GB20602 - Programming Challenges

Week 4 - Dynamic Programming

Claus Aranha
caranha@cs.tsukuba.ac.jp

University of Tsukuba, Department of Computer Sciences

(last updated: May 6, 2023)

Version 2022.1

Part I – Introduction

Search Algorithms and Dynamic Programming

- Search Algorithms explore the search space of a problem in a systematic manner;
- Last week we studied three types of Search Algorithms:
  - Complete Search
  - Binary Search
  - Greedy Search
- This week, we study a new search algorithm: Dynamic Programming (DP).
- The key idea of DP is:

Store partial calculation in memory, to avoid duplicated work. (Related to Memoization)

Standard Dynamic Programming (DP) Algorithm

- Create a DP table:
  - The DP table stores partial calculation
  - The rows and columns of the table are the parameters of the calculation;
- Option 1: Top Down DP:
  - Write a recursive function to calculate the answer;
  - In the function, first test if the answer exist;
- Option 2: Bottom Up DP:
  - Loop through all the table (usually 2D loop);
  - In the loop, write each value of the table;
  - Return the answer in the end.
- Choosing the DP table is usually the hard part of the problem.

When do we use Dynamic Programming (DP)?

If a problem requires optimization or counting, then it "smells of DP"

- “Count the number of solutions...”
- “Find the minimum cost...”
- “Find the maximum length...”

What is the cost of running DP?

- Let the size of the DP table T be (a, b)
- Then let the cost of processing T[i,j] be $O(c)$
- Cost of DP: $O(abc)$ (can be pruned in some cases!)

You can prove the correctness of a DP algorithm using Proof by Induction.

Problem Example: Wedding Shopping

Problem Summary

We have to choose a set of items to buy, within a maximum budget M.

- There are C classes of items $(k_0, k_2, \ldots, k_{c-1})$;
- Each class k_i has N_i options;
- Each option j of class k_i has a cost $v_{i,j}$;

- You must buy 1 item from each class;
- Maximize the total cost, but do not exceed M;

- Limits: $M \leq 200$, $0 < C \leq 20$, $0 < N_i \leq 20$

QUIZ: How many possible combinations exist in the largest case?

Problem Example: Wedding Shopping

Solution Example

Sample case 1: $C = 3, N_i = \{3, 2, 4\}$

Class	0	1	2	3
k_0	6	4	8	
k_1	5	10		
k_2	1	5	3	5

If the budget is $M = 20$, the answer is 19. Three ways to reach this answer:

- $8(v_{0,2}) + 10(v_{1,1}) + 1(v_{2,0})$
- $6(v_{0,0}) + 10(v_{1,1}) + 3(v_{2,2})$
- $4(v_{0,1}) + 10(v_{1,1}) + 5(v_{2,1}$ or $v_{2,3})$

However, if the budget is $M = 9$, There is no solution for the problem. Because the minimum possible cost is 10 $(4(v_{0,1}) + 5(v_{1,0}) + 1(v_{2,0}))$

Problem Example: Wedding Shopping

Complete Search Solution

This is a Search problem: one solution is defined as "one choice from each class".

Unfortunately, a Greedy Algorithm will not work in this algorith. So first let's describe a full recursive search:

```
shop(m, g):                   // Recursive function. Returns the money used
                              // after start buying from category "g"
  if (m > M) return -1        // End case -- we spend more money than the budget.
  if (g == C) return m        // End case -- we bought all categories.
                              //             Return the total money used.
  for each i in Kc:
    totals[i] = shop(m + v[g][i], g+1) // try buying item i at category g.

  return max(totals)          // Return the value of the best item.

// First call of the recursive function: Start at category 0 with no money spent.
result = shop(0,0)
```

Problem Example: Wedding Shopping

Complete Search Solution – Time Limited Exceeded :-(

In the worst case, there are a total of 20^{20} possible combinations/choices.
So the complete search will be TLE...

Problem: Too many repeated subproblems

Class	0	1	2	3
0	6	4	8	12
1	4	6	6	2
2	1	5	1	5
3	2	4	6	2

Consider: How many times the program in the last slide will call "shop(10,2)?"

- shop(0,0) $\rightarrow$ shop(6,1) $\rightarrow$ shop(10,2)
- shop(0,0) $\rightarrow$ shop(4,1) $\rightarrow$ shop(10,2) x2
- shop(0,0) $\rightarrow$ shop(8,1) $\rightarrow$ shop(10,2)

Every time shop(10,2) is called, the return value is always the same.

Wedding Shopping using DP

When a problem has this characteristic (repeated sub-problems), it is a strong hint we should use DP.

First, we create a DP table using the parameters of the "shop(m, g)" function.

Remember: "shop(m, g)" always returns the same value.

How big is the table?

The table stores all possible calls of shop(m, g), so the table size is $|M| \times |C|$.

