CS3230 Tutorial 6

Dynamic Programming

(AY 25/26 Semester 2)

March 2, 2026

(Prepared by Benson and Hua Jun)

Contents

Recap: Optimal Substructure

Formal Understanding:

Optimal Substructure:
An optimal solution to a problem (instance) contains optimal solutions to subproblems.

+

Overlapping Subproblems:
A recursive solution contains a "small" number of distinct subproblems repeated many times.

Recap: Optimal Substructure

The Change-Making Problem

Consider a coin system with denominations $c_1 < c_2 < \cdots < c_m$.
What is the minimum number of coins needed to pay $\$d$ **without change**?

- Example: What is the minimum number of coins needed to pay \$12, using \$1, \$3 and \$5 coins?
- Assume we have already decided for sure to take a \$3 coin.

Minimum number of coins to pay \$9

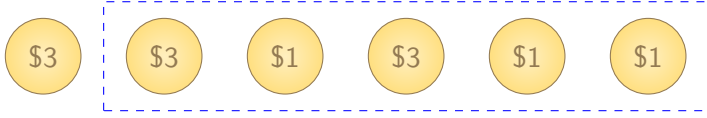
Optimal Substructure: An optimal solution to a problem (instance) contains optimal solutions to subproblems.

Recap: Optimal Substructure

Optimal Substructure: An optimal solution to a problem (instance) contains optimal solutions to subproblems.

Proof of Optimal Substructure Property (by contradiction):

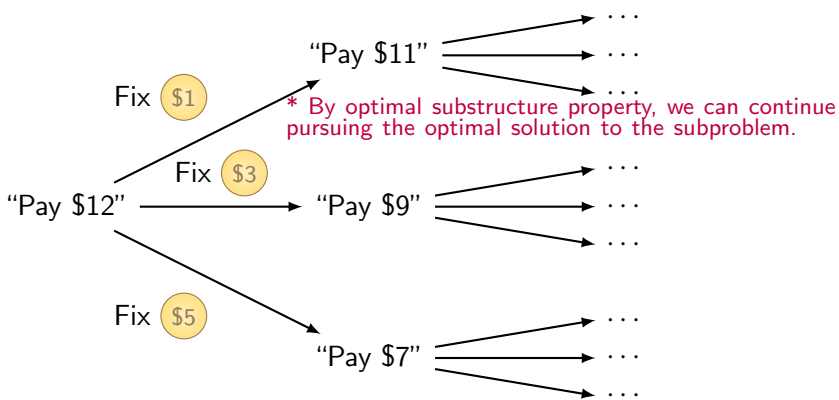
- Denote our solution by S where $\sum S = x$.
- Suppose the solution to a subproblem is suboptimal, i.e. there exists $T \subseteq S$ such that $\sum T = \sum T^* = y$, yet $|T^*| < |T|$.
- Replace the coins in T by the coins in T^*, i.e. consider $S' = (S \setminus T) \cup T^*$.
- Then, we have obtained a better solution, i.e. $\sum S' = x$ yet $|S'| < |S|$. Contradiction!



Recap: Optimal Substructure

Main Idea: Exhaust all possibilities on what to fix.

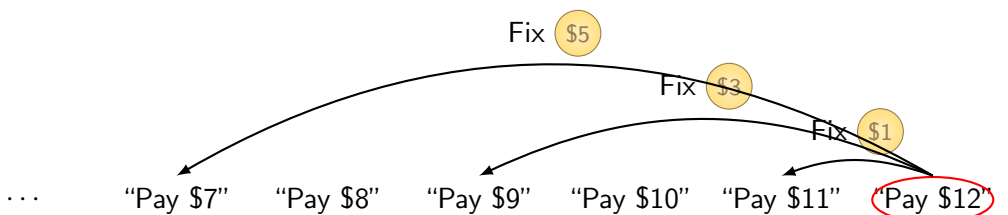
Top-down approach (start from the entire problem):



Recap: Optimal Substructure

Main Idea: Exhaust all possibilities on what to fix.

Bottom-up approach (start from subproblems with smaller "sizes"):



Recap: Dynamic Programming in Action

1 State → 2 Recurrence → 3 Base

State: A snapshot of the problem at a particular point during the computation.

- Fix part of the solution.
  - In what meaningful order do we fill in the solutions step by step?
- What do we want to calculate?
  - Let $dp(i)$ be the optimal solution for fixing items $1, 2, \ldots, i$.
  - Let $dp(i)$ be the optimal solution for fixing items $i, i+1, \ldots$ given we have already fixed items $1, 2, \ldots, i-1$. (For decision problems: 1 if possible, 0 if impossible.)
- What states are needed when we **transition** from a state to another?
  - Sometimes, we need to maintain extra quantities to calculate the contribution of a decision to the final answer (e.g. the most recent decision made).
- What states are needed to verify if we satisfied the **constraints** of the problem?
  - E.g. Knapsack problem: Maintain the total weight of items chosen.

