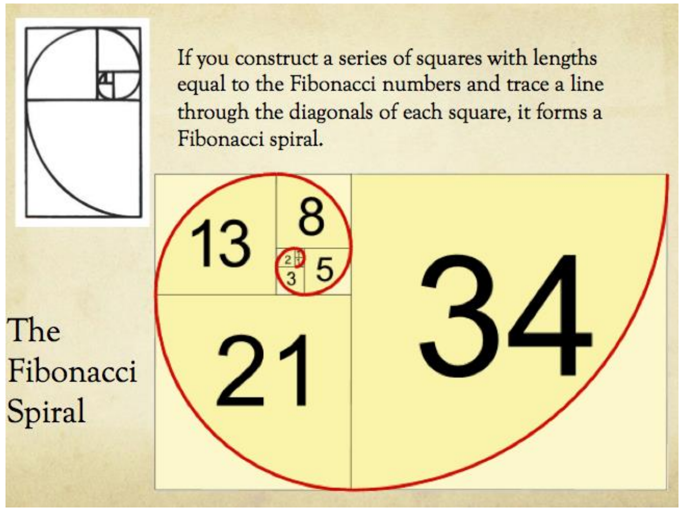
Dynamic programming 1

Outline

- Introduction
  - Fibonacci Number
  - Climbing Stairs
  - Unique Paths
  - House Robber
- SubArray, Substring, Subsequeces
  - Maximum Subarray (DP)
  - Longest Common Subsequence/SubString
  - Edit Distance
  - Longest Increasing Subsequence
- Knapsack Problems
  - 0/1 Knapsack, Target Sum
  - Unbounded Knapsack, Coin Change

Why DP?

- DP: dynamic programming.
- Key **idea**: **memorize** all the computation results.
  - In case later on you will reuse it.
  - -> query the results directly (fast) without re-compute (slow).
- Fibonacci numbers.
  - Base cases: $F_0 = 0$ and $F_1 = 1$.
  - General cases: $F_n = F_{n-1} + F_{n-2}$.
  - Examples:
    - $F_2 = F_{2-1} + F_{2-2} = F_1 + F_0 = 1 + 0 = 1$.
    - $F_3 = F_{3-1} + F_{3-2} = F_2 + F_1 = 1 + 1 = 2$.
    - $F_4 = F_{4-1} + F_{4-2} = F_3 + F_2 = 2 + 1 = 3$.
    - $F_5 = F_{5-1} + F_{5-2} = F_4 + F_3 = 3 + 2 = 5$.
- How to compute the Fibonacci numbers till n?
- An easy algorithm, recursion.


If you construct a series of squares with lengths equal to the Fibonacci numbers and trace a line through the diagonals of each square, it forms a Fibonacci spiral.
The Fibonacci Spiral
13
8
2
3
5
21
34

https://en.wikipedia.org/wiki/Fibonacci_number

https://leetcode.com/problems/fibonacci-number/

Fibonacci Numbers

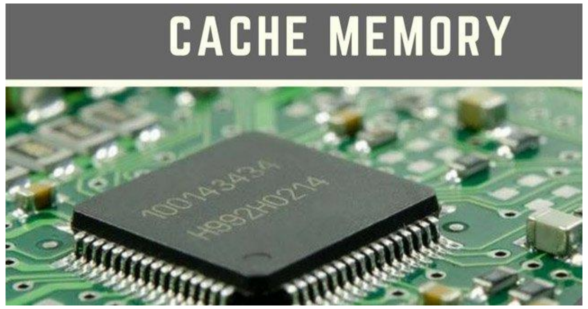
- Recursive algorithm:

```
$Fib(m)$
    if(m<=1) return m;
    return $Fib(m-1)+Fib(m-2)$
```

  Any drawbacks?
  - To compute $Fib(m)$, we have to compute all $Fib(2), Fib(3), \dots, Fib(m-1)$.
  - Actually compute multiple times except for $Fib(m-1)$.
- Example: $Fib(5)$.
  - We know $Fib(0)$ and $Fib(1)$.
  - 1st recursion: compute $Fib(4)$ and $Fib(3)$ for $Fib(5)$.
  - 2nd recursion: compute $Fib(3)$ and $Fib(2)$ for $Fib(4)$.
  - 3rd recursion: compute $Fib(2)$ for $Fib(3)$.
  - 4th recursion: compute $Fib(2)$ for $Fib(3)$.
- Observation: a tree structure with many **duplicated** nodes/computations.
- Memorization!

```
memo = []
$Fib(m)$
    if(m<=1) return m;
    if(m is in memo) return memo[m];
    else $memo[m] == Fib(m-1)+Fib(m-2);$ $return\ remo[m];$
```

(Faster) Fibonacci Numbers



- **Issue**: duplication -> **solution**: avoid duplication.
- How? -> **Save** intermediate **results**.
- Iterative algorithm:

```
Fib(n) {
    int dp[n+1];                                    //table to store the results
    dp[0] = 0; dp[1] = 1;                           //former terminal values
    for (i=2; i<n+1; ++i)                           //former recursion
        dp[i] = dp[i-2] + dp[i-1];                  //former main logic
    return dp[n+1];
}
```

