COMPLETE SEARCH+
DYNAMIC PROGRAMMING + GREEDY

COMP 321 – McGill University

These slides are mainly compiled from the following resources.

- Professor Jaehyun Park' slides CS 97SI
- Top-coder tutorials.
- Programming Challenges books.

Algorithmic Paradigms

- General approaches to the construction of *efficient* solutions to problems.
- Such methods are of interest because:
  - They provide templates suited to solving a broad range of diverse problems.
  - They can be translated into common control and data structures provided by most high-level languages.
  - The temporal and spatial requirements of the algorithms which result can be precisely analyzed.
- Although more than one technique may be applicable to a specific problem, it is often the case that an algorithm constructed by one approach is clearly superior to equivalent solutions built using alternative techniques.

Outline

- Complete Search.
  - Iterative.
  - Recursive Backtracking.
- Divide and Conquer.
- Dynamic Programming.
  - 1 Dimensional.
  - 2 Dimensional.
  - Interval.
  - Tree
  - Subset
- Greedy
- Conclusions

Complete search

- Also know as recursive backtracking or brute force.
- It is a method for solving a problem by searching (up to) the entire search space in bid to obtain the required solution.
- A bug-free Complete Search solution should never receive Wrong Answer (WA) response in programming contests as it explores the entire search space; however, it may receive a Time Limit Exceeded (TLE) verdict.

Iterative Complete Search– example 1

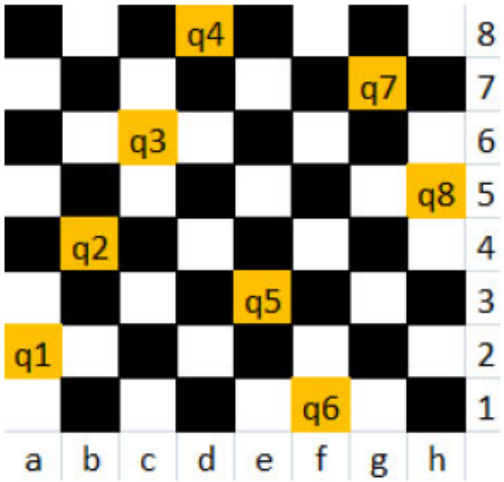
- Problem: Find and display all pairs of 5-digit numbers that between them use the digits 0 through 9 once each, such that the first number divided by the second is equal to an integer N, where $2 \leq N \leq 79$.
  - That is, abcde / fghij = N, where each letter represents a different digit.
  - The first digit of one of the numerals is allowed to be zero, e.g.
    - 79546 / 01283 = 62; 94736 / 01528 = 62.

Iterative Complete Search– example 1

- Solution 1:
  - Run an O(10!) algorithm that permutes *abcdefghij*.
- Solution 2:
  - *fghij* can only be from 01234 to 98765, which is ≈ 100K possibilities
  - For each tried *fghij*, we can get *abcde* from *fghij* * N and then check if all digits are different.

Recursive Backtracking– example 2

- Problem: In chess (with a standard 8x8 board), it is possible to place eight queens on the board so that no queen can be taken by any other. Write a program that will determine all such possible arrangements given the initial position of one of the queens (i.e. coordinate (a, b) in the board must contain a queen).



Recursive Backtracking– example 1

- Solution 1:
  - Use a solution vector where a_i is true iff there is a queen on the ith square.
  - There are $2^{64} \approx 1.84 \times 10^{19}$ different true/false vectors for 8X8 board.
- Solution 2:
  - Have the ith element of a solution vector explicitly list the square where the ith queen resides. a_i will be an integer from 1 to n^2.
  - There are $64^8 \approx 2.81 \times 10^{14}$ vectors.
- Solution 3:
  - Prune solution 2 by removing symmetries. Ensure that the queen in a_i sits on a higher number square that the queen in a_{i-1}
  - This change will reduce the search space to $\binom{64}{8} = 4.426 \times 10^9$

Recursive Backtracking– example 1

- Solution 4:
  - Note that there must be exactly one queen per row for an n-queens solution. Then you can limit the candidates of the ith queen to the eight squares on the ith row.
  - There are $8^8 \approx 1.67 \times 10^7$ vectors for 8X8 board.
- Solution 5:
  - Since no two queens can share the same column, we know that the n columns of a complete solution must form a permutation of n.
  - We reduce then our search space to just 8! = 40320 vectors.
- Solution 6:
  - You can improve solution 5 by knowing that no two queens can share any of the two diagonals.