Recap: Dynamic Programming in Action

Recurrence: How you would calculate the value of a state from its subproblems.

- Consider fixing a new part of the solution.
  - What are the decisions available?
  - What is the extra cost incurred / profit gained? (for optimization problems)
  - How does the state change after adding/removing the part?
  - What are the constraints so that you can fix this new part of the solution?

Recap: Dynamic Programming in Action

Base case: States whose values are **trivial to calculate**.

- What is the case that is trivial to calculate?
  - Let $dp(i)$ be the optimal solution for fixing items $1, 2, \ldots, i$.
    $\Rightarrow$ When $i = 0$, no items need to be fixed.
  - Let $dp(i)$ be the optimal solution for fixing items $i, i+1, \ldots$ given we have already fixed items $1, 2, \ldots, i-1$.
    $\Rightarrow$ When $i = n+1$ (n = number of items), no items need to be fixed.
- What is the value? (Usually 0 / $-\infty$ / ∞, but it can differ sometimes.)

Recap: Dynamic Programming in Action

And here is your DP solution! Remember that you are also expected to present:

- Your algorithm used to compute the *dp* function.
  - Bottom-up with a specified computation order.
  - Top-down with memoization.
- A time complexity analysis. Time complexity = Number of distinct states $\times$ Time needed to compute each state.

Recap: Dynamic Programming in Action

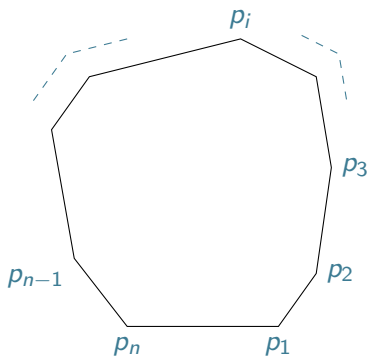
	Top-Down (Memoization)	**Bottom-Up (Tabulation)**
Formulation	Intuitive (recursive thinking)	Needs practice
States	States will be determined on the fly	Need to be clear what states are needed
Computation	Recursive	**Order is important**
Speed	Good when many states are unused **Overhead in memoization table**	Good when most states are used Time wastage for **unnecessary states**
Implementation	Longer	Shorter

Sharpen your bottom-up DP skills by practicing!

- Atcoder Educational DP Contest Problems A-P (Solutions)
- DP Problems on Leetcode

Optimal Triangulation – Introduction

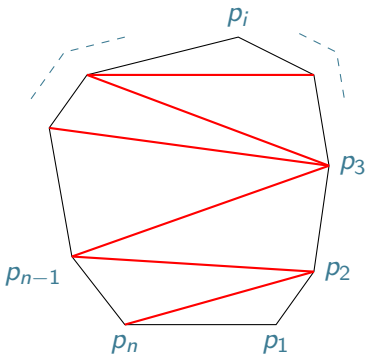
Given a convex polygon with n vertices. Let's denote them as $p_1, p_2, \ldots, p_n$.
We want to divide (or triangulate) the polygon into $n-2$ triangles.

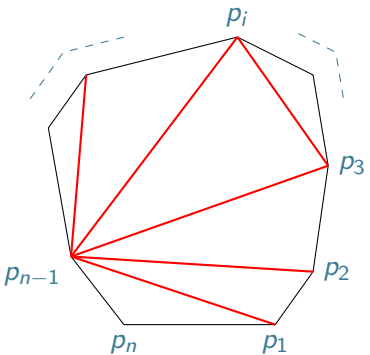


Polygon Triangulation: Drawing $n-3$ diagonals to divide the polygon into $n-2$ triangles.

Optimal Triangulation – Introduction

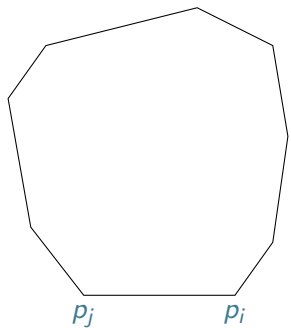
Let us denote the weight of a triangle with its vertices being p_i, p_j, p_k as $\omega(i, j, k)$. The cost of a triangulation is the sum of the weights of all triangles in the triangulation. **Minimize this sum!**





Q1. Recursive Formulation

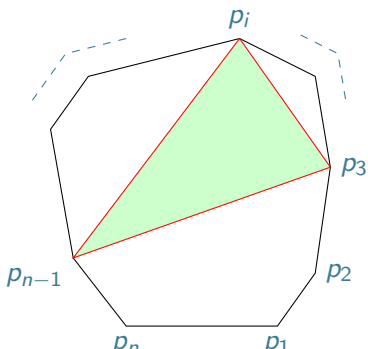
Let $\tau(i, j)$ be the cost of an optimal triangulation of the polygon with vertices $p_i, \ldots, p_j$.
Write down a recursive formula for the above problem, i.e. express $\tau(i, j)$ in terms of $\tau(i', j')$'s where $j' - i' < j - i$ and $j' \leq j$, $i' \geq i$.



