

Dynamic Programming (Part1)

Group 9



Review

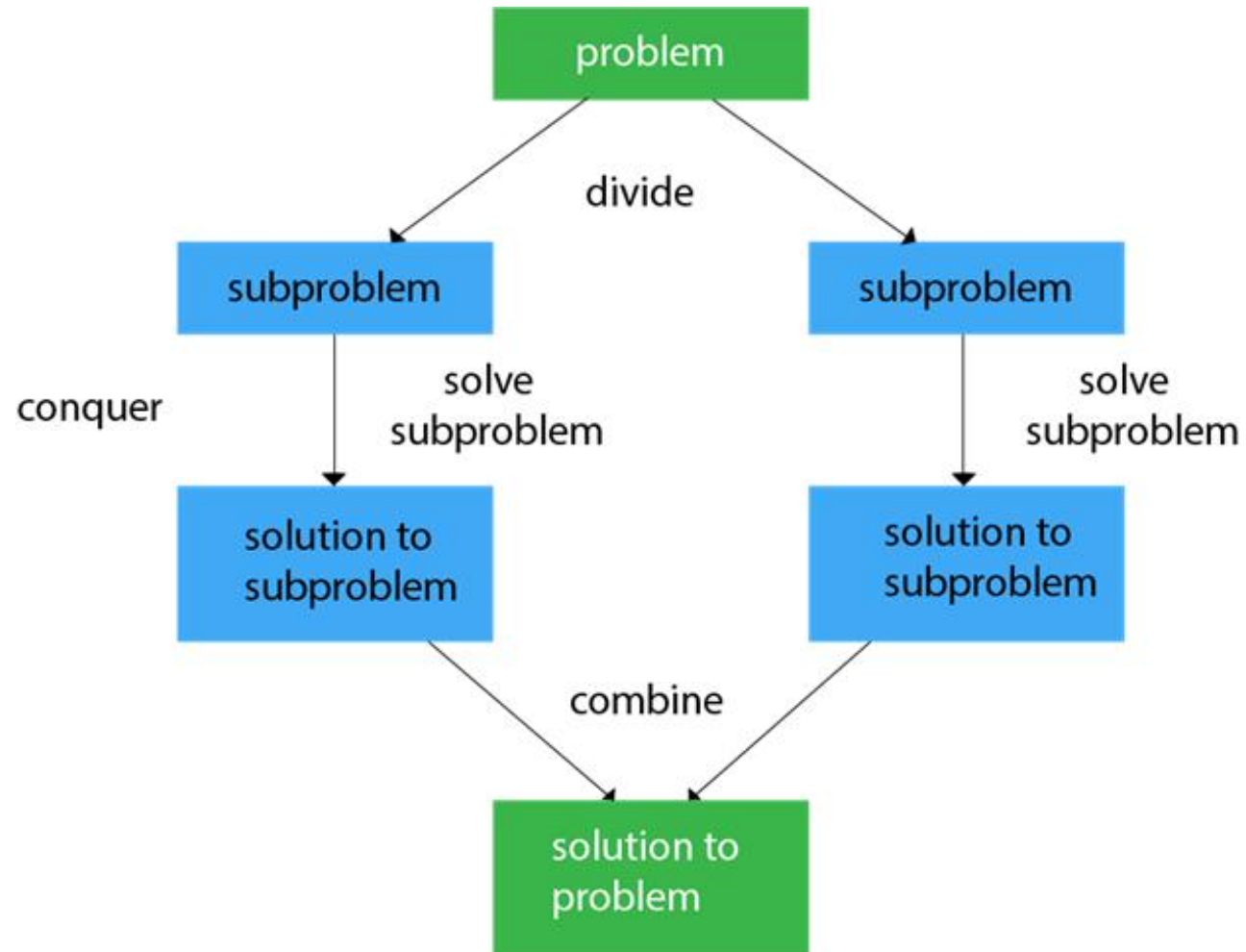


Figure 1: Divide and Conquer

Review

$$f(n) = \begin{cases} n = 1 & 1 \\ n = 2 & 1 \\ n > 2 & f(n-1) + f(n-2) \end{cases}$$