Tips

- Generating versus Filtering:
  - Programs that generate lots of candidate solutions and then choose the ones that are correct are called ‘filters’ - recall the naive 8-queens solvers.
  - Those that hone in exactly to the correct answer without any false starts are called ‘generators’ – recall the improved 8-queens solver with 8! complexity plus diagonal checks.
  - Generally, filters are easier to code but run slower. Do the math to see if a filter is good enough
- Prune Infeasible Search Space Early:
- Utilize Symmetries.
  - In the 8-queens problem, there are 92 solutions but there are only 12 unique (or fundamental) solutions as there are rotations and reflections symmetries in this problem. You can utilize this fact by only generating the 12 unique solutions and, if needed, generate the whole 92 by rotating and reflecting these 12 unique solutions

Tips

- Pre-Computation:
  - If we know that there are only 92 solutions, then we can create a 2-dimensional array int solution[92][8] and then fill it with all 92 valid permutations. Then, we generate a new program and submit the code that just prints out the correct permutations with 1 queen at (a, b) (very fast).
- Try Solving the Problem Backwards.
  - Some contest problems seem far easier when they are solved backwards than when they are solved using a frontal attack.

Tips

- Optimizing Source Code:
  - Use the faster C-style scanf/printf rather than cin/cout.
  - Use the expected O(n log n) but cache-friendly quicksort rather than the true O(n log n) but not (cache) memory friendly mergesort.
  - Access a 2-D array in a row major fashion (row by row) rather than column by column.
  - Bitwise manipulation on integer is faster than using an array of bits.
  - Declare a bulky data structure just once by setting it to have global scope, so you do not have to pass the structure as function arguments.
  - Allocate memory just once, according to the largest possible input in the problem description, rather than re-allocating it for every test case in a multiple-input problem.

Tips

- Use Better Data Structure & Algorithm:
  - Using better data structures and algorithms always outperforms any optimization tips mentioned by the previous Tips. If all else fails, abandon Complete Search approach.

Divide and Conquer

- It is a problem solving paradigm where we try to make a problem simpler by ‘dividing’ it into smaller parts and ‘conquering’ them. The steps:
  - 1. Divide the original problem into sub-problems – usually by half or nearly half,
  - 2. Find (sub-)solutions for each of these sub-problems – which are now easier,
  - 3. If needed, combine the sub-solutions to produce a complete solution for the main problem.

Dynamic Programming

- Steps for Solving DP Problems
  - 1. Define subproblems
  - 2. Write down the recurrence that relates subproblems
  - 3. Recognize and solve the base cases
- Key ingredients to make DP works
  - This problem has optimal sub-structures.
    - Solution for the sub-problem is part of the solution of the original problem.
  - This problem has overlapping sub-problems.

1-dimensional DP – example 1

- Problem: given n, find the number of different ways to write n as the sum of 1, 3, 4.
- Example: for n = 5, the answer is 6

5 = 1 + 1 + 1 + 1 + 1

= 1 + 1 + 3

= 1 + 3 + 1

= 3 + 1 + 1

= 1 + 4

= 4 + 1

1-dimensional DP – example 1

- Define subproblems
  - Let D_n be the number of ways to write *n* as the sum of 1, 3, 4
- Find the recurrence
  - Consider one possible solution $n = x_1 + x_2 + \ldots + x_m$
  - If $x_m = 1$, the rest of the terms must sum to $n - 1$
  - Thus, the number of sums that end with $x_m = 1$ is equal to D_{n-1}
  - Take other cases into account ($x_m = 3$, $x_m = 4$)

1-dimensional DP – example 1

- Recurrence is then

$$D_n = D_{n-1} + D_{n-3} + D_{n-4}$$

- Solve the base cases
  - $D_0 = 1$
  - $D_n = 0$ for all negative n
  - Alternatively, can set: $D_0 = D_1 = D_2 = 1$, and $D_3 = 2$