Q1. Recursive Formulation

How can we make a recusive formulation?

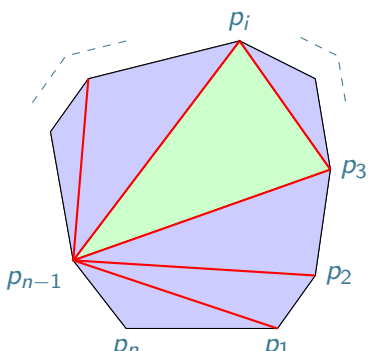
- **Important DP Step:** Let's try to fix one part of the solution and see if we can reduce it to smaller problems.
- Let's try to fix one of the triangles?



Q1. Recursive Formulation

Can this be a recusive formulation?

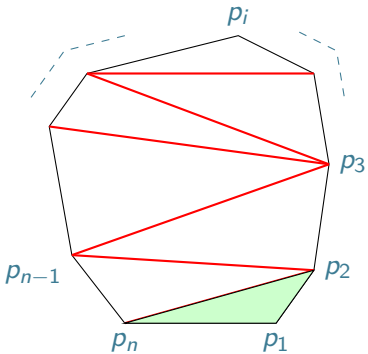
- The triangle splits the polygon into three smaller polygons, these are optimal substructures.
- Optimal substructure property: If our solution is optimal for the large polygon $(p_1, \ldots p_n)$, then the sub-solution is optimal for the substructures as well!

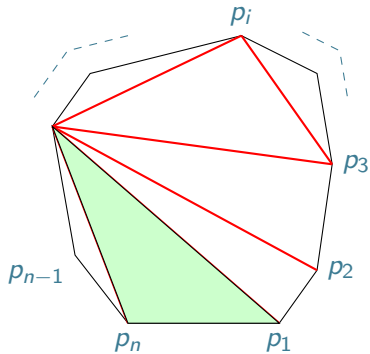


Q1. Recursive Formulation

Can this be a recursive formulation?

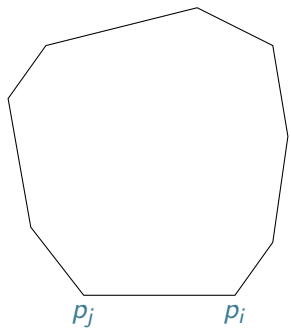
- Possible idea, can actually lead to viable solution, but note that trying all possible triangles would take $O(N^3)$.
- Is there a way we can reduce the number of points we need to try?
- Note that for every edge, it will be an edge for one of the triangles, thus we can pick a random edge and try to find what is the third vertex.





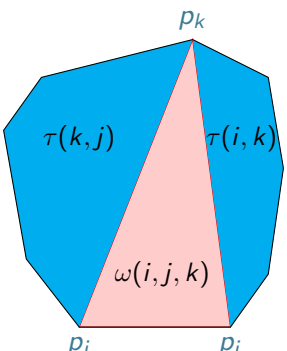
Q1. Recursive Formulation

Let $\tau(i, j)$ be the cost of an optimal triangulation of the polygon with vertices $p_i, \ldots, p_j$.
Write down a recursive formula for the above problem, i.e. express $\tau(i, j)$ in terms of $\tau(i', j')$'s where $j' - i' < j - i$ and $j' \leq j$, $i' \geq i$.



Q1. Recursive Formulation

Let $\tau(i,j)$ be the cost of an optimal triangulation of the polygon with vertices $p_i, \ldots, p_j$.
Write down a recursive formula for the above problem, i.e. express $\tau(i,j)$ in terms of $\tau(i',j')$'s where $j'-i' < j-i$ and $j' \leq j$, $i' \geq i$.



- Suppose the optimal triangulation has a triangle with vertices p_i, p_j, p_k.
- Then the polygons $p_i, \ldots, p_k$ and $p_k, \ldots, p_j$ are optimally triangulated. (optimal substructure)
- Therefore, $\tau(i,j) = \omega(i,j,k) + \tau(i,k) + \tau(k,j)$.

$$\therefore \tau(i,j) = \begin{cases} 0 & \text{if } j \leq i+1 \\ \min\limits_{i<k<j} \omega(i,j,k) + \tau(i,k) + \tau(k,j) & \text{otherwise} \end{cases}$$

Q2. Our First Attempt

Consider the following algorithm to find the value of $\tau(i, j)$:

Algorithm Our first attempt

```
1:  procedure FIND-τ(i, j)
2:      if j ≤ i + 1 then
3:          return 0
4:      else
5:          t ← ∞
6:          for k ← i + 1 to j − 1 do
7:              temp ← ω(i, j, k)+ FIND-τ(i, k) + FIND-τ(k, j)
8:              if t > temp then
9:                  t ← temp
10:         return t
```

Consider FIND-$\tau(1, n)$.
What is the running time?