Figure 1: *Fibonacci sequence definition*

Review

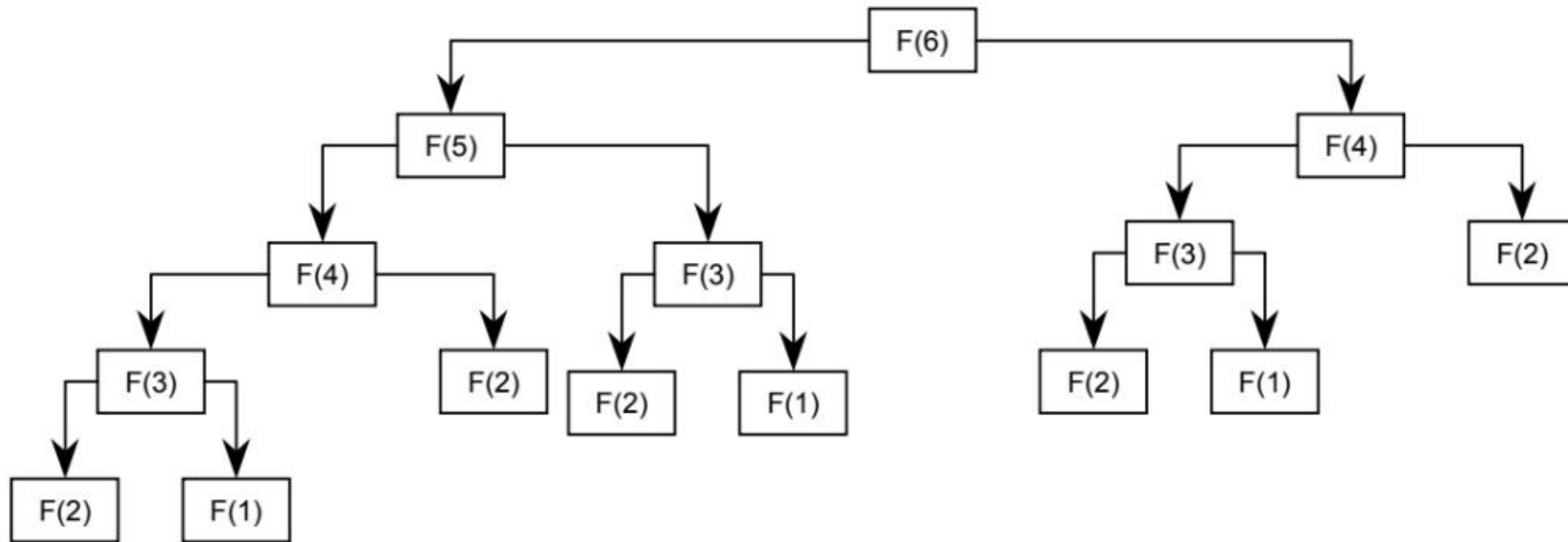


Figure 2: 6th Fibonacci number

Review

How many times do we
need to calculate the value
of $F(3)$?



We can use a **table** to
store the result

Dynamic Programming

- Dynamic Programming is a general algorithm design technique for solving problems defined by recurrences with overlapping subproblems
- Invented by American mathematician Richard Bellman in the 1950s to solve optimization problems and later assimilated by CS
- “Programming” here means “planning”



Applications

Application areas:

- Computer science: AI, compilers, systems, graphics, theory,....
- Operations research.
- Information theory.
- Control theory.
- Bioinformatics.

Applications

Some famous dynamic programming algorithms:

- Avidan–Shamir for seam carving.
- Unix diff for comparing two files.
- Viterbi for hidden Markov models.
- De Boor for evaluating spline curves.
- Bellman–Ford–Moore for shortest path.
- Knuth–Plass for word wrapping text in .
- Cocke–Kasami–Younger for parsing context-free grammars.
- Needleman–Wunsch/Smith–Waterman for sequence alignment.

Dynamic Programming

Main idea:

- Set up a recurrence relating a solution to a larger instance to solutions of some smaller instances.
- Solve smaller instances once.
- Record solutions in a table .
- Extract solution to the initial instance from that table.

Difference between DP and Divide-and-Conquer

- Using Divide-and-Conquer to solve these problems is **inefficient** because the **same** common **subproblems** have to be solved **many times**.
- DP will solve each of them **once** and **their answers are stored in a **table**** for future use.