Remember that $0 \leq M \leq 200$ and $1 < C \leq 20$, so our table has $201 * 20 = 4020$ states.

That is a very small number! This algorithm will be FAST, compared to 20^{20}.

Wedding Shopping – the DP approach

How to fill the table?

There are two main approaches for filling the DP table:

- Top-down approach:
  Use the DP table as a "memory" table.
  Every time we call the function: If the result is in the table, use that result. If not, calculate and store in the table. Very common with "recursive functions".

- Bottom-up approach:
  First we complete the starting values of the table. Then we fill other values based on the starting values. Very common with "for loops".

Wedding Shopping – the DP approach

Using Top-down DP – very easy to program!

```
memset(table, -2, sizeof(table))  //-1 = "no result", -2 = "not visited yet"

shop(m, g):
  if (m > M) return -1                            // End States are the same;
  if (g == C) return m
  if (table[m][g] != -2) return table[m][g] // Check if the result is in memory

  for each i in Kc:                               // Calculate as before;
    totals[i] = shop(m + v[g][i], g+1)

  table[m][g] = max(totals)                       // Store new result in table;
  return table[m][g]

shop(0,0)                                         // That's the only change!
```

Wedding Shopping – The DP approach

Using bottom-up DP

Algorithm:

- Prepare a table with the problem states (same table as top-down);
- Choose the initial states of the table;
- Mark the initial states as "unprocessed";
- (Loop) For each unprocessed value, calculate its value, and add the new unprocessed values.

The main difficulties in bottom-up DP are:

- To find the initial states;
- To choose the processing function;

After that, it is just a big "for loop".

Wedding Shopping – Bottom-up DP

One possible solution

Example: M=10, $v_{0,x} = \{2, 4\}$, $v_{1,x} = \{4, 6\}$, $v_{2,x} = \{1, 3, 2, 1\}$

M ->	0	1	2	3	4	5	6	7	8	9	10
$i = 0$	X										
$i = 1$											
$i = 2$											
$i = 3$											

- Start state: We use no money, so mark $T(0, 0)$ as "reached (X)".
- Table Loop Loop i on all categories (0 to $C - 1$):
  - Loop j on all money: $j = 0 \rightarrow M$
  - If $T(i, j)$ is "reached (X)":
    - Loop f on all item costs (0 to $k_i - 1$):
    - Mark $T(i + 1, j + v_{i,f})$ as "reached (X)"
- Solution: The solution is the maximum column marked when $i = C$

Wedding Shopping – Bottom-up DP

One possible solution

Example: M=10, $v_{0,x} = \{2, 4\}$, $v_{1,x} = \{4, 6\}$, $v_{2,x} = \{1, 3, 2, 1\}$

M ->	0	1	2	3	4	5	6	7	8	9	10
$i = 0$	X										
$i = 1$			X		X						
$i = 2$											
$i = 3$											

- Start state: We use no money, so mark $T(0, 0)$ as "reached (X)".
- Table Loop Loop i on all categories (0 to $C - 1$):
  - Loop j on all money: $j = 0 \rightarrow M$
  - If $T(i, j)$ is "reached (X)":
    - Loop f on all item costs (0 to $k_i - 1$):
    - Mark $T(i + 1, j + v_{i,f})$ as "reached (X)"
- Solution: The solution is the maximum column marked when $i = C$

Wedding Shopping – Bottom-up DP

One possible solution

Example: M=10, $v_{0,x} = \{2, 4\}$, $v_{1,x} = \{4, 6\}$, $v_{2,x} = \{1, 3, 2, 1\}$

M ->	0	1	2	3	4	5	6	7	8	9	10
$i = 0$	X										
$i = 1$			X		X						
$i = 2$							X		X		X
$i = 3$											

- Start state: We use no money, so mark $T(0, 0)$ as "reached (X)".
- Table Loop Loop i on all categories (0 to $C - 1$):
  - Loop j on all money: $j = 0 \rightarrow M$
  - If $T(i, j)$ is "reached (X)":
    - Loop f on all item costs (0 to $k_i - 1$):
    - Mark $T(i + 1, j + v_{i,f})$ as "reached (X)"
- Solution: The solution is the maximum column marked when $i = C$

Wedding Shopping – Bottom-up DP

One possible solution

Example: M=10, $v_{0,x} = \{2, 4\}$, $v_{1,x} = \{4, 6\}$, $v_{2,x} = \{1, 3, 2, 1\}$

M ->	0	1	2	3	4	5	6	7	8	9	10
$i = 0$	X										
$i = 1$			X		X						
$i = 2$							X		X		X
$i = 3$								X	X	X	X