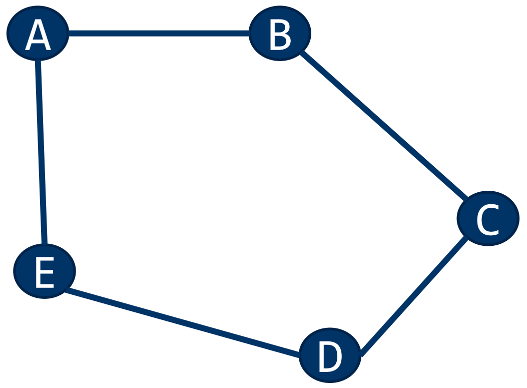
- Difference: recursive + memorization (last to first, **top-down**) vs. iterative (first to last, **bottom-up**)
  - Recursive: values that haven't been calculated yet, thus wait for the result (call another recursion).
  - Iterative: values in a table -> check the table, no re-calculating.
- Time complexity? **Linear**, only one for loop without recursive operations.

DP

- Hallmarks of DP
  - Hallmark 1:**Optimal substructure** An optimal solution to a problem (instance) contains optimal solutions to subproblems.
  - Hallmark 2:**Overlapping subproblems** A recursive solution contains a “small” number of distinct subproblems repeated many times.
- Another property to consider: Markov property, given a specific state, its future development depends only on the current state and is independent of all previously experienced states.
- Recursive algorithm implies a graph of computation
- Existence of recursive solution implies decomposable subproblems
- “Careful brute force” (Bottom up: do each subproblem in order)
- “Recurse but re-use” (Top down: record and lookup subproblem solutions)
- Often useful for **counting/optimization** problems: almost trivially correct recurrences

Annex: Problem without Optimal Substructure

- Assume the distance of each edge is the same one unit.
- Longest path from A to C: A -> E -> D -> C.
- According to optimal substructure (if it holds):
  - Path can be decomposed.
  - A -> E -> D -> C == (A -> E -> D + D -> C).
  - A -> E -> D should be the longest path from A to D.
  - True?
  - The longest path form A to D is: A -> B -> C -> D.
- Fact: this problem has **no optimal substructure.**

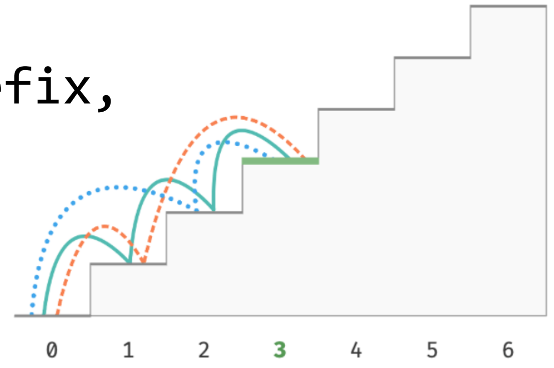


Solving DP Problems

- Hard part is thinking inductively to construct recurrence on subproblems
- How to solve a problem recursively (SRTBOT)
  1. **Subproblem** definition,
     - Describe the meaning of a subproblem in words, in terms of parameters (or define state)
     - Often subsets of input: prefixes, suffixes, contiguous substrings of a sequence
     - Often record partial state: add subproblems by incrementing some auxiliary variables
  2. **Relate** subproblem solutions recursively (or transit state)
     - dp(i)= f(dp(j),...) or dp[i] = f(dp[j],…) for one or more j<i
     - Common idea: **choose or not choose, which one to choose**
  3. **Topological** order on subproblems (⇒subproblem DAG (Directed acyclic graph)), e.g., the iteration order/direction
  4. **Base** cases of relation, initialization and boundary cases
  5. **Original** problem solution via subproblem(s)
  6. **Time** analysis: # of subproblems * cost to relate (transit)

Climbing Stairs

- Problem:
  - You are climbing a staircase. It takes n steps to reach the top.
  - Each time you can either climb 1 or 2 steps. In how many distinct ways can you climb to the top?
- Subproblems: dp[i]: distinct ways to get to step i, prefix,
- Relate: choose 1 or 2 steps: dp[i] = dp[i-1]+dp[i-2]
- Topological order: increasing i
- Base cases: dp[1] = 1, dp[2] = 2
- Original problem: dp[n]
- Time analysis: O(n)