Elements of Dynamic Programming

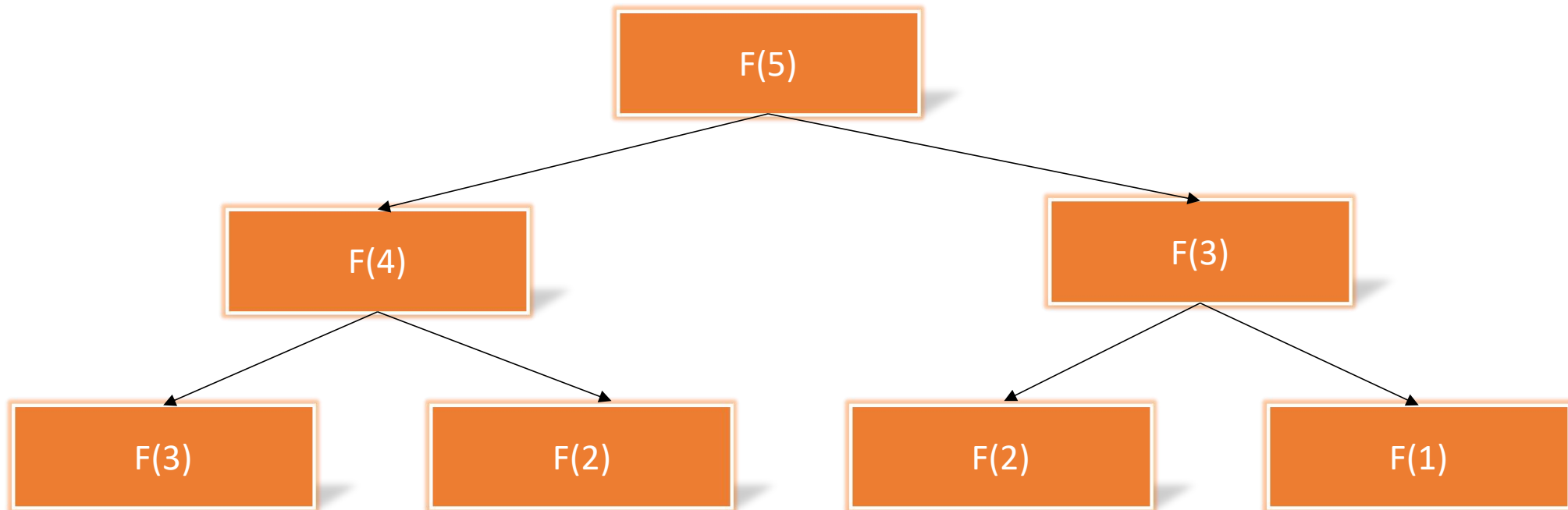
DP is used to solve problems with the following characteristics:

- **Simple subproblems:** We should be able to break the original problem to **smaller subproblems** that have the same structure
- **Optimal substructure of the problems:** The optimal solution to the problem contains within **optimal solutions to its subproblems**.
- **Overlapping sub-problems:** there exist some places where we solve the same subproblem more than **once**.

Top-down approach (memoization)

- Break down the problem into smaller overlapping subproblems.
- Start with the main problem and recursively solves smaller subproblems.
- Store the solutions of subproblems in a lookup table (cache) to avoid redundant calculations.
- Whenever a subproblem needs to be solved, it first checks if its solution is already available in the lookup table. If so, it uses the stored value; otherwise, it computes the solution and stores it for future use.
- This approach is often implemented using recursion

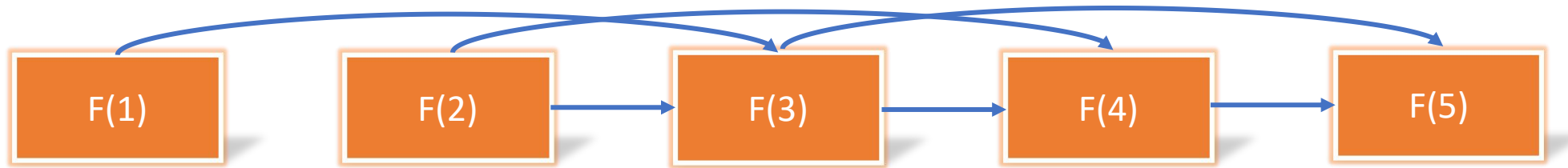
Top-down approach (memoization)



Bottom-up approach (Tabular)

- Solve a problem by iteratively building the solution from the smallest subproblems up to the main problem.
- Start by solving the smallest subproblems, storing the solutions in a table, and using their solutions to solve larger subproblems until the main problem is solved.
- This approach is often implemented using iteration, such as loops, and is efficient as it avoids recursive function calls and redundant computations.