A. $\Theta(2^n)$
B. $\Theta(3^n)$
C. $\Theta(n^2)$
D. $\Theta(n^3)$

Q2. Our First Attempt

```
1: for k ← i + 1 to j − 1 do
2:     temp ← ω(i, j, k)+ FIND-τ(i, k) + FIND-τ(k, j)
3:     if t > temp then
4:         t ← temp
```

$$T(n) = (T(2) + T(n-1) + c) + (T(3) + T(n-2) + c) + \cdots + (T(n-1) + T(2) + c)$$
$$= 2\sum_{i=2}^{n-1} T(i) + c(n-2)$$

$\Rightarrow T(n) - T(n-1) = 2T(n-1) + c$
$\Rightarrow T(n) = 3T(n-1) + c$
By substitution method, $T(n) = \Theta(3^n)$.

Q3. How many sub-problems?

Consider the previous FIND-$\tau(1, n)$ algorithm. Which one of the following is/are true?

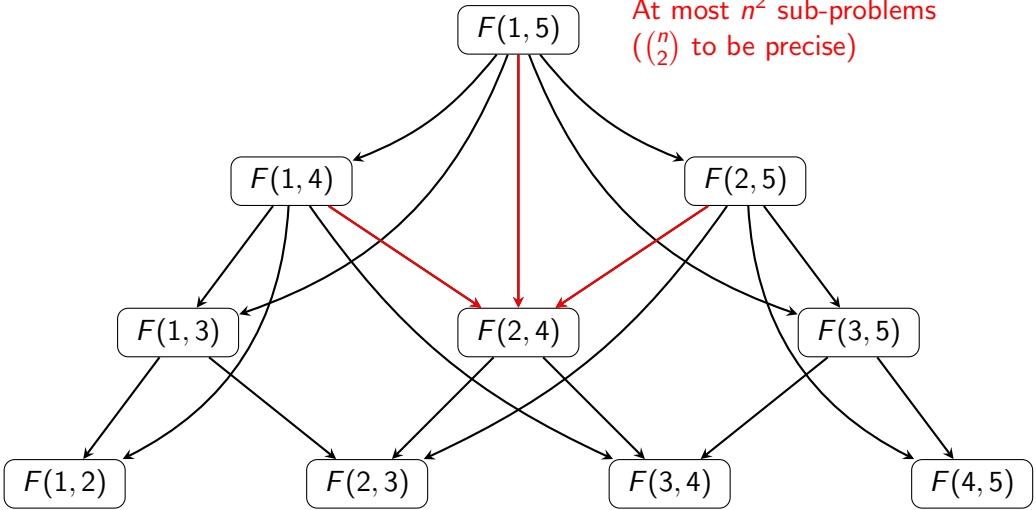
A. FIND-$\tau(1, n)$ computes $\Omega(3^n)$ different sub-problems.

B. FIND-$\tau(1, n)$ computes only at most n^2 different sub-problems, but to compute each sub-problem (non-recursively) it takes $\Omega\left(\frac{3^n}{n^2}\right)$ time.

C. FIND-$\tau(1, n)$ computes only at most n^2 different sub-problems, but each sub-problem multiple times.

Answer. C. There are at most n^2 combinations for (i, j) in FIND-$\tau(i, j)$. This is an instance of **overlapping subproblems**.

Q3. How many sub-problems?



Q4a. Top-down DP in Action

Design an $O(n^3)$ time dynamic programming algorithm to compute $\tau(1, n)$.

Algorithm Our first attempt

```
1:  procedure FIND-τ(i, j)
2:      if j ≤ i + 1 then
3:          return 0
4:      else
5:          t ← ∞
6:          for k ← i + 1 to j − 1 do
7:              temp ← ω(i, j, k)+ FIND-τ(i, k) + FIND-τ(k, j)
8:              if t > temp then
9:                  t ← temp
10:         return t
```

Q4a. Top-down DP in Action

Solution: **Memoize** $\tau(i, j)$ values that have been computed.

Algorithm $O(n^3)$ Top-down DP

```
1:  procedure FIND-τ(i, j)
2:      if j ≤ i + 1 then
3:          return 0
4:      else if Memo[i][j] ≠ ∞ then          ▷ Assume: Memo[i][j] is initialized to ∞
5:          return Memo[i][j]
6:      else
7:          for k ← i + 1 to j − 1 do
8:              temp ← ω(i, j, k) + FIND-τ(i, k) + FIND-τ(k, j)
9:              if Memo[i][j] > temp then
10:                 Memo[i][j] ← temp
11:         return Memo[i][j]
```

Q4a. Top-down DP in Action

Solution: **Memoize** $\tau(i,j)$ values that have been computed.