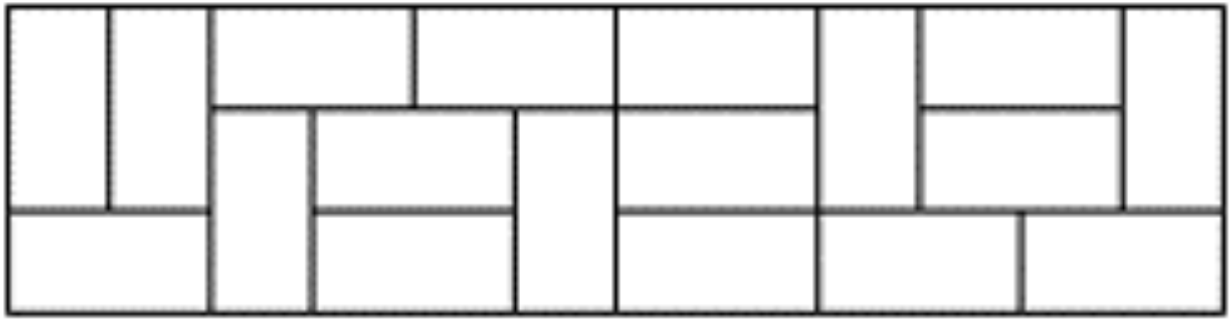
1-dimensional DP – example 1

```
D[0] = D[1] = D[2] = 1; D[3] = 2;
for(i = 4; i <= n; i++)
    D[i] = D[i-1] + D[i-3] + D[i-4];
```

- Extension: solving this for huge n, say $n \approx 10^{12}$
  - PLEASE check the matrix form of Fibonacci numbers

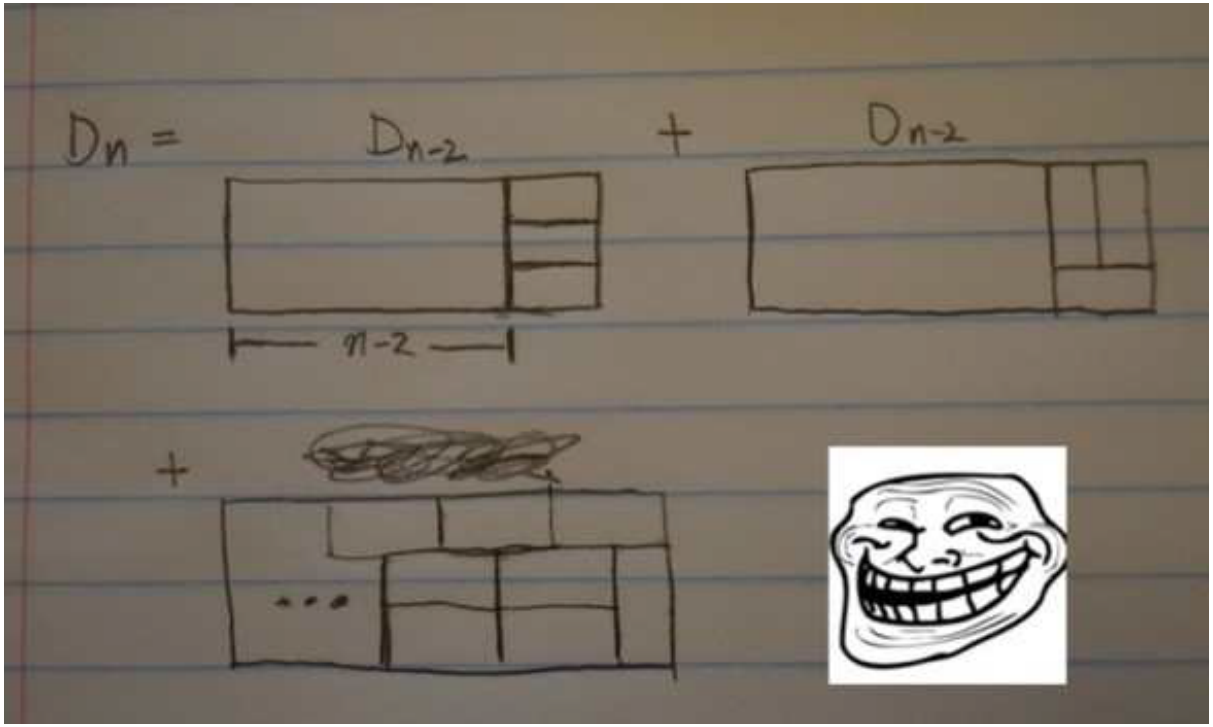
1-dimensional DP – example 2

- Given *n*, find the number of ways to fill a 3 X *n* board with dominoes
- Here is one possible solution for n = 12