- Start state: We use no money, so mark $T(0, 0)$ as "reached (X)".
- Table Loop Loop i on all categories (0 to $C - 1$):
  - Loop j on all money: $j = 0 \rightarrow M$
  - If $T(i, j)$ is "reached (X)":
    - Loop f on all item costs (0 to $k_i - 1$):
    - Mark $T(i + 1, j + v_{i,f})$ as "reached (X)"
- Solution: The solution is the maximum column marked when $i = C$

Wedding Shopping – Bottom-up DP

Example: M=10, $K_0 = \{2, 4\}$, $K_1 = \{4, 6\}$, $K_2 = \{1, 3, 2, 1\}$

M ->	0	1	2	3	4	5	6	7	8	9	10
$i = 0$	X										
$i = 1$			X		X						
$i = 2$							X		X		X
$i = 3$								X	X	X	X

```
memset(table,0,sizeof(table))
table[0][0] = 1

for i in (0 to C-1)
  for j in (0 to M):
    if table[i][j] == 1:
      for f in (0 to K[i]-1):
        table[i + 1][j + cost[i][f]] = 1 // Don't forget out of bounds check!
```

DP: Should you use Top-down or Bottom-up?

Top-Down

Pros: Easy to implement, just add memory to a recursive search. Only computes the visited states of the DP table.

Cons: Overhead of recursive function. Hard to reduce the size of the DP table.

Bottom-Up

Pros: Faster if you have to visit most of the table. It is possible to save memory by discarding old rows.

Cons: Harder to think the algorithm. If the DP table is sparse, the loop will visit every state.

Finding the Decision Set with DP

This example program only returns the total money used.

Sometimes we also need to output the optimal solution. How do we do that?

It is not very hard. You need TWO tables:

- Table 1: The DP table (same as before);
- Table 2: The "Parent" table, which indicates the previous choice.

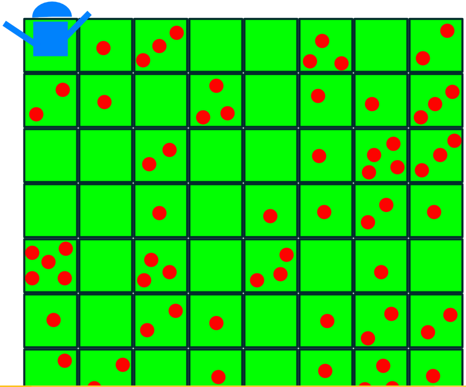
The next example will show the use of the "Parent" table.

When filling the parent table, be careful about the rules for tie breaking!
(Lexographical order, smallest solution, etc).

Example 2: Apple Field

A farmer has an apple field, and a robot to collect the apples. However, the robot can only move right or down. The robot starts at position $(0, 0)$, and ends at (n, n).

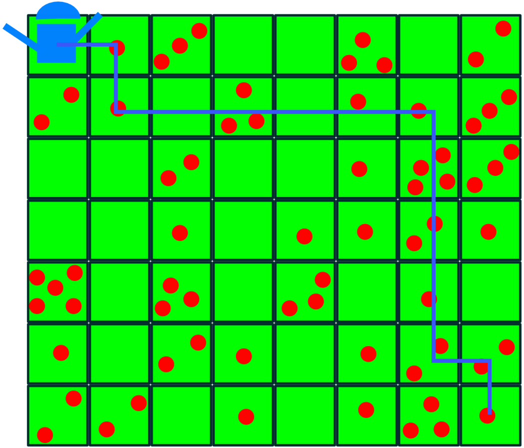
You know how many apples are in each cell (A[i][k]). What is the path with maximum apples?



Example 2: Apple Field

One Possible Solution (Not maximum)

L, D, L, L, L, L, L, L, D, D, D, D, L, D



Example 2: Apple Field

Complete Search

How many different paths are possible?

- A path has n steps right (0), and n steps down (1), in any order.
- A path is a string with size $2n$, n "0"s, and n "1"s.
- Permutation of $2n$ with n "0"s and n "1"s: $\binom{2n}{n} = \frac{(2n)!}{n!n!}$
  - Too big for full search!

Like in the "Wedding Shopping" problem, we have repeating subproblems:

For example, the optimal path from (x, y) to (n, n) is always the same, regardless of the path from $(0, 0)$ to (x, y). So let's try DP!