Bottom-up approach (Tabular)



Top-down vs Bottom-up

	Top-down	Bottom-up
Easier to implement	✓	
Time efficiency		✓
Memory efficiency		✓
Doesn't need to compute all subproblems	✓	

Steps to Designing a Dynamic Programming Algorithm

1. Identify if it is a Dynamic programming problem.
2. Decide a state expression with the Least parameters.
3. Formulate state and transition relationship.
4. Do tabulation (or memorization).

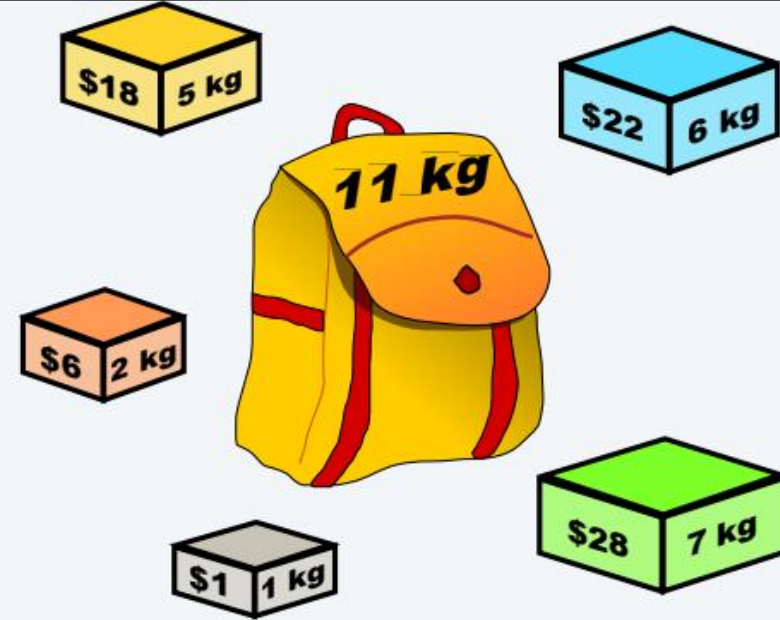
Knapsack problem

Goal. Pack knapsack so as to maximize total value of items taken.

- There are n items: item i provides value $v_i > 0$ and weighs $w_i > 0$ (Assume that all values and weights are integral).
- Value of a subset of items = sum of values of individual items.
- Knapsack has weight limit of W .

Ex. The subset { 1, 2, 5 } has value \$35 (and weight 10).

Ex. The subset { 3, 4 } has value \$40 (and weight 11).



i	v_i	w_i
1	\$1	1 kg
2	\$6	2 kg
3	\$18	5 kg
4	\$22	6 kg
5	\$28	7 kg

knapsack instance
(weight limit $W = 11$)

Knapsack problem

Def. $\text{OPT}(i, w)$ = optimal value of knapsack problem with items 1, ..., i , subject to weight limit w .

Goal. $\text{OPT}(n, W)$.

Case 1. $\text{OPT}(i, w)$ does not select item i .