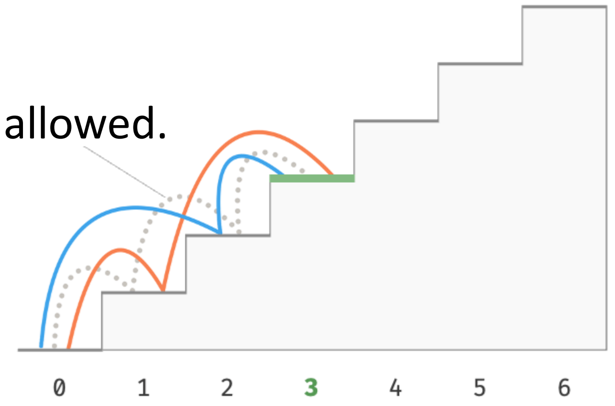


https://leetcode.com/problems/Climbing-Stairs/

Climbing Stairs

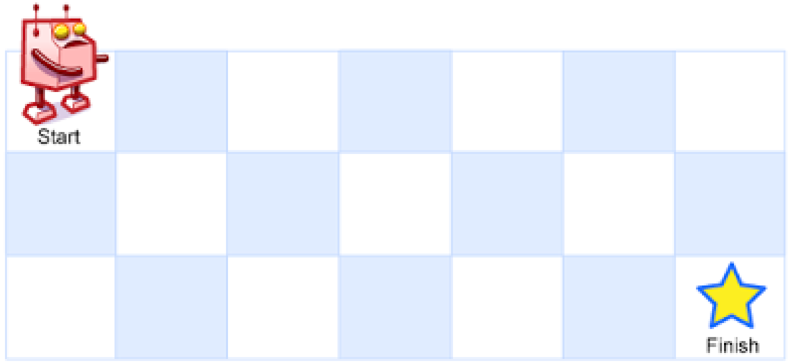
- If the problem becomes: Each time you can either climb 1 or 2 steps, but you can not climb two 1 steps consecutively. In how many distinct ways can you climb to the top?
- The next step choice cannot be determined solely by the current state (the current step number), but is also related to the previous state (the step number of the previous round), this is against Markov property
- dp[i] = dp[i-1]+dp[dp-2] does not work, need to reformulate the subproblems: dp[i,j]: at step i, through climb j steps at last jump
- Relate:
  - dp[i,1] = dp[i-1,2]
  - dp[i,2] = dp[i-2,1] + dp[i-2,2]
- Original problem: dp[n,1]+dp[n,2]

1,1,1 is not allowed.



Unique Paths

- Problem:
  - There is a robot on an m x n grid. The robot is initially located at the top-left corner (i.e., grid[0][0]). The robot tries to move to the bottom-right corner (i.e., grid[m - 1][n - 1]). The robot can only move either down or right at any point in time.
  - Given the two integers m and n, return the number of possible unique paths that the robot can take to reach the bottom-right corner.
  - Input: m = 3, n = 7
  - Output: 28
- Subproblems: dp[i,j]: distinct ways to get to get to grid i,j
- Relate: choose down or left: dp[i,j] = dp[i-1,j]+dp[i,j-1]
- Topological order: increasing i, increasing j
- Base cases: dp[i,0] = 1, dp[0,i] = 1
- Original problem: dp[m,n]
- Time analysis: O(m*n)
- How about "can move down or up or right or left"?



https://leetcode.com/problems/unique-paths/

House robber

- Problem:
  - You are a professional robber planning to rob houses along a street. Each house has a certain amount of money stashed, the only constraint stopping you from robbing each of them is that adjacent houses have security systems connected and it will automatically contact the police if two adjacent houses were broken into on the same night.
  - Given an integer array nums representing the amount of money of each house,return the maximum amount of money you can rob tonight without alerting the police.
  - Input: nums = [2,7,9,3,1]
  - Output: 12
  - Explanation: Rob house 1 (money = 2), rob house 3 (money = 9) and rob house 5 (money = 1).
  - Total amount you can rob = 2 + 9 + 1 = 12.

https://leetcode.com/problems/house-robber/

House robber

- Subproblems: dp[i]: maximum amount can rob in **first** i houses, prefix
- Relate:
  - Not choose i: maximum amount can rob in **first** i-1 houses
  - Choose i: maximum amount can rob in **first** i-2 houses + nums[i] (amount)
  - dp[i] = max(dp[i-1], dp[i-2]+nums[i])
- Topological order: increasing i
- Base cases: dp[0] = 0
- Original problem: dp[n], may need to use n-1 based on your implementation
- Time analysis: O(n)

Maximum Subarray (DP)

- Problem:
  - Given an integer array nums, find the subarray with the largest sum, and return its sum.
  - Input: nums = [-2,1,-3,4,-1,2,1,-5,4]
  - Output: 6
  - Explanation: The subarray [4,-1,2,1] has the largest sum 6.
- Subproblems: dp[i]: largest sum of nums[0…i], must include nums[i], subarray is continuous
- Relate:
  - Not choose prefix: nums[i]
  - Choose prefix: largest sum of nums[0…i-1] + nums[i]
  - dp[i] = max(nums[i], dp[i-1] + nums[i])
- Topological order: increasing i
- Base cases: dp[0] = nums[0]
- Original problem: max(dp[0],…,dp[n-1])
- Time analysis: O(n)

https://leetcode.com/problems/Maximum-Subarray/

Longest Common Subsequence (LCS)