Algorithm $O(n^3)$ Top-down DP

```
1:  procedure FIND-τ(i, j)
2:      if j ≤ i + 1 then
3:          return 0
4:      else if Memo[i][j] ≠ ∞ then
5:      else
6:          for k ← i + 1 to j − 1 do
7:              temp ← ω(i, j, k) + FIND-τ(i, k) +
                FIND-τ(k, j)
8:              if Memo[i][j] > temp then
9:                  Memo[i][j] ← temp
10:         return Memo[i][j]
```

- Unique number of substructures: $O(N^2)$.
- Time to compute a substructure: $O(N)$.
- Total time complexity: $O(N^3)$.
- Total space complexity: $O(N^2)$.

Q4b. Bottom-up DP in Action

Complete the following bottom-up dynamic programming implementation.

Algorithm $O(n^3)$ Bottom-up DP to find $\tau(1, n)$

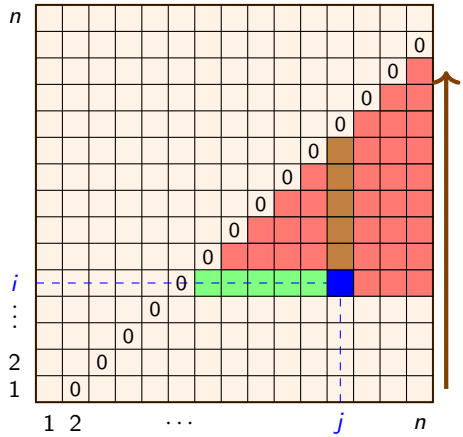
```
1:  procedure FIND-τ(1, n)
2:      for i ← 1 to n − 1 do
3:          T[i][i + 1] ← 0
4:
5:
6:
7:
8:          for k ← i + 1 to j − 1 do
9:
10:
11:
12:     return T[1][n]
```

Q4b. Bottom-up DP in Action

First Try:

Algorithm $O(n^3)$ Bottom-up DP to find $\tau(1, n)$

```
1:  procedure FIND-τ(1, n)
2:      for i ← 1 to n − 1 do
3:          T[i][i + 1] ← 0
4:      for i ← 1 to n do
5:          for j ← i + 2 to n do
6:              T[i][j] ← ∞
7:
8:              for k ← i + 1 to j − 1 do
9:                  temp ← T[i][k] + T[k][j] + ω(i, j, k)
10:                 if T[i][j] > temp then
11:                     T[i][j] ← temp
12:     return T[1][n]
```



Question 5: How would you fix this?

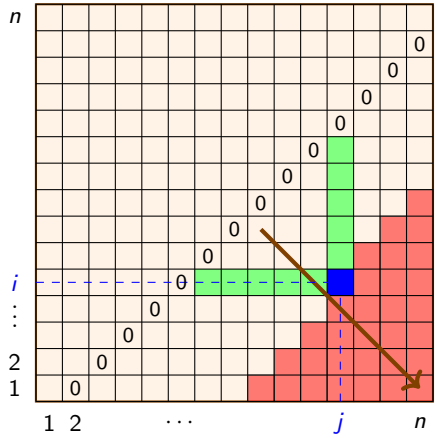
Q4b. Bottom-up DP in Action

Idea: We should compute the DP values in increasing $j - i + 1$.

- We should solve subproblems with less vertices in the polygon first.

Algorithm $O(n^3)$ Bottom-up DP to find $\tau(1, n)$

```
1:  procedure FIND-τ(1, n)
2:      for i ← 1 to n − 1 do
3:          T[i][i + 1] ← 0
4:      for Δ ← 2 to n − 1 do
5:          for i ← 1 to n − Δ do
6:              j ← i + Δ
7:              T[i][j] ← ∞
8:              for k ← i + 1 to j − 1 do
9:                  temp ← T[i][k] + T[k][j] + ω(i, j, k)
10:                 if T[i][j] > temp then
11:                     T[i][j] ← temp
12:     return T[1][n]
```



Q4b. Bottom-up DP in Action

Idea: We should compute the DP values in increasing $j - i + 1$.

- We should solve subproblems with less vertices in the polygon first.

Algorithm $O(n^3)$ Bottom-up DP to find $\tau(1, n)$

```
1:  procedure FIND-τ(1, n)
2:      for i ← 1 to n − 1 do
3:          T[i][i + 1] ← 0
4:      for Δ ← 2 to n − 1 do
5:          for i ← 1 to n − Δ do
6:              j ← i + Δ
7:              T[i][j] ← ∞
8:              for k ← i + 1 to j − 1 do
9:                  temp ← T[i][k] + T[k][j] + ω(i, j, k)
10:                 if T[i][j] > temp then
11:                     T[i][j] ← temp
12:     return T[1][n]
```

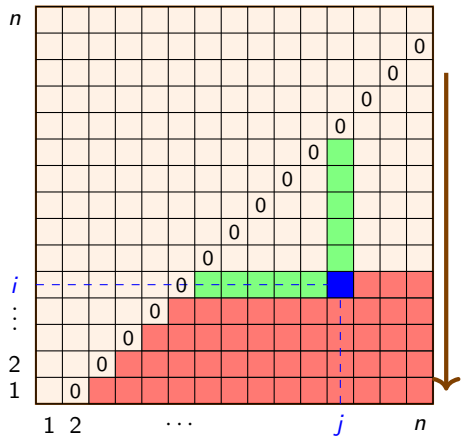
- Total time complexity: $O(N^3)$.
- Total space complexity: $O(N^2)$
- Asymptotically equivalent to Top-Down approach but can benefit from reduced recursion overhead in practice.