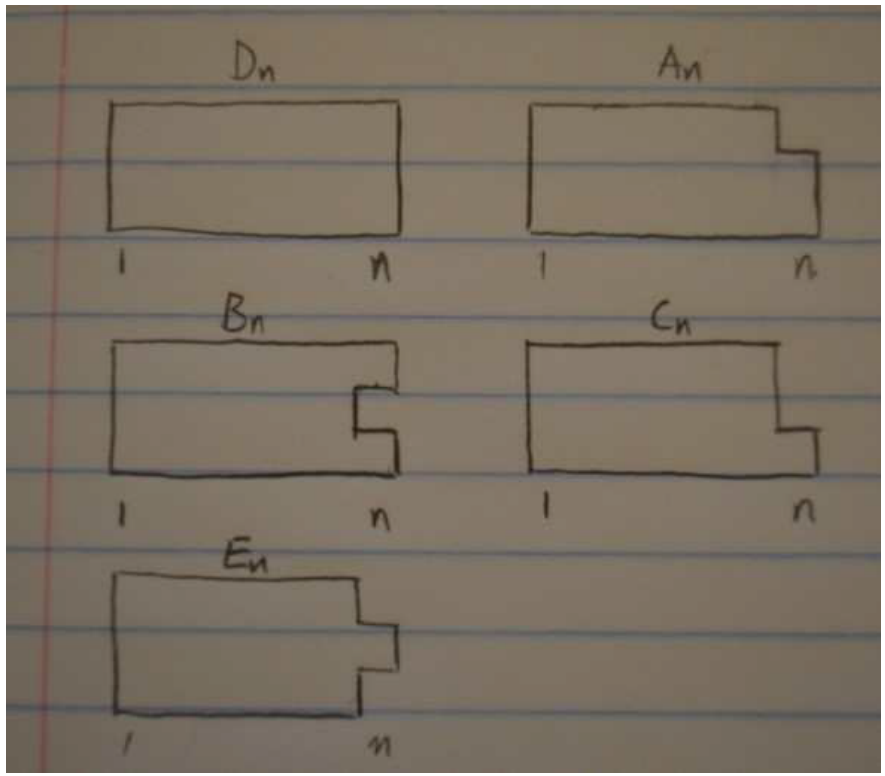
1-dimensional DP – example 2

- Define subproblems
  - Define D_n as the number of ways to tile a 3 X *n* board
- Find recurrence
  - Lets try the previous definition (example 1)