- Problem: 2 strings s,t -> find a longest subsequence, a subsequence of both.
- Idea: **global** (whole sequences) optimal from **local** (subsequences) opt.
- Question: how to **decompose** the whole sequences into subsequences.
- Case 1: Prefix **end** in the **same** character. (prove with contradiction)
  - LCS(AxBxC, AyyyyBC) = LCS(AxBx, AyyyyB) + “C”
- Case2: Prefix **don’t have** a **common end** character. (prove with subcases)
  - LCS(AxBx, AyyyyB) = Longest(LCS(AxBx, Ayyyy), LCS(AXB, AyyyyB))
- Recursive algorithm:
- dfs(s1, s2) {

```
    i = s.size()-1;          //last index in the first string
    j = t.size()-1;          //last index in the second string
    if ( i==0 or j==0 ) return 0;
    if ( s[i] == t[j] ) return 1 + dfs(s[0:i-1], t[0:j-1]);
    return longest(dfs(s[0:i-1], t[0:j]), dfs(s[0:i], t[0:j-1]) );
}
```

https://leetcode.com/problems/longest-common-subsequence/

Relate Subproblems of LCS

- Subproblems: dp[i,j]: length of LCS of s[0…i] and t[0…j]
- Relate:
  - Not choose s[i], choose t[j]: dp[i-1,j]
  - Choose s[i], not choose t[j]: dp[i,j-1]
  - Choose s[i], choose t[j]: dp[i-1,j-1] + 1
  - Not choose s[i], not choose t[j]: dp[i-1,j-1]
  - dp[i,j] = max(dp[i-1,j],dp[i,j-1], dp[i-1,j-1]+1) if s[i]==t[j] LCS(AxBxC, AyyyyBC)

    max(dp[i-1,j],dp[i,j-1], dp[i-1,j-1]) if s[i]!==t[j] LCS(AxBx, AyyyyB)
- Two cases:
  - if s[i]==t[j] consider dp[i-1,j],dp[i,j-1]?
    - No, max(dp[i-1,j],dp[i,j-1])<=dp[i-1,j-1]+1, e.g., dp[i-1,j-1]+1 vs dp[i-1,j]

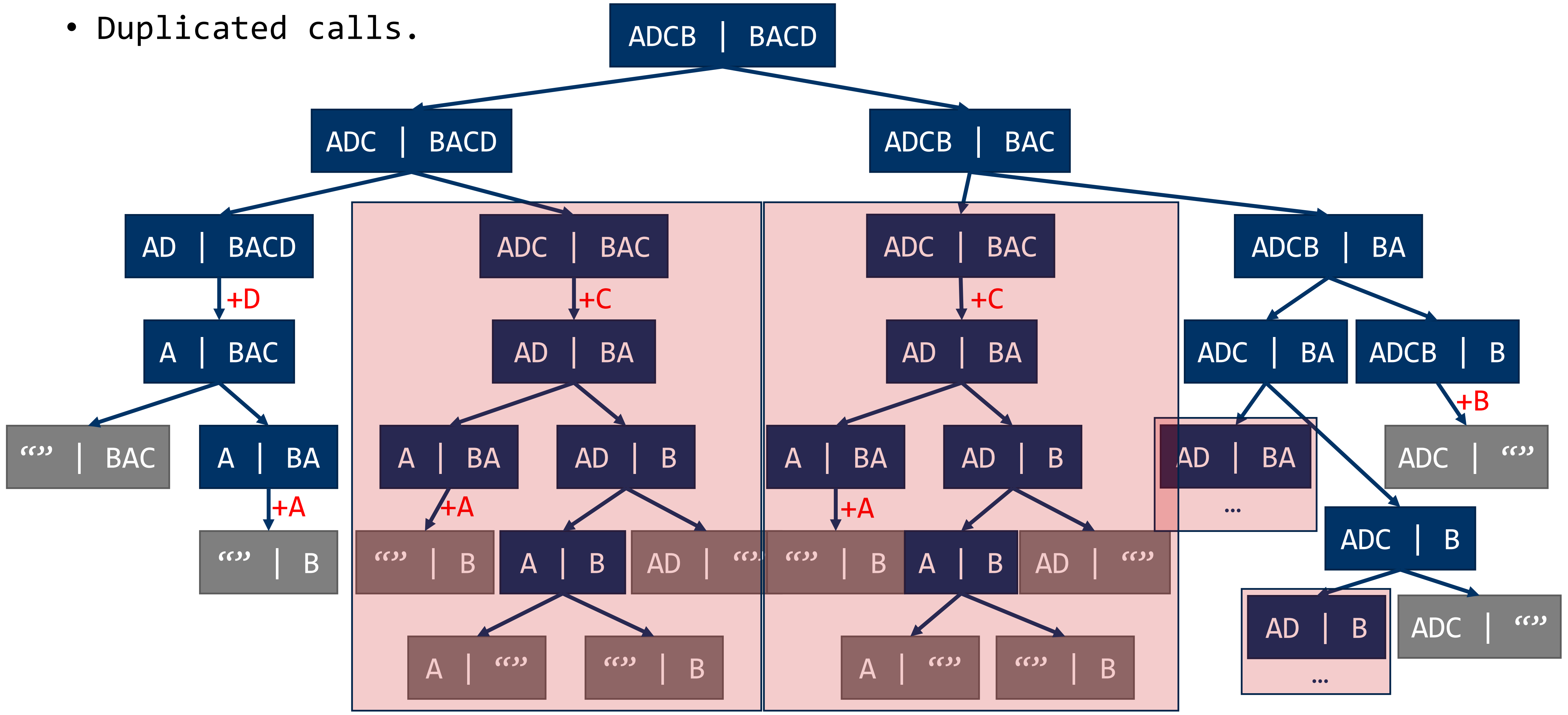
s: abcdc	abcd	abcd	abd
t: abc	ab	abc	ab
	x=dp[i-1,j-1]	suppose dp[i-1,j]>x+1	then>x contradiction it should be <=x

