case**: Cannot triangulate a line (adjacent vertices x and y).

b. **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 $W(x, k, y)$

Illustration

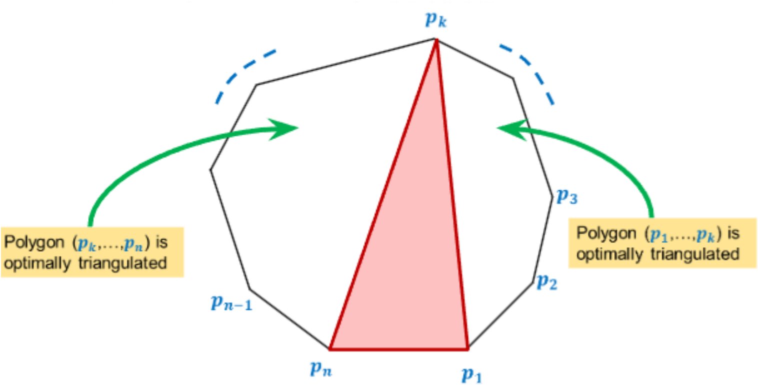

Figure 2: Optimal substructure

Question 2

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

a. $O(n^2)$

b. $O(n^3)$

c. $O(3^n)$

Answer

Let $T(n)$ be the worst-case running time of $TRI(1, n)$.
Note: $T(N)$ is also a recurrence of the time complexity of $TRI(1, n)$ whereas $TRI(1, n)$ itself is the DP recurrence to solve the problem.

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

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

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

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

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

Question 3

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 $\Theta\left(\frac{3^n}{n^2}\right)$.

c. It has n^2 subproblems, but there are many overlaps.

Question 3

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 $\Theta\left(\frac{3^n}{n^2}\right)$.

c. It has n^2 subproblems, but there are many overlaps.

Answer

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

Question 4

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

a. Using Top-Down DP

b. Using Bottom-Up DP

Answer

Using Top-Down DP

Use a 2D $memo$ table of size $n \times n$ ($O(n^2)$ space).

Algorithm

1. $TRI(x, y)$ is previously computed: return $memo[x][y]$
2. otherwise recursively solve $O(n)$ subproblems:
   a. Compute the $\min$ for $x < k < y : TRI(x, k) + W(x, k, y) + TRI(k, y)$
   b. Store it in $memo[x][y]$

Analysis

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

Illustration's Weights

Different weight functions can be used. Standard implementations typically define a triangle's weight as its perimeter, the sum of its side lengths. For illustration, we use LeetCode 1039 definition, i.e. $W(x, k, y) = values[x] * values[\cancel{y}] * values[y]$

k

Table 1: Truncated table of weights of LeetCode 1035 (example 3).

x	k	y	$W(x, k, y)$
0	1	2	$1 \cdot 3 \cdot 1 = 3$
0	1	3	$1 \cdot 3 \cdot 4 = 12$
0	1	4	$1 \cdot 3 \cdot 1 = 3$
0	1	5	$1 \cdot 3 \cdot 5 = 15$
0	2	3	$1 \cdot 1 \cdot 4 = 4$

Illustration

Show animation of the memoization table: [T06.q4a.gif].

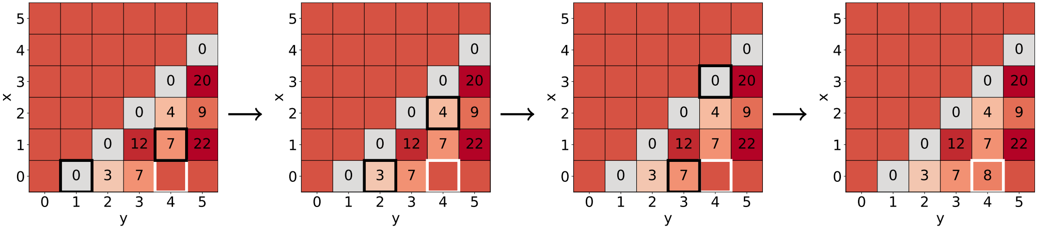

Figure 3: $memo$ table: $TRI(0,4)$ with its subproblems in row $TRI(0,?)$ and column $TRI(?,4)$. The final answer is at $TRI(0,5)$ - similar process

Using Bottom-Up DP

Use a 2D TRI DP table of size $n \times n$ ($O(n^2)$ space, same as $memo$), but now we must determine the **correct filling order** (**topological order** of the underlying recursion DAG).