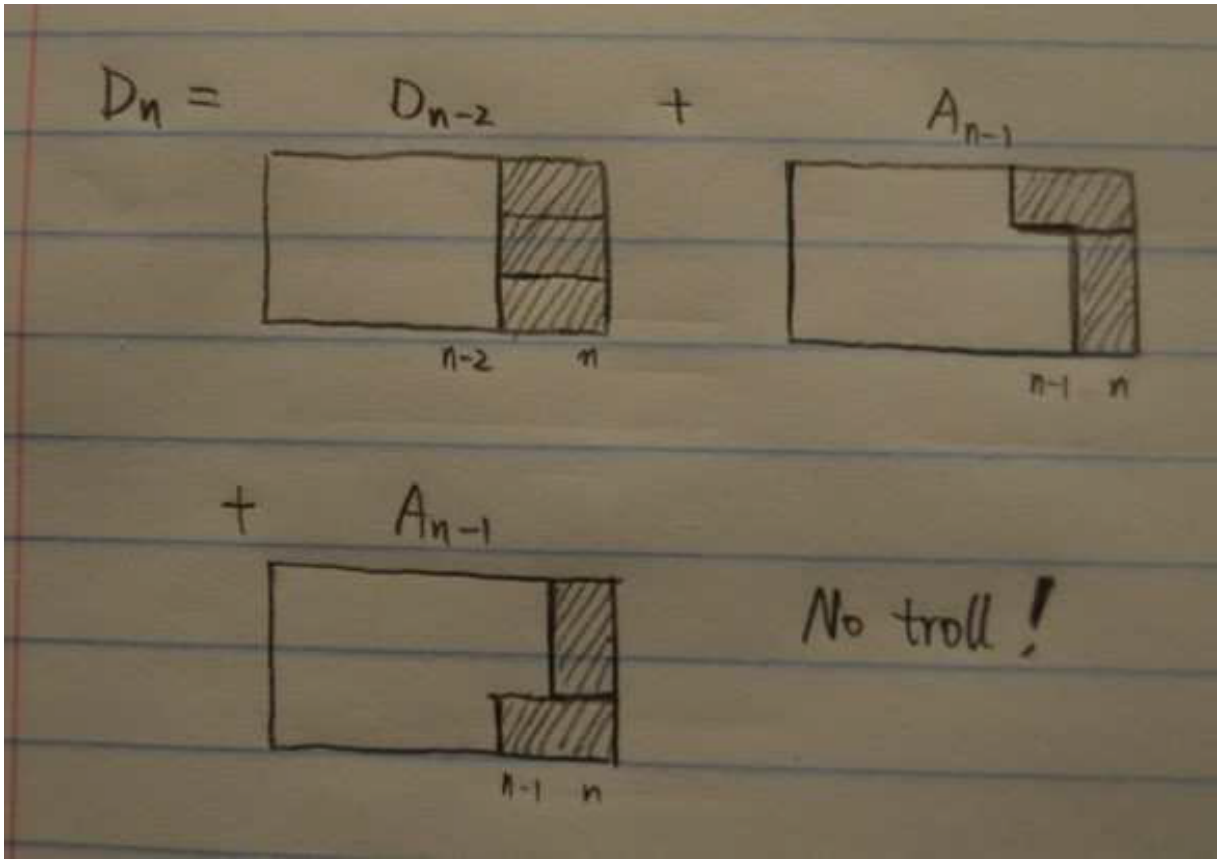
1-dimensional DP – example 2

- The problem is that D_n's don't relate in simple terms
- What if we introduce more subproblems?
  - Consider different ways to fill the nth column and see what the remaining shape is



1-dimensional DP – example 2

- We can now define $D_n = D_{n-2} + A_{n-1} + A_{n-1}$



2-dimensional DP – example 1

- Problem: given two strings x and y, find the longest common subsequence (LCS) and print its length
- Example:
  - x : A**BC**BD**AB**
  - y : **B**D**CAB**C
  - “BCAB ” is the longest subsequence found in both sequences, so the answer is 4

2-dimensional DP – example 1

- Define subproblems
  - Let D_{ij} be the length of the LCS of $x_{1...i}$ and $y_{1...j}$
- Find the recurrence
  - If $x_i = y_j$, they both contribute to the LCS => match
    - $D_{ij} = D_{i-1,j-1} + 1$
  - Otherwise, either x_i or y_j does not contribute to the LCS, so one can be dropped
  - $D_{ij} = \max\{D_{i-1,j}, D_{i,j-1}\}$
- Find and solve the base cases: $D_{i0} = D_{0j} = 0$

2-dimensional DP – example 1

```
for(i = 0; i <= n; i++) D[i][0] = 0;
for(j = 0; j <= m; j++) D[0][j] = 0;
for(i = 1; i <= n; i++) {
    for(j = 1; j <= m; j++) {
        if(x[i] == y[j])
            D[i][j] = D[i-1][j-1] + 1;
        else
            D[i][j] = max(D[i-1][j], D[i][j-1]);
    }
}
```

Interval DP – example 1

- Problem: given a string $x = x_{1...n}$, find the minimum number of characters that need to be inserted to make it a palindrome
- Example:
  - x : Ab3bd
  - Can get “dAb3bAd ” or “Adb3bdA ” by inserting 2 characters (one ‘d ’, one ‘A ’)

Interval DP – example 1

- Define subproblems
  - Let D_{ij} be the minimum number of characters that need to be inserted to make $x_{i...j}$ into a palindrome
- Find the recurrence
  - Consider a shortest palindrome $y_{1...k}$ containing $x_{i...j}$
  - Either $y_1 = x_i$ or $y_k = x_j$
  - $y_{2...k-1}$ is then an optimal solution for $x_{i+1...j}$ or $x_{i...j-1}$ or $x_{i+1...j-1}$
    - Last case possible only if $y_1 = y_k = x_i = x_j$

Interval DP – example 1

- Find the recurrence

$$D_{ij} = \begin{cases} 1 + \min\{D_{i+1,j}, D_{i,j-1}\} & x_i \neq x_j \\ D_{i+1,j-1} & x_i = x_j \end{cases}$$

- Find and solve the base cases: $D_{ii} = D_{i,i-1} = 0$ for all I
- The entries of D must be filled in increasing order of j−i interval

Interval DP – example 1

- Find the recurrence

```
// fill in base cases here
for(t = 2; t <= n; t++)
    for(i = 1, j = t; j <= n; i++, j++)
        // fill in D[i][j] here
```

- Note how we use an additional variable t to fill the table in correct order

Interval DP – example 1

- An alternate solution.
  - Reverse x to get x^R
  - The answer is n − L, where L is the length of the LCS of x and x^R