  - if s[i]!==t[j] consider dp[i-1,j-1]?
    - No, dp[i-1, j-1] is included in dp[i-1,j],dp[i,j-1], that is dp[i-1,j-1] <= min(dp[i-1,j],dp[i,j-1])
  - dp[i,j] = dp[i-1,j-1]+1 if s[i]==t[j]

    max(dp[i-1,j],dp[i,j-1]) if s[i]!==t[j]

Drawback of Recursive Algorithm

- Duplicated calls.



Solution: Recursive -> Iterative

- Iterative algorithm (from small to large)
  - if (s1[i] == s2[j])
    - dp[i][j] = 1 + dp[i-1][j-1]);
  - else
    - dp[i][j] = max(dp[i-1][j], dp[i][j-1]);
- Solution: top-left to bottom-right corner
- Look for the value changes. Multiple paths possible.

Topological order: increasing i,j
Base cases: dp[0,j] = 0, dp[i,0] = 0
Original problem: backtrace dp[m,n]
Time analysis: O(m*n)

	Ø	a	d	c	b
Ø	Ø	Ø	Ø	Ø	Ø
b	Ø				
a	Ø				
c	Ø				
d	Ø				

	Ø	a	d	c	b
Ø	Ø	Ø	Ø	Ø	Ø
b	Ø	Ø	Ø	Ø	1
a	Ø	1	1	1	1
c	Ø	1	1	2	2
d	Ø	1	2	2	2

	Ø	a	d	c	b
Ø	Ø	Ø	Ø	Ø	Ø
b	Ø	Ø	Ø	Ø	1
a	Ø	**1**	1	1	1
c	Ø	1	1	**2**	2
d	Ø	1	**2**	2	2

Annex: Optimal Substructure – Proof (LCS)

- Recap:
  - Two strings X[1:m] = x1,x2,...,xm and Y[1:n] = y1,y2,...,yn.
  - If xm == yn: LCS(X[1:m], Y[1:n]) = LCS(X[1:m-1], Y[1:n-1]).
  - If xm != yn: LCS(X[1:m], Y[1:n]) = longest(LCS(X[1:m], Y[1:n-1]), LCS(X[1:m-1], Y[1:n])).
- Not all problem have optimal substructure. We need to prove it.
- Suppose Z[1:k] = LCS(X,Y) = z1,z2,...,zk.
- Case 1: xm == yn
  - zk == xm == yn
  - LCS(X[1:m-1], Y[1:n-1]) = Z[1:k-1]
- Case 2: xm != yn
  - zk != xm OR zk != yn
  - Z[1:k] = longest(LCS(X[1:m], Y[1:n-1]), LCS(X[1:m-1], Y[1:n]))

Annex: Optimal Substructure – Proof (LCS)

- Case 2: xm != yn
  - zk != xm OR zk != yn
  - Z[1:k] = longest(LCS(X[1:m], Y[1:n-1]), LCS(X[1:m-1], Y[1:n]))
- Case 2.1: xm != yn AND zk != xm; then Z[1:k] = LCS(X[1:m-1], Y[1:n])).
- Case 2.2: xm != yn AND zk != yn; then Z[1:k] = LCS(X[1:m], Y[1:n-1])).
- Idea: prove 3 cases, Case 1, Case 2.1, and Case 2.2, one by one.
- Case 1: xm == yn. How to prove? (Hint: contradiction)
  - zk == xm == yn. Why?
    - If zk != xm, then Z[1:k]xm is the LCS of X and Y, longer than Z, contradiction!
  - LCS(X[1:m-1], Y[1:n-1]) = Z[1:k-1]. Why?
    - Assume W[1:w] is also a common sequence of (X[1:m-1], Y[1:n-1]), and w > k-1.
    - W[1:w]xm is a common sequence of (X[1:m], Y[1:n]) of length w+1 > k.
    - Z is the LCS (no one can be longer than k) based on the definition.
    - Contradiction! No such W[1:w].