Example 2: Apple Field

Bottom-up DP

- DP table and Parent table:
  - The DP table is a $n+1 \times n+1$ table. At each position (x, y), we store the maximum number of apples from $(0,0) \rightarrow (x, y)$.
  - The Parent table is a $n+1 \times n+1$ table. At each position, we store the back step (up or left) of the optimal path to (x, y).
- Initial Condition: (DP table only)
  - To avoid special treatment of the first row and first column, we include a "boundary" at the top and left sides of the table. Every cell at the boundary has "0" apples
- Transition:
  - We double loop over the DP table (row $\rightarrow$ column). For every cell (x, y):
    $DP[x][y] = A[x][y] + \max(DP[x-1][y], DP[x][y-1])$
    $\text{Parent}[x][y] = \text{if } DP[x-1][y] > DP[x][y-1] : \text{"left"}, \text{ else "top"}$

Example 2: Apple Field

Pseudocode

```
int A[n+1][n+1];      // Input Data. Index is (1, 1) to (n, n)

int DP[n+1][n+1];                            // DP Table
DP[0][0..n+1] and DP[0..n+1][0] = 0;    // Initial states;

int parent[n+1][n+1];                        // Parent Table;

for (int i = 1; i < n+1; i++) {
  for (int j = 1; j < n+1; j++) {
    DP[i][j] = A[i][j] + max(DP[i][j-1], DP[i-1][j]);  // Update DP
    if (DP[i][j-1] > DP[i-1][j]):                      // Update Parent
       parent[i][j] = "left";
    else:
       parent[i][j] = "up";
  }
}
```

Example 2: Apple Field

Simulating the algorithm

```
DP[i][j] = apple[i][j] + max(DP[i][j-1], DP[i-1][j]);
if (DP[i][j-1] > DP[i-1][j]):
   parent[i][j] = "left";
else:
   parent[i][j] = "up";
```

Input Table

	1		2			6	
	2	2			2		4
		3		1	1		

DP Table

0	0	0	0	0	0	0	0
0							
0							
0							

Parent Table

Example 2: Apple Field

Simulating the algorithm

```
DP[i][j] = apple[i][j] + max(DP[i][j-1], DP[i-1][j]);
if (DP[i][j-1] > DP[i-1][j]):
   parent[i][j] = "left";
else:
   parent[i][j] = "up";
```

Input Table

	1		2			6	
	2	2			2		4
		3		1	1		

DP Table

0	0	0	0	0	0	0	0
0	1						
0							
0							

Parent Table

	U						

Example 2: Apple Field

Simulating the algorithm

```
DP[i][j] = apple[i][j] + max(DP[i][j-1], DP[i-1][j]);
if (DP[i][j-1] > DP[i-1][j]):
   parent[i][j] = "left";
else:
   parent[i][j] = "up";
```

Input Table

	1		2			6	
	2	2			2		4
		3		1	1		

DP Table

0	0	0	0	0	0	0	0
0	1	1					
0							
0							

Parent Table

	U	L					

Example 2: Apple Field

Simulating the algorithm

```
DP[i][j] = apple[i][j] + max(DP[i][j-1], DP[i-1][j]);
if (DP[i][j-1] > DP[i-1][j]):
   parent[i][j] = "left";
else:
   parent[i][j] = "up";
```

Input Table

	1		2			6	
	2	2			2		4
		3		1	1		

DP Table

0	0	0	0	0	0	0	0
0	1	1	3				
0							
0							

Parent Table

	U	L	L				

Example 2: Apple Field

Simulating the algorithm

```
DP[i][j] = apple[i][j] + max(DP[i][j-1], DP[i-1][j]);
if (DP[i][j-1] > DP[i-1][j]):
   parent[i][j] = "left";
else:
   parent[i][j] = "up";
```

Input Table

	1		2			6	
	2	2			2		4
		3		1	1		

DP Table

0	0	0	0	0	0	0	0
0	1	1	3	3	3		
0							
0							

Parent Table

	U	L	L	L	L		

Example 2: Apple Field

Simulating the algorithm

```
DP[i][j] = apple[i][j] + max(DP[i][j-1], DP[i-1][j]);
if (DP[i][j-1] > DP[i-1][j]):
   parent[i][j] = "left";
else:
   parent[i][j] = "up";
```

Input Table

	1		2			6	
	2	2			2		4
		3		1	1		

DP Table

0	0	0	0	0	0	0	0
0	1	1	3	3	3	9	9
0							
0							

Parent Table

	U	L	L	L	L	L	L

Example 2: Apple Field

Simulating the algorithm

```
DP[i][j] = apple[i][j] + max(DP[i][j-1], DP[i-1][j]);
if (DP[i][j-1] > DP[i-1][j]):
   parent[i][j] = "left";
else:
   parent[i][j] = "up";
```

Input Table

	1		2			6	
	2	2			2		4
		3		1	1		

DP Table

0	0	0	0	0	0	0	0
0	1	1	3	3	3	9	9
0	3						
0							

Parent Table

	U	L	L	L	L	L	L
	U						

Example 2: Apple Field

Simulating the algorithm

```
DP[i][j] = apple[i][j] + max(DP[i][j-1], DP[i-1][j]);
if (DP[i][j-1] > DP[i-1][j]):
   parent[i][j] = "left";
else:
   parent[i][j] = "up";
```