Tree DP – example 1

- Problem: given a tree, color nodes black as many as possible without coloring two adjacent nodes
- Subproblems:
  - First, we arbitrarily decide the root node r
  - B_v : the optimal solution for a subtree having v as the root, where we color v black
  - W_v : the optimal solution for a subtree having v as the root, where we don't color v
  - Answer is $\max\{B_r, W_r\}$

Tree DP – example 1

- Find the recurrence
  - Crucial observation: once v's color is determined, subtrees can be solved independently
  - If v is colored, its children must not be colored

$$B_v = 1 + \sum_{u \in \text{children}(v)} W_u$$

  - If v is not colored, its children can have any color

$$W_v = 1 + \sum_{u \in \text{children}(v)} \max\{B_u, W_u\}$$

- Base cases: leaf nodes

Subset DP – example 1

- Problem: given a weighted graph with n nodes, find the shortest path that visits every node exactly once (Traveling Salesman Problem)
- Wait, isn't this an NP-hard problem?
  - Yes, but we can solve it in $O(n^2 2^n)$ time
  - Note: brute force algorithm takes $O(n!)$ time

Subset DP – example 1

- Define subproblems
  - $D_{S,v}$: the length of the optimal path that visits every node in the set S exactly once and ends at v
  - There are approximately $n2^n$ subproblems
  - Answer is $\min_{v \varepsilon V} D_{V,v}$, where V is the given set of nodes
- Let's solve the base cases first
  - For each node v , $D_{\{v\},v} = 0$

Subset DP – example 1

- Find the recurrence
  - Consider a path that visits all nodes in S exactly once and ends at v
  - Right before arriving v, the path comes from some u in S − {v}
  - And that subpath has to be the optimal one that covers S − {v} , ending at u
  - We just try all possible candidates for u

$$D_{S,v} = \min_{u \in S - \{v\}} \left(D_{S-\{v\},u} + \text{cost}(u, v) \right)$$

Subset DP – example 1

- When working with subsets, it's good to have a nice representation of sets
- Idea: Use an integer to represent a set
  - Concise representation of subsets of small integers {0, 1, . . .}
  - If the i-th (least significant) digit is 1, i is in the set
  - If the i th digit is 0, i is not in the set
  - e.g. , 19 = $010011_{(2)}$ in binary represent a set {0, 1, 4}

Subset DP – example 1 (extra)

- Union of two sets x and y: x | y
- Intersection: x & y
- Symmetric difference: x ^ y
- Singleton set {i}: 1 << i
- Membership test: x & (1 << i) != 0
- Set bit: x | (1 << i)
- Clear bit: x & ~(1 << i)

Subset DP – example 1 (extra)

- x ^ 0s = x
- x ^ 1s = ~x
- x ^ x = 0
- x & 0s = 0
- x & 1s = x
- x & x = x
- x | 0s = x
- x | 1s = 1s
- x | x = x

Bottom-Up DP

- 1. Identify the Complete Search recurrence.
- 2. Initialize some parts of the DP table with known initial values.
- 3. Determine how to fill the rest of the DP table based on the Complete Search recurrence, usually involving one or more nested loops to do so.

Top-Down DP

- 1. Initialize a DP ‘memo’ table with dummy values, e.g. ‘-1’.
  - The dimension of the DP table must be the size of distinct sub-problems.
- 2. At the start of recursive function, simply check if this current state has been computed before.
  - (a) If it is, simply return the value from the DP memo table, O(1).
  - (b) If it is not, compute as per normal (just once) and then store the computed value in the DP memo table so that further calls to this sub-problem is fast.

Top-Down VS Bottom-Up

Top-Down	Bottom-Up
Pro: 1. It is a natural transformation from normal Complete Search recursion 2. Compute sub-problems only when necessary (sometimes this is faster)	Pro: 1. Faster if many sub-problems are revisited as there is no overhead from recursive calls 2. Can save memory space with DP 'on-the-fly' technique (see comment in code above)
Cons: 1. Slower if many sub-problems are revisited due to recursive calls overhead (usually this is not penalized in programming contests) 2. If there are M states, it can use up to $O(M)$ table size which can lead to Memory Limit Exceeded (MLE) for some hard problems	Cons: 1. For some programmers who are inclined with recursion, this may be not intuitive 2. If there are M states, bottom-up DP visits and fills the value of *all* these M states