Annex: Optimal Substructure – Proof (LCS)

- Case 2.1: xm != yn AND zk != xm; then Z[1:k] = LCS(X[1:m-1], Y[1:n])).
  - Assume W[1:w] is also a common sequence of (X[1:m-1], Y[1:n]), and w > k.
  - W[1:w] is also a common sequence of X[1:m] and Y[1:n], of length w > k.
  - Z is the LCS (no one can be longer than k) based on the definition.
  - Contradiction! No such W[1:w]. |LCS(X[1:m-1], Y[1:n]))| = k.
- Case 2.2: xm != yn AND zk != yn; then Z[1:k] = LCS(X[1:m], Y[1:n-1])).
  - Copy the proof of Case 2.1.
  - Assume W[1:w] is also a common sequence of (X[1:m], Y[1:n-1]), and w > k.
  - W[1:w] is also a common sequence of X[1:m] and Y[1:n], of length w > k.
  - Z is the LCS (no one can be longer than k) based on the definition.
  - Contradiction! No such W[1:w]. |LCS(X[1:m], Y[1:n-1]))| = k.
- Proof done.
- Fact: optimal substructure is a privilege. Does not hold for all.

Longest Common Substring

- Problem: 2 strings s,t -> find a longest string, a substring of both.
- Subproblems: dp[i,j]: length of the longest common substring of s[0…i] and t[0…j] ended with i, j; must include i and j, substring is continuous
- Relate: e.g., AxBxxC, AxByBxxC
  - Choose, s[i]==t[j]: dp[i-1,j-1]+1
  - Not choose, s[i]!=t[j]: 0
  - dp[i,j] = dp[i-1,j-1]+1 if s[i]==t[j]

    0 if s[i]!=t[j]
- Topological order: increasing i, j
- Base cases: dp[i,0] = 0, dp[0,j] = 0
- Original problem: backtrace max(dp[0,0], … dp[m,n])
- Time analysis: O(m*n)

Edit Distance

- Problem:
  - Given two strings s and t, return the minimum number of operations required to convert s to t. You have the following three operations permitted on a word: **Insert/Delete/Replace** a character
  - Input: word1 = "horse", word2 = "ros"
  - Output: 3
  - Explanation:
    - horse -> rorse (replace 'h' with 'r')
    - rorse -> rose (remove 'r')
    - rose -> ros (remove 'e')
- Subproblems: dp[i,j]: minimum edit distance of s[0…i] to t[0…j]
- Relate:
  - s[i]==t[j]: dp[i,j] = dp[i-1,j-1]
  - s[i]!=t[j]: dp[i,j] = min(dp[i,j-1], dp[i-1,j], dp[i-1,j-1])+1

```
                         insert      delete      replace
```

- Topological order: increasing i,j
- Base cases: dp[i,0] = i, dp[0,j] = j,
- Original problem: dp[m,n]
- Time analysis: O(m*n)

https://leetcode.com/problems/edit-distance/

Longest Increasing Subsequence

- Problem:
  - Given an integer array nums, return the length of the longest strictly increasing subsequence.
  - Input: nums = [410,9,2,5,3,7,101,18]
  - Output: 4
  - Explanation: The longest increasing subsequence is [2,3,7,101], therefore the length is 4.
- Subproblems: dp[i]:length of the longest strictly increasing subsequence **ended at i**
- Relate:
  - for j<i

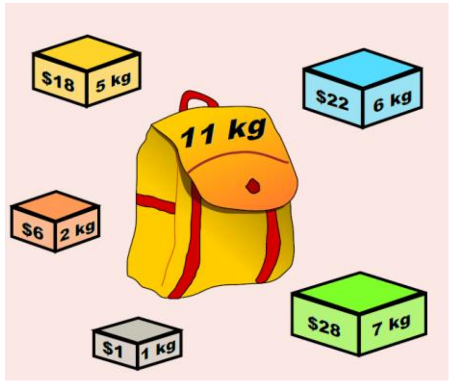
```
    if nums[i] > nums[j]                  // find the candidates
            dp[i] = max(dp[i],dp[j]+1) // update the record if needed
```

- Topological order: increasing i
- Base cases: dp[i] = 1
- Original problem: dp[n]
- Time analysis: O(n*n)

https://leetcode.com/problems/longest-increasing-subsequence/

0/1 Knapsack

- 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.
  - Value of a subset of items = sum of values of individual items.
  - Knapsack has weight limit of W. 0/1 Knapsack: take 0 or 1 for one item.



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

- Which algorithm solves knapsack problem?
  A. Greedy-by-value: repeatedly add item with maximum v[i].
  B. Greedy-by-weight: repeatedly add item with minimum w[i].
  C. Greedy-by-ratio: repeatedly add item with maximum ratio v[i]/w[i].
  D. None of the above.