Input Table

	1		2			6	
	2	2			2		4
		3		1	1		

DP Table

0	0	0	0	0	0	0	0
0	1	1	3	3	3	9	9
0	3	5					
0							

Parent Table

	U	L	L	L	L	L	L
	U	L					

Example 2: Apple Field

Simulating the algorithm

```
DP[i][j] = apple[i][j] + max(DP[i][j-1], DP[i-1][j]);
if (DP[i][j-1] > DP[i-1][j]):
   parent[i][j] = "left";
else:
   parent[i][j] = "up";
```

Input Table

	1		2			6	
	2	2			2		4
		3		1	1		

DP Table

0	0	0	0	0	0	0	0
0	1	1	3	3	3	9	9
0	3	5	5	5			
0							

Parent Table

	U	L	L	L	L	L	L
	U	L	L	L			

Example 2: Apple Field

Simulating the algorithm

```
DP[i][j] = apple[i][j] + max(DP[i][j-1], DP[i-1][j]);
if (DP[i][j-1] > DP[i-1][j]):
   parent[i][j] = "left";
else:
   parent[i][j] = "up";
```

Input Table

	1		2			6	
	2	2			2		4
		3		1	1		

DP Table

0	0	0	0	0	0	0	0
0	1	1	3	3	3	9	9
0	3	5	5	5	7		
0							

Parent Table

	U	L	L	L	L	L	L
	U	L	L	L	L		

Example 2: Apple Field

Simulating the algorithm

```
DP[i][j] = apple[i][j] + max(DP[i][j-1], DP[i-1][j]);
if (DP[i][j-1] > DP[i-1][j]):
   parent[i][j] = "left";
else:
   parent[i][j] = "up";
```

Input Table

	1		2			6	
	2	2			2		4
		3		1	1		

DP Table

0	0	0	0	0	0	0	0
0	1	1	3	3	3	9	9
0	3	5	5	5	7	9	
0							

Parent Table

	U	L	L	L	L	L	L
	U	L	L	L	L	U	

Example 2: Apple Field

Simulating the algorithm

```
DP[i][j] = apple[i][j] + max(DP[i][j-1], DP[i-1][j]);
if (DP[i][j-1] > DP[i-1][j]):
   parent[i][j] = "left";
else:
   parent[i][j] = "up";
```

Input Table

	1		2			6	
	2	2			2		4
		3		1	1		

DP Table

0	0	0	0	0	0	0	0
0	1	1	3	3	3	9	9
0	3	5	5	5	7	9	13
0							

Parent Table

	U	L	L	L	L	L	L
	U	L	L	L	L	U	U

Example 2: Apple Field

Simulating the algorithm

```
DP[i][j] = apple[i][j] + max(DP[i][j-1], DP[i-1][j]);
if (DP[i][j-1] > DP[i-1][j]):
   parent[i][j] = "left";
else:
   parent[i][j] = "up";
```

Input Table

	1		2			6	
	2	2			2		4
		3		1	1		

DP Table

0	0	0	0	0	0	0	0
0	1	1	3	3	3	9	9
0	3	5	5	5	7	9	13
0	3	8	8	9	10	10	13

Parent Table

	U	L	L	L	L	L	L
	U	L	L	L	L	U	U
	U	U	L	L	L	L	U

Part II – Classical DP Problems

Classical DP Problems

There are some classical problems that have well known DP solutions:

- Max sum;
- Max sum 2D;
- Longest Increasing Subsequence;
- Knapsack Problem;
- Coin Change;

We will show some examples from each category so you can have a better understanding of the DP philosophy.

After each problem is explained, try to find the DP table, and the transition function.

The 1D Range Sum Problem

Consider an array A containing N integers. We want to find the indexes $i, j, (0 \leq i < j \leq N - 1)$ that maximize the sum from A_i to A_j $(\sum_{k=i}^{j} A_k)$.

Example:

```
Array A = 1, -3, 20, -2, -5, 10, 5, -4, 6, 47, -30, -3
  Total = 42
RangeSum=        20, -2, -5, 10, 5, -4, 6, 47
  Total = 77
```

How do you solve this problem?

The 1D Range Sum Problem

Complete Search

Calculate the range sum for every possible pair (i, j).

```
int minindex, maxindex;
int maxsum = 0;
for (int i = 0; i < n; i++)             // Loop 1
   for (int j = 0; i < n; j++)          // Loop 2
      int sum = 0;
      for (int k = i; k < j+1; k++)     // Loop 3
         sum += k;
      if sum > maxsum:
         maxsum = sum;
         minindex = i; maxindex = j;
```

Because of three loops, this approach is $O(n^3)$. For large values of N ($N > 10.000$), this is a bad idea.

The 1D Range Sum Problem
DP Sum Table

Note that sum(i,j) = sum(0,j) - sum(0,i-1).