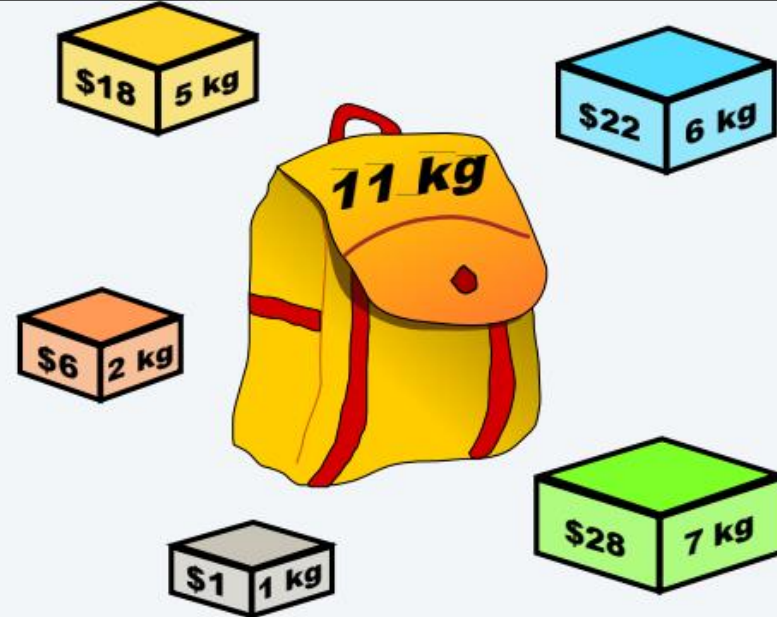
- $\text{OPT}(i, w)$ selects best of $\{ 1, 2, \dots, i - 1 \}$ subject to weight limit w .

$$\rightarrow \text{OPT}(i, w) = \text{OPT}(i-1, w)$$

Case 2. $\text{OPT}(i, w)$ selects item i .

- Collect value v_i .
- New weight limit = $w - w_i$.
- $\text{OPT}(i, w)$ selects best of $\{ 1, 2, \dots, i - 1 \}$ subject to new weight limit

$$\rightarrow \text{OPT}(i, w) = \text{OPT}(i-1, w-w_i) + v_i$$



i	v_i	w_i
1	\$1	1 kg
2	\$6	2 kg
3	\$18	5 kg
4	\$22	6 kg
5	\$28	7 kg

knapsack instance
(weight limit $W = 11$)

Knapsack problem

Base case:

$$OPT(0, w) = 0$$

Formula:

$$OPT(i, w) = \begin{cases} 0 & \text{if } i = 0 \\ OPT(i - 1, w) & \text{if } w_i > w \\ \max \{ OPT(i - 1, w), v_i + OPT(i - 1, w - w_i) \} & \text{otherwise} \end{cases}$$

Knapsack problem

KNAPSACK($n, W, w_1, \dots, w_n, v_1, \dots, v_n$)

FOR $w = 0$ TO W

$M[0, w] \leftarrow 0.$

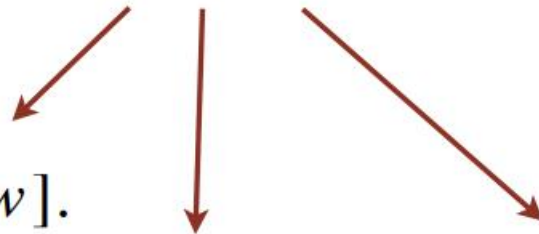
FOR $i = 1$ TO n

FOR $w = 0$ TO W

IF ($w_i > w$) $M[i, w] \leftarrow M[i-1, w].$

ELSE $M[i, w] \leftarrow \max \{ M[i-1, w], v_i + M[i-1, w - w_i] \}.$

previously computed values




RETURN $M[n, W].$

Knapsack problem: Time complexity

Theorem. The DP algorithm solves the knapsack problem with n items and maximum weight W in $\Theta(n W)$ time and $\Theta(n W)$ space.

Pf.

- Takes $O(1)$ time per table entry.
- There are $\Theta(n W)$ table entries.



weights are integers
between 1 and W

A problem about divisors

Problem: Given a number n , find the smallest positive integer with exactly n divisors.

- **Input:** A single integer n ($1 \leq n \leq 1000$).
- **Output:** The smallest positive integer that satisfies the requirement (It is guaranteed that the output of all test cases does not exceed 10^{18}).

A problem about divisors

For every integer $n > 1$, we can represent n using the following form:

$$n = p_1^{\alpha_1} p_2^{\alpha_2} p_3^{\alpha_3} \dots p_k^{\alpha_k}$$

where p_1, p_2, \dots, p_k are distinct primes and $\alpha_1, \alpha_2, \dots, \alpha_k$ are positive integers.

Therefore, **the number of divisors is:**

$$\pi(n) = (\alpha_1 + 1)(\alpha_2 + 1) \dots (\alpha_k + 1)$$

A problem about divisors

How can we define $dp(i, j)$?

$dp(i, j)$ is the largest number has exactly i divisors and the greatest prime divisor is j , then?



THANK YOU