0/1 Knapsack

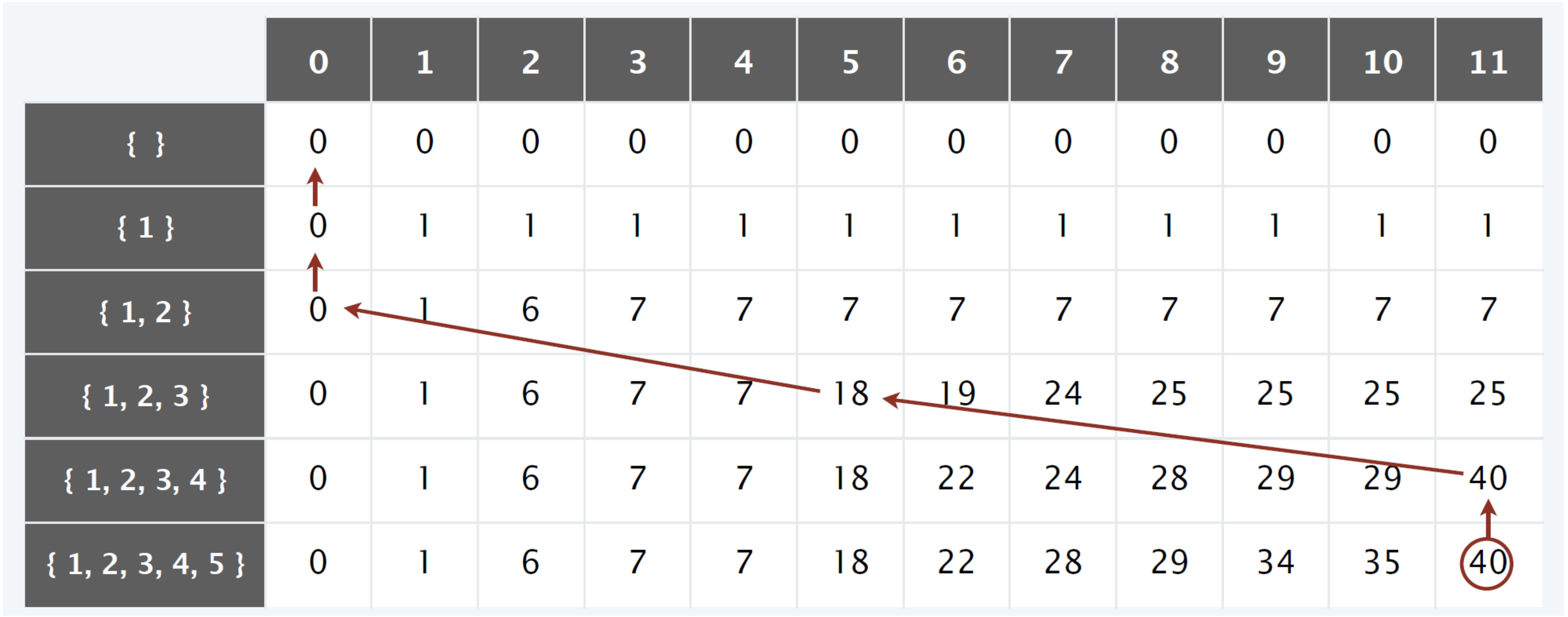
- Pack knapsack so as to maximize total value of items taken.
  - There are n items: item i provides value v[i] > 0 and weighs w[i] > 0.
  - Value of a subset of items = sum of values of individual items.
  - Knapsack has weight limit of W.
- Subproblems: dp[i,j]: max value from the **first** i items s.t j weight limit
- Relate:
  - Choose item i: the left weight limit decrease
    - max value from the first i-1 items s.t (j-w[i]) weight limit
    - dp[i,j] = dp[i-1, j-w[i]] + v[i]
  - Not choose item i: the left weight limit does not change
    - max value from the first i-1 items s.t j weight limit
    - dp[i,j] = dp[i-1, j]
  - dp[i,j] = max(dp[i-1, j], dp[i-1, j-w[i]] + v[i])   if (j-w[i]>0)

    = dp[i-1, j]   if (j-w[i]<0) weigh limit not enough
- Topological order: increasing i, increasing j
  - Is it ok to change the order?
- Base cases: dp[0,j] = 0 (dp[i,0] = 0)
- Original problem: dp[n,W], using an auxiliary matrix to record the trace
- Time analysis: O(n*W)

0/1 Knapsack

```
dp[i,j] = max(dp[i-1, j], dp[i-1, j-w[i]] + v[i]) if (j>w[i]>0)
        = dp[i-1, j] if (j-w[i]<0)
dp[0,j] = 0 (dp[i,0] = 0)
```