Using this fact, we can create a sum table (ST) to calculate the result faster:

Using Sum Table – $O(n^2)$

```
int[] ST; int maxsum = 0; int sum_s = 0; int sum_e = 0; ST[0] = 0;

for (int i = 1; i < N+1; i++) { ST[i] = ST[i-1] + A[i]; } // preprocessing;

for (int i = 1; i < N+1; i++)
  for (int j = i; j < N+1; j++)
    if (ST[j] - ST[i-1] > maxsum) {
      maxsum = ST[j] - ST[i-1];
      sum_s = i; sum_e = j;
    }
```

The 1D Range Sum Problem
DP Sum Table Simulation

Let's visualize how the DP sum table transforms the problem:

```
i  =       1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12
A  =       1, -3, 20, -2, -5, 10,  5, -4,  6, 47,-30, -3
ST = [0], 1, -2, 18, 16, 11, 21, 26, 22, 28, 75, 45, 42

i, j  | ST[j] - ST[i-1] | Total Sum
===================================
1, 12 | 42    - 0       | 42
3, 10 | 75    - (-2)    | 77
6, 8  | 22    - 11      | 11
===================================
```

Can we do even better?

The 1D Range Sum Problem

Kadane's Greedy Algorithm – $O(n)$ mix of Sum Table and Greedy Approach

```
A[] = { 4, -5, 4, -3, 4, 4, -4, 4, -5}; // Example
int sum = 0, ans = 0;
for (i in 0:n):
   sum += A[i], ans = max(ans, sum)      // Add to running total
   if (sum < 0) sum = 0;                 // If total is negative
                                         // reset the sum;
```

- Basic idea: it is always better to increase the sum, unless a very large negative sum appears.
- In that case, it is better to start from zero after the negative sum.

```
A   :  4 | -5 | 4 -3  4  4 -4  4 | -5
Sum:   4 |  0 | 4  1  5  9  5  9 |  4
ans:   4 |  4 | 4  4  5  9  9  9 |  9
```

Maximum Range Sum – Now in 2D!

Problem Summary

Given an array of positive and negative numbers, find the subarray with maximum sum.

0	-2	-7	0
9	2	-6	2
-4	1	-4	1
-1	8	0	-2

This is the same problem as the previous one, but the second dimention adds some interesting complications.

QUIZ:

- What is the cost of a complete search in this case?
- How would you write a DP (table and loop)?

Maximum Range Sum 2D

Complete Search

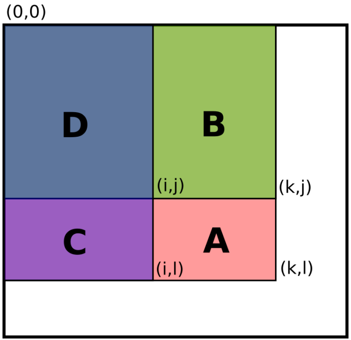
The complete search approach needs 6 loops (2 for horizontal axis, 2 for vertical axis, 2 for calculating the sum). So the total complexity is O(n^6).

```
minvalue = -MIN_INT
for i in (0:n):
   for j in (0:n):
      for k in (i:n):
         for l in (j:n):
         sum = 0
         for a in (i:k):
            for b in (j:l):
               sum += A[a,b]
         if sum > minvalue:
            minvalue = sum
```

Maximum Range Sum 2D

Using the Sum Table

We can use the Sum Table idea from 1D, but be careful about the Principle of Inclusion-Exclusion. We subtract the partial sum of two axis, and add back the intersection of that sum.



$$A = ABCD - BD - CD + D$$

Maximum Range Sum 2D

2D Sum Table Pseudocode

```
for i in (0:n):                        // Precalculation: Creating ST
   for j in (0:n):
       ST[i][j] = A[i][j]                              // A[i][j] is the input
       if (i > 0) ST[i][j] += ST[i-1][j]
       if (j > 0) ST[i][j] += ST[i][j-1]
       if (i > 0 && j > 0) ST[i][j] -= ST[i-1][j-1]  // Avoid double count

for i,j in (0:n)(0:n):
   for k,l in (i:n)(j:n):
      sum = ST[k][l]                          // Total Sum (0,0)->(k,l)
      if (i > 0) sum -= ST[i-1][l];           // Remove (0,0)->(i-1,l)
      if (j > 0) sum -= ST[k][j-1];           // Remove (0,0)->(k,j-1)
      if (i > 0 && j > 0) sum += A[i-1][j-1] // Add back double remove
      maxsum = max(sum,maxsum)
```

Problem 3: Longest Increasing Subsequence

Problem Definition

Given a sequence A of integers, find the longest subsequence $S \in A$ where $S_i < S_{i+1} < S_{i+2} < \ldots$.

Example:

```
A   = [-7, 10, 9, 2, 3, 8, 8, 1]
S_1 = [-7,         2, 3, 8]          // size 4 -- LIS
S_2 = [-7,      9]                   // size 2
```

Note that because the subsequence is not contiguous, this problem is more difficult than Range Sum.