Q4b. Bottom-up DP in Action

Alternative solution:

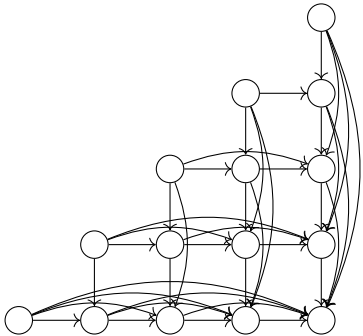
Algorithm $O(n^3)$ Bottom-up DP to find $\tau(1, n)$

```
1:  procedure FIND-τ(1, n)
2:      for i ← 1 to n − 1 do
3:          T[i][i + 1] ← 0
4:      for i ← n − 2 down to 1 do
5:          for j ← i + 2 to n do
6:              T[i][j] ← ∞
7:
8:              for k ← i + 1 to j − 1 do
9:                  temp ← T[i][k] + T[k][j] + ω(i, j, k)
10:                 if T[i][j] > temp then
11:                     T[i][j] ← temp
12:     return T[1][n]
```



Q4b. Bottom-up DP in Action

- Moral of the story: Computation order needs to be carefully decided for bottom-up DP – The subproblems need to be computed before the problem is computed.
- We can visualize these "dependencies" as a **Directed Acyclic Graph (DAG)**, and each valid computation order is a **topological ordering** of the graph.



Bonus: A recurrence relation

Extra Slide

Let $h(1) = 1$ and $h(n) = \sum_{i=2}^{n} h\left(\left\lfloor \frac{n}{i} \right\rfloor\right)$ for $n \geq 2$.

Show how you can calculate $h(n)$ in $o(n)$ time and prove rigorously it is indeed $o(n)$ (note the small o). You will need to obtain the bound by evaluating integrals.

Hint (copy to view): How many distinct values are there in $\left(\left\lfloor \frac{n}{1} \right\rfloor, \left\lfloor \frac{n}{2} \right\rfloor, \ldots, \left\lfloor \frac{n}{n} \right\rfloor\right)$?

Bonus: A recurrence relation

Extra Slide

Consider recursion with memoization. We claim that only $2\sqrt{n}$ states are visited, which are the distinct numbers $\left(\left\lfloor \frac{n}{1} \right\rfloor, \left\lfloor \frac{n}{2} \right\rfloor, \ldots, \left\lfloor \frac{n}{n} \right\rfloor\right)$.

- Part 1: To show that only the states $\left(\left\lfloor \frac{n}{1} \right\rfloor, \left\lfloor \frac{n}{2} \right\rfloor, \ldots, \left\lfloor \frac{n}{n} \right\rfloor\right)$ are visited.

  Idea: We first show that $\left\lfloor \frac{\left\lfloor \frac{n}{p} \right\rfloor}{q} \right\rfloor = \left\lfloor \frac{n}{pq} \right\rfloor$.

  - Proof: Using quotient-remainder theorem, write $n = ap + b$ and $a = cq + d$.
    Then $\left\lfloor \frac{n}{pq} \right\rfloor = \left\lfloor \frac{cpq + dp + b}{pq} \right\rfloor = c$ (since $dp + b \leq (q-1)p + (p-1) < pq$)
    and $\left\lfloor \frac{\left\lfloor \frac{n}{p} \right\rfloor}{q} \right\rfloor = \left\lfloor \frac{a}{q} \right\rfloor = c$.

  Then all recursive calls will be in the form of $h\left(\left\lfloor \frac{n}{i_1 i_2 \cdots i_k} \right\rfloor\right)$.

Bonus: A recurrence relation

Extra Slide

Consider recursion with memoization. We claim that only $2\sqrt{n}$ states are visited, which are the distinct numbers $\left(\left\lfloor \frac{n}{1} \right\rfloor, \left\lfloor \frac{n}{2} \right\rfloor, \ldots, \left\lfloor \frac{n}{n} \right\rfloor\right)$.

- Part 2: To show that $\left(\left\lfloor \frac{n}{1} \right\rfloor, \left\lfloor \frac{n}{2} \right\rfloor, \ldots, \left\lfloor \frac{n}{n} \right\rfloor\right)$ only contains $2\sqrt{n}$ distinct numbers.
  For $i \leq \sqrt{n}$, then $\left(\left\lfloor \frac{n}{1} \right\rfloor, \left\lfloor \frac{n}{2} \right\rfloor, \ldots, \left\lfloor \frac{n}{i} \right\rfloor\right)$ has at most $\sqrt{n}$ distinct numbers.
  For $i > \sqrt{n}$, we have $\left\lfloor \frac{n}{i} \right\rfloor \leq \frac{n}{i} < \frac{n}{\sqrt{n}} = \sqrt{n}$. Hence $\frac{n}{i}$ is always a positive integer between 1 and $\sqrt{n}$, for a total of at most $\sqrt{n}$ distinct numbers.