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



	0	1	2	3	4	5	6	7	8	9	10	11
{ }	0	0	0	0	0	0	0	0	0	0	0	0
{ 1 }	0	1	1	1	1	1	1	1	1	1	1	1
{ 1, 2 }	0	1	6	7	7	7	7	7	7	7	7	7
{ 1, 2, 3 }	0	1	6	7	7	18	19	24	25	25	25	25
{ 1, 2, 3, 4 }	0	1	6	7	7	18	22	24	28	29	29	40
{ 1, 2, 3, 4, 5 }	0	1	6	7	7	18	22	28	29	34	35	$\boxed{40}$

dp[j] = max(dp[j], dp[j-w[i]] + v[i]), increasing i (outer) then decreasing j (inner)

Target Sum

- Problem
  - You are given an integer array nums and an integer target.
  - You want to build an expression out of nums by adding one of the symbols '+' and '-' before each integer in nums and then concatenate all the integers.
  - For example, if nums = [2, 1], you can add a '+' before 2 and a '-' before 1 and concatenate them to build the expression "+2-1".
  - Return the number of different expressions that you can build, which evaluates to target.
  - Input: nums = [1,1,1,1,1], target = 3
  - Output: 5
  - Explanation: There are 5 ways to assign symbols to make the sum of nums be target 3.
  - -1 + 1 + 1 + 1 + 1 = 3
  - +1 - 1 + 1 + 1 + 1 = 3
  - +1 + 1 - 1 + 1 + 1 = 3
  - +1 + 1 + 1 - 1 + 1 = 3
  - +1 + 1 + 1 + 1 - 1 = 3

https://leetcode.com/problems/target-sum/

Target Sum

- Formulation: sum:sum of all items, p:sum of items with '+'
  - sum-p: sum of items with '–'
  - p-(sum-p) = target-> p = (target + sum)/2, let p be the new target
- Subproblems: dp[i,j]: number of ways to select the first i items to achieve a sum of j
- Relate:
  - Choose item i: dp[i,j] = dp[i-1, j-nums[i]]
  - Not choose item i: dp[i,j] = dp[i-1, j]

```
dp[i,j] = dp[i-1, j] + dp[i-1, j-nums[i]]     if (j-nums[i]>=0)
        = dp[i-1, j]                          if (j-nums[i]<0)
```

- Topological order: increasing i, increasing j
- Base cases: dp[0,0] = 1, the rest are initialized as 0
- Original problem: dp[n,target]
- Time analysis: O(n*target)

Unbounded Knapsack: Coin Change

- Unlimited supply of each item
- Problem:
  - You are given an integer array coins representing coins of different denominations and an integer amount representing a total amount of money.
  - Return the fewest number of coins that you need to make up that amount. If that amount of money cannot be made up by any combination of the coins, return -1.
  - You may assume that you have an infinite number of each kind of coin.
  - Input: coins = [1,2,5], amount = 11
  - Output: 3
  - Explanation: 11 = 5 + 5 + 1

https://leetcode.com/problems/coin-change/

Unbounded Knapsack: Coin Change

- Subproblems: dp[i,j]: minimum number of coins from first i coins to achieve an amount of j
- Relate:
  - Choose item i: dp[i,j] = dp[i, j-coins[i]] + 1, unlimited coins, still i options
  - Not choose item i: dp[i,j] = dp[i-1, j]

  ```
  dp[i,j] = min(dp[i-1, j], dp[i, j-coins[i]]+1)  if (j-coins[i]>=0)
          = dp[i-1, j]                           if (j-coins[i]<0)
  ```
- Topological order: increasing i, increasing j
- Base cases: dp[0,0] = 0, the rest are initialized as INT_MAX/2
- Original problem: dp[n,amount]
- Time analysis: O(n*target)

Typical Knapsack problems

- At most one capacity
  - Number of ways
  - Maximum value
- Equal to one capacity
  - Number of ways
  - Minimum/Maximum value
- At least one capacity
  - Number of ways
  - Minimum value

Summery

- Bottom-up vs Top-down
- Properties of DP
  - Hallmark 1:**Optimal substructure** An optimal solution to a problem (instance) contains optimal solutions to subproblems.
  - Hallmark 2:**Overlapping subproblems** A recursive solution contains a “small” number of distinct subproblems repeated many times.
  - Another property to consider: Markov property, given a specific state, its future development depends only on the current state and is independent of all previously experienced states.
- SRTBOT
  1. **Subproblem** definition,
     - Describe the meaning of a subproblem in words, in terms of parameters (or define state)
     - Often subsets of input: **prefixes**, suffixes, contiguous substrings of a sequence
     - Often record partial state: add subproblems by incrementing some auxiliary variables
  2. **Relate** subproblem solutions recursively (or transit state)
     - dp(i)= f(dp(j),...) or dp[i] = f(dp[j],…) for one or more j<i
     - Common idea: **choose or not choose, which one to choose**
  3. **Topological** order on subproblems (⇒subproblem DAG (Directed acyclic graph)), e.g., the iteration order/direction
  4. **Base** cases of relation, initialization and boundary cases
  5. **Original** problem solution via subproblem(s)
  6. **Time** analysis: # of subproblems * cost to relate (transit)