QUIZ: What is the Complete Search and DP approach (Table and Loop) for this problem?

Complete Search for LIS

As other "find the subset" problems, the complete search of LIS can be done by testing all binary strings of size "n". This costs $O(2^n)$.

```
// Complete Subset Search using bitmasks
vector<int> S_max; int max_len = 0;// Final Result

for (int i = 0; i < (1<<n); i++) {                    // Loop all bitstrings
  vector<int> S; int min = -99999; int len = 0;
  for (int j = 0; j < n; j++) {                       // Creat subset from bitstring
    if ((1<<j)&i) {                                   // Add j to subset
      if (A[j] > min) {                               // Test if subset is increasing
        S.push_back(A[j]);
        min = A[j]; len ++;
      } else { break; }                               // Subset not increasing
  } }
  if (len > max_len) { max_len = len; S_max = S; }// Found a longer subset
} }
```

DP for Longest Increasing Subsequence

As usual, to prepare a DP we decide the Table and Transition.

Transition

Loop each element A[i], and choose:

- Check A[0] to A[i-1], see if A[i] can enter an existing LIS
- If not, A[i] is the beginning of a new LIS

Tables

- A[i]: Has the value of the number;
- Parent[i]: Has the index of the previous number in the LIS;
- LIS[i]: Size of the longest LIS that this number is a member;

DP for Longest Increasing Subsequence

Example

```
A      = [  0, 10, 9,  0, 3, 8, 8, 1 ]
parent = [ -1,  0, 0, -1, 3, 4, 4, 3 ]
LIS    = [  1,  2, 2,  1, 2, 3, 3, 2 ]
```

Pseudocode (O(n^2))

```
LIS[0:n] = 1
parent[0:n] = -1
for i in (1 to n):
   for j in (0 to i): // Try to add to longest LIS
      if (LIS[j] >= LIS[i]) && (A[j] < A[i]):
         LIS[i] = LIS[j] + 1
         parent[i] = j
```

There is a faster $O(n \log k)$ approach that uses greedy and binary search.

Classic DP: The 0-1 Knapsack Problem

In the 0-1 Knapsack problem (also known as "subset sum"), there is a set A of items with size S and value V.

You have to select a subset $X \subseteq A$ where the sum of sizes $\leq M$, and the sum of values is maximum.

```
Input:
  A<S,V> = [ (10, 100), (4, 70), (6, 50), (12, 10)]
  M = 12

Solution:
  [ (4,70), (6,50) ]
```

QUIZ: What is the complete search and the DP (Table, Transition)?
Hint: This problem is similar to the "Wedding Problem".

0-1 Knapsack – Complete Search

The solution to the complete search is to test all subsets of A. This approach, as you know, takes $O(2^n)$.

This time, instead of a binary string, we will test all combinations using recursion.

Complete Search Recursive Solution

Recursive function: value(id,size), where id is the item we want to add, and size is the size remaining after we add id in the backpack.

```
value(id,size):
   if (size < 0): return 0   # bag is full
   if (id == n):  return 0   # checked all items
   # either add the item, or do not add the item
   return max(value(id+1,size),
              V[id] + value(id+1, size - S[id]))
```

0-1 Knapsack – Top-down DP

From the recursive function, it is very easy to use a DP table as memory for value(id, size).

Be careful: The DP table size (and the execution time) is $|A| \times M$. If M is too big ($>> 10^6$), you might get TLE or MLE.

```
A<S,V> = [ (10, 100), (4, 70), (6, 50), (12, 10)]
M = 12

value(i,size):
```

	-	0	1	2	3	4	5	6	7	8	9	10	11	12
0														
1														
2														
3														
4														

Classical DP – The Coin Change Problem (CC)

Problem Summary

You are given a target value V, and a set A of coin sizes. You have to find the smallest sequence of coins (with repetition) that adds to V.

Example:

```
V = 7
A = {1, 3, 4, 5}
  S_0 = { 1, 1, 1, 1, 3}
  S_1 = { 5, 1, 1}
  S_2 = { 3, 3, 1}
  S_3 = { 4, 3}
```

The best solution is S_3.

QUIZ:

- How do you solve this by complete search?