Algorithm

1. Base Case: For each $x \in [1..n-1]$, set $TRI[x][x+1] = 0$. This is **one index away** from the anti-diagonal of the $n \times n$ DP table.
2. Recursive Case: Fill the table anti-diagonally, starting from **2 indices away** from the anti-diagonal. Each $TRI(x, y)$ needs to compute the min over previously computed values in its row and column, requiring a anti-diagonal filling order.

Analysis

Overall time complexity is $O(n^3)$,

- which is the same as Top-Down DP approach,
- Bottom-Up method can benefit from reduced recursion overhead.

Implementation

```
def compute_bottomup(n, w):
    TRI = [[ -1 ] * n for _ in range(n)]  # Initialize the DP table
    for x in range(n - 1):  # Base case, notice the 0-based indexing
        TRI[x][x + 1] = 0

    # Fill the table anti-diagonally
    for delta in range(2, n):  # Delta is the gap between x and y
        for x in range(n - delta):  # Iterate over all valid x
            y = x + delta
            t = float('inf')
            for k in range(x + 1, y):  # min over all x < k < y
                t = min(t, TRI[x][k] + w(x, k, y) + TRI[k][y])
            TRI[x][y] = t

    return TRI[0][n - 1]
```

Illustration

Show animation of the DP table: [T06.q4b.gif].

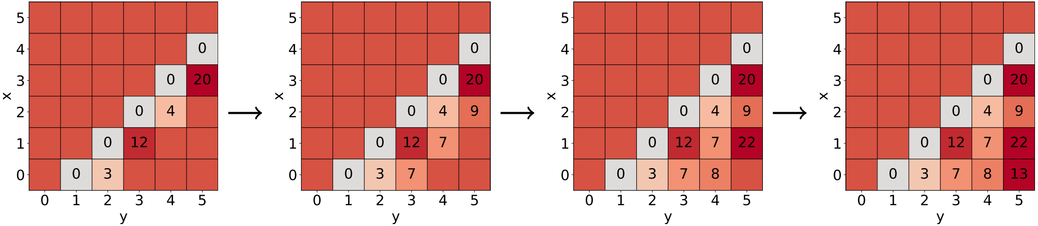

Figure 4: TRI DP table: Progressing through each anti-diagonal.

Question 5

As another practice, let's solve LeetCode 0279 - perfect-squares

Hint
Let S be the set of perfect squares under 10^4, there are at most $?$ such perfect squares.

Solution

Let S be the set of perfect squares under 10^4, there are at most 100 such perfect squares.

$$numSquares(n) = \begin{cases} -\infty, & \text{if } n < 0 \\ 0, & \text{if } n = 0 \\ \min_{x \in S} \left[1 + numSquares(n - x)\right]. & \text{otherwise} \end{cases}$$

a. **Base Cases**: 0 if $n = 0$, or $-\infty$ if $n < 0$

b. **Recursive Case**: Try all possible small perfect squares under 10^4

Time complexity is exponential if this recurrence is run verbatim, but only $O(n)$ if the states are not recomputed (either via top-down with memoization, or via bottom-up)

Question 6 (Optional)

Discuss the following LeetCode task:

1. Wednesday class: uncrossed-lines (isn't that just LCS?)
2. Thursday class: combination-sum-iv (somewhat like 0/1-Knapsack, actually SUBSET-SUM, counting version)
3. Friday class: minimum-score-triangulation-of-polygon (THIS TUT06!!)