Bonus: A recurrence relation

Extra Slide

- Is the time complexity of the program $O(\sqrt{n})$? Not really!
- If we apply the recurrence formula naively, we have to sum up the $n-1$ subproblems and this takes $O(n)$ time.
- Then
$$T(n) = \left(1 + 2 + \ldots + (\sqrt{n} - 1)\right) + \left(\left\lfloor \frac{n}{1} \right\rfloor + \left\lfloor \frac{n}{2} \right\rfloor + \ldots + \left\lfloor \frac{n}{\sqrt{n}} \right\rfloor\right) = \Theta(n \log n).$$
- Not fast enough!

Bonus: A recurrence relation

Extra Slide

We can borrow the previous idea to handle $i \leq \sqrt{n}$ and $i > \sqrt{n}$ separately.

- For $i \leq \sqrt{n}$, we could call $h\left(\frac{n}{i}\right)$ naively for all i.
- For $i > \sqrt{n}$, we could call $h(1), h(2), \ldots, h(\lfloor \sqrt{n} \rfloor)$ directly and calculate the number of times each of them should be added to the sum in $O(1)$ time.

Then we can handle $h(n)$ with at most $\sqrt{n}$ overhead, instead of n!

Bonus: A recurrence relation

Extra Slide

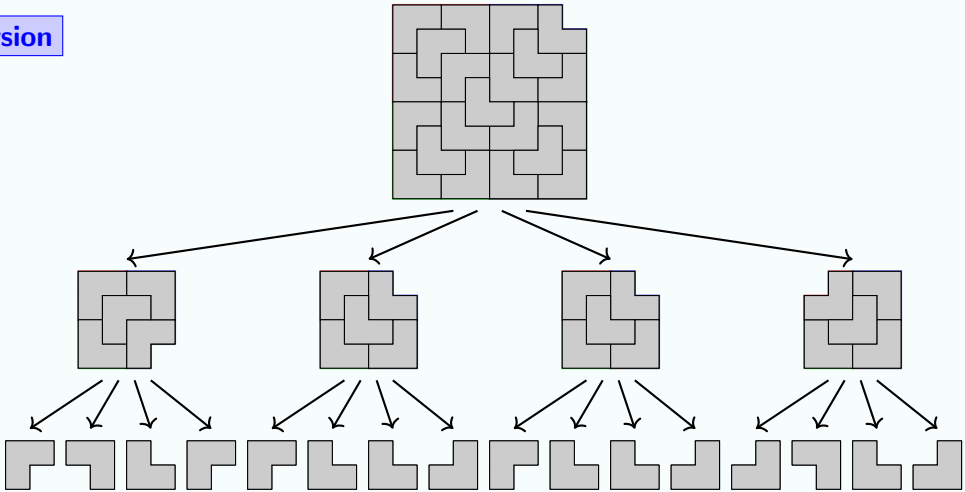
Now, let's analyze $T(n)$. (Note: A lower bound can be obtained similarly.)

$$
\begin{aligned}
T(n) &\leq \left(\sqrt{1}+\sqrt{2}+\ldots+\sqrt{\sqrt{n}}\right)+\left(\sqrt{\left\lfloor\frac{n}{1}\right\rfloor}+\sqrt{\left\lfloor\frac{n}{2}\right\rfloor}+\ldots+\sqrt{\left\lfloor\frac{n}{\sqrt{n}}\right\rfloor}\right) \\
&\leq \left(\sqrt{1}+\sqrt{2}+\ldots+\sqrt{\sqrt{n}}\right)+\left(\sqrt{\frac{n}{1}}+\sqrt{\frac{n}{2}}+\ldots+\sqrt{\frac{n}{\sqrt{n}}}\right) \\
&\leq \int_{x=1}^{\sqrt{n}+1}\sqrt{x}\,dx+\sqrt{\frac{n}{1}}+\int_{x=1}^{\sqrt{n}}\sqrt{\frac{n}{x}}\,dx \\
&= \left[\frac{2}{3}x^{3/2}\right]_1^{\sqrt{n}+1}+\sqrt{n}+\left[2\sqrt{n}\cdot\sqrt{x}\right]_1^{\sqrt{n}} \\
&= O(n^{3/4})+O(n^{1/2})+O(n^{3/4}) \\
&= O(n^{3/4})
\end{aligned}
$$

Appendix

Extra: Top-down vs Bottom-up

Recursion



Extra: Top-down vs Bottom-up

Let the board size be $n \times n$. What is the time complexity of this algorithm?