Complete Search for Coin Change

We can build a recursive search using the following recurrence on the number of coins N necessary for a given value V:

$$N(V) = 1 + N(V - \text{ size of coin})$$

Recursive Complete Search

```
coins(V):                          // Number of coins for value V:
   if V == 0: return 0             // 0 coins for value 0
   if V < 0:  return MAX_INT       // Can't satisfy for this value
   min = INF                       // Minimum number of coins
   for i in (coins):               // Test each coin
      t = 1 + coins(value - A[i])
      if (t < min): min = t
   return t
```

DP for Coin Change

- Implementing a Top-down DP should be easy for you now;
- Let's make a Bottom-UP DP for practice.
- The DP table is [coin-type][value], DP[0][v] = 0; C(i) is coin i

Bottom-UP DP

```
DP[c][v] = -1 // Set all locations as "can't reach"
DP[c][0] = 0  // 0 coins when value is 0

for i = 1 to c:   // loop over coin types
  for j = 1 to v: // loop over coin values
    above = DP[i-1][j]       // Test not using C(i)
    left = DP[i][j-C(i)]+1   // Test using C(i)
                             // Remember to test for boundaries!
    DP[i][j] = min(above, left) // update cell
                                // remember to ignore "-1"s!

return DP[c][v]
```

DP for Coin Change

Simulation

```
V = 7
A = {1, 3, 4, 5}
```

	0	1	2	3	4	5	6	7
1 (1)	0	1	2	3	4	5	6	7
2 (3)	0							
3 (4)	0							
4 (5)	0							

It is interesting to note that the calculation of row i depends only on row $i-1$. Using this information, you can implement the program with a much smaller table.

DP for Coin Change

Simulation

```
V = 7
A = {1, 3, 4, 5}
```

	0	1	2	3	4	5	6	7
1 (1)	0	1	2	3	4	5	6	7
2 (3)	0	1	2					
3 (4)	0							
4 (5)	0							

It is interesting to note that the calculation of row i depends only on row $i-1$. Using this information, you can implement the program with a much smaller table.

DP for Coin Change

Simulation

```
V = 7
A = {1, 3, 4, 5}
```

	0	1	2	3	4	5	6	7
1 (1)	0	1	2	3	4	5	6	7
2 (3)	0	1	2	1	2	3		
3 (4)	0							
4 (5)	0							

It is interesting to note that the calculation of row i depends only on row $i-1$. Using this information, you can implement the program with a much smaller table.

DP for Coin Change

Simulation

```
V = 7
A = {1, 3, 4, 5}
```

	0	1	2	3	4	5	6	7
1 (1)	0	1	2	3	4	5	6	7
2 (3)	0	1	2	1	2	3	2	3
3 (4)	0							
4 (5)	0							

It is interesting to note that the calculation of row i depends only on row $i-1$. Using this information, you can implement the program with a much smaller table.

DP for Coin Change

Simulation

```
V = 7
A = {1, 3, 4, 5}
```

	0	1	2	3	4	5	6	7
1 (1)	0	1	2	3	4	5	6	7
2 (3)	0	1	2	1	2	3	2	3
3 (4)	0	1	2	1				
4 (5)	0							

It is interesting to note that the calculation of row i depends only on row $i-1$. Using this information, you can implement the program with a much smaller table.

DP for Coin Change

Simulation

```
V = 7
A = {1, 3, 4, 5}
```

	0	1	2	3	4	5	6	7
1 (1)	0	1	2	3	4	5	6	7
2 (3)	0	1	2	1	2	3	2	3
3 (4)	0	1	2	1	1	2		
4 (5)	0							

It is interesting to note that the calculation of row i depends only on row $i-1$. Using this information, you can implement the program with a much smaller table.

DP for Coin Change

Simulation

```
V = 7
A = {1, 3, 4, 5}
```

	0	1	2	3	4	5	6	7
1 (1)	0	1	2	3	4	5	6	7
2 (3)	0	1	2	1	2	3	2	3
3 (4)	0	1	2	1	1	2	2	2
4 (5)	0							

It is interesting to note that the calculation of row i depends only on row $i - 1$. Using this information, you can implement the program with a much smaller table.

DP for Coin Change

Simulation

```
V = 7
A = {1, 3, 4, 5}
```

	0	1	2	3	4	5	6	7
1 (1)	0	1	2	3	4	5	6	7
2 (3)	0	1	2	1	2	3	2	3
3 (4)	0	1	2	1	1	2	2	2
4 (5)	0					1		

It is interesting to note that the calculation of row i depends only on row $i-1$. Using this information, you can implement the program with a much smaller table.

DP for Coin Change

Simulation

```
V = 7
A = {1, 3, 4, 5}
```

	0	1	2	3	4	5	6	7
1 (1)	0	1	2	3	4	5	6	7
2 (3)	0	1	2	1	2	3	2	3
3 (4)	0	1	2	1	1	2	2	2
4 (5)	0	1	2	1	1	1	2	2

It is interesting to note that the calculation of row i depends only on row $i-1$. Using this information, you can implement the program with a much smaller table.

About these Slides

Individual images in some slides might have been made by other authors. Please see the following pages for details.

Image Credits I

[Page 6] Wedding Dress Image CC-By-2.0 by
`https://www.flickr.com/photos/vancouver125/5634967507`