A. $\Theta(n^2)$

B. $\Theta(n^2 \log n)$

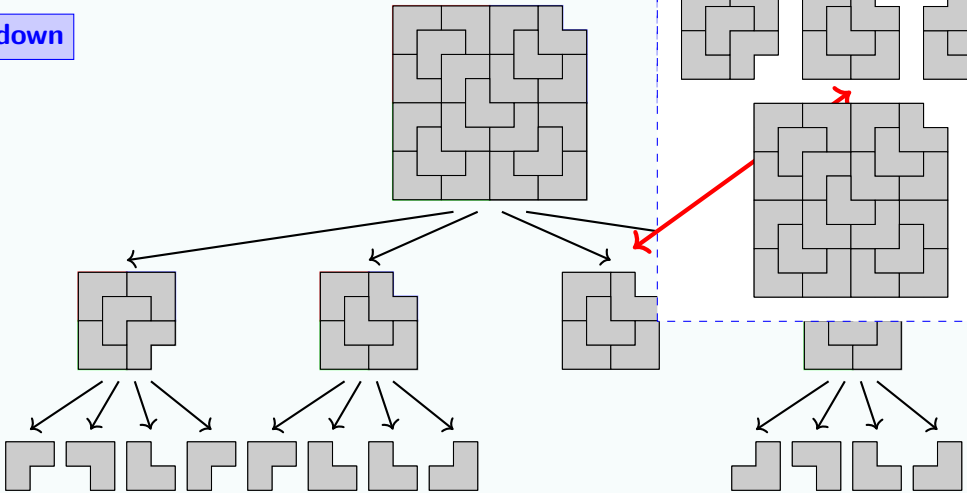
C. $\Theta(n^2 \log^2 n)$

D. $\Theta(n^{\log_2 3})$

Solution.
The recurrence is $T(n) = 4T(\frac{n}{2}) + cn^2$.
$f(n) = \Theta(n^2) = \Theta(n^{\log_2 4}) \Rightarrow$ Case 2 of master theorem applies.
$\therefore T(n) = \Theta(n^2 \log n)$.

Extra: Top-down vs Bottom-up

Top-down



Extra: Top-down vs Bottom-up

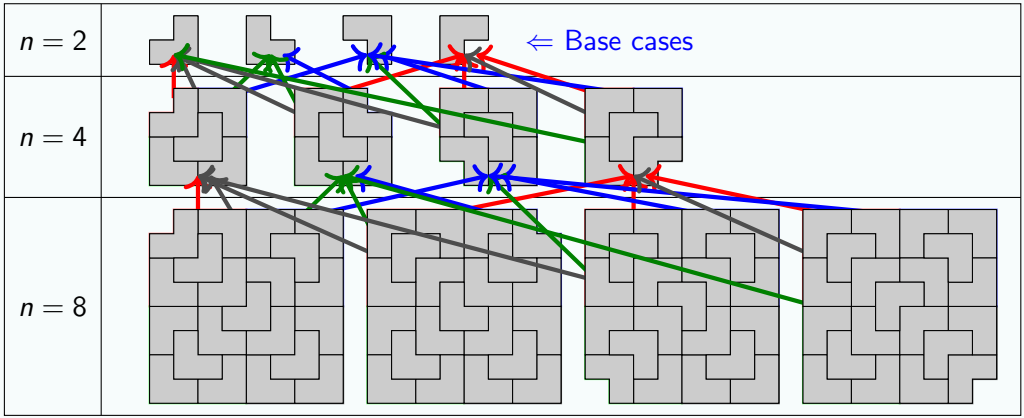
Let the board size be $n \times n$. What is the time complexity of this algorithm?
(Note: We do not consider rotated states as overlapping subproblems.)

A. $\Theta(n^2)$
B. $\Theta(n^2 \log n)$
C. $\Theta(n^2 \log^2 n)$
D. $\Theta(n^{\log_2 3})$

Solution.
We simply sum up the time needed to compute each **distinct** subproblem.
$T(n) = cn^2 + c(\frac{n}{2})^2 + c(\frac{n}{4})^2 + \cdots + c2^2 \leq 2cn^2$.
$\therefore T(n) = \Theta(n^2)$.

Extra: Top-down vs Bottom-up

Bottom-Up



Extra: Top-down vs Bottom-up

Let the board size be $n \times n$. What is the time complexity of this algorithm?
(Note: We do not consider rotated states as overlapping subproblems.)

A. $\Theta(n^2)$
B. $\Theta(n^2 \log n)$
C. $\Theta(n^2 \log^2 n)$
D. $\Theta(n^{\log_2 3})$

Solution.
This algorithm is iterative.
$T(n) = cn^2 + c(\frac{n}{2})^2 + c(\frac{n}{4})^2 + \cdots + c2^2 \leq 2cn^2$.
$\therefore\ T(n) = \Theta(n^2)$.