Programming Challenges (GB21802)

Week 4 - Dynamic Programming

Claus Aranha
caranha@cs.tsukuba.ac.jp

University of Tsukuba, Department of Computer Sciences

2020/5/19

(last updated: May 19, 2020)

Version 2020.1

Review of Last Week

- **Search Algorithms** are defined by the systematic checking of the **Search Space** of a problem;
- We studied three types of Search Algorithms:
  - Complete Search
  - Divide and Conquer
  - Greedy Search

This week we introduce a fourth search algorithm: **Dynamic Programming.**

Dynamic Programming is arguably the most used algorithm in programming competitions. It's basic idea is that we can exchange "computation time" for "memory".

What is Dynamic Programming (DP)?

DP is a Search Algorithm based on the idea of **building partial solutions** and storing them on memory.

Basic Idea of DP

- Create a **DP table** where the axis are the parameters of a recurrent function that generates the solution to the problem;
- Fill in the table with the starting condition of the recurrence;
- Fill in the rest of the table recursively or or using an iterator, and find the answer;

What is Dynamic Programming (DP)?

Characteristics

When is DP useful?

In programming contests, a problem that requires optimization or counting "smells of DP"

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

What is the running cost of DP?

The DP algorithm evaluates each element of the DP table.

Therefore, DP costs "size of DP table $\times$ cost of one element.

The proof of correctness of DP algorithm is **Proof by Induction**.

Problem Example: Wedding Shopping – UVA 11450

Best way to understand DP is to do a lot of examples!

Problem summary:

- There are C classes of items;
- Each class has K_c options;
- Each option $K_{c,i}$ has a different cost;

- Buy one of each option from each Class;
- Maximize the total cost;
- **However**, you cannot exceed the budget M

- $M \leq 200$, $1 < C \leq 20$, $0 < K_c \leq 20$

The total number of possible buying combinations is 20^{20}

Problem Example: Wedding Shopping – UVA 11450

Solution Example

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

Class	1	2	3	4
K_0	6	4	8	
K_1	5	10		
K_2	1	5	3	5

If budget $M = 20$, the answer is 19. You can reach this answer by buying:

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

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

Problem Example: Wedding Shopping – UVA 11450

Complete Search Solution

This is a **Search problem**: our solution selects one item from each class. Unfortunately, the *Greedy* approach does not work here.

Recursive Complete Search

- Function $\text{shop}(m, g)$ finds the best item to buy in K_g after spending m, and return the remaining budget;
- It tests every $K_{g,i}$ by first calling $\text{shop}(m + K_{g,i}, g + 1)$, recursively;
- End condition 1: $m > M$, return -1
- End condition 2: $\text{shop}(m, |C|)$, return m
- The initial call: $\text{shop}(0, 0)$.

```
shop(m,g):
  if (m > M) return -1                          // no money
  if (g == C) return m                          // one solution
  return (max(shop(m+price[g][i], g+1)), i in Kc)// choose max
```

Wedding Shopping (11450) - Complete search

Time Limited Exceeded

The **complete search** solution works, but because we have a total of 20^{20} possible choices, it does not finish in time;

Problem: Too many overlapping subproblems

Sample case

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

How many times the program does the function calls *shop(10,2)*?

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

Is it possible to reduce the number of identical calls?

Wedding Shopping – the DP approach

When a problem has this characteristic (**repeated sub-structures**), it is a strong hint that DP is a good solution.

First, we create a **DP table** using the parameters of the "shop" recurrent function.

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.

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 look-up (memory) table. Every time you visit a state, save the result and do not recalculate. Very common with **recursion**.

- Bottom-up approach:
  First complete the starting values of the table, then progressively fill the other states based on the starting ones. Very common with **for loops**.

Wedding Shopping – the DP approach

Top-down DP

```
memset(table, -2, sizeof(table))                  //-2 = "not visited"

shop(m,g):
  if (m > M) return -1                            // no money
  if (g == C) return m                            // one solution
  if (table[m][g] != -2) return table[m][g]      // table lookup

  table[m][g] = (max(shop(m+price[g][i],[g+1])), i in Kc)
  return table[m,g]                               // calculate&return
```

Top Down DP Implementation is very easy

Just add a table, and every time you enter your recursive function, test if the parameters exist in the table.

Make sure that the result of your function is **always the same** when the DP parameters are the same! (usually not a problem)

Wedding Shopping – bottom-up DP

Algorithm:

- Prepare a table with the problem states (same as top-down);
- Add the the initial values in the table as "unprocessed";
- **(Loop)** For each unprocessed value, process it, and add the new unprocessed values.

The main difficulty in bottom-up DP is to find the base cases and the transition function. After that, it is just a big for loop.

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
$s = 0$	X										
$s = 1$											
$s = 2$											
$s = 3$											

- **Start state**: We use no money, so mark $T(0, 0)$
- **Transition** $(s \rightarrow s + 1)$:
  - Loop: $i = 0 \rightarrow m$
  - If $T(s, i)$ is marked:
    - Loop: $j = 0 \rightarrow |K_c|$
    - Mark $T(s + 1, i + K_{c,j})$
- **Solution**: Maximum marked column of the last row.
- **Note**: Other solutions are possible!

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
$s = 0$	X										
$s = 1$			X		X						
$s = 2$											
$s = 3$											

- **Start state**: We use no money, so mark $T(0, 0)$
- **Transition** $(s \to s + 1)$:
  - Loop: $i = 0 \to m$
  - If $T(s, i)$ is marked:
    - Loop: $j = 0 \to |K_c|$
    - Mark $T(s + 1, i + K_{c,j})$
- **Solution**: Maximum marked column of the last row.
- **Note**: Other solutions are possible!

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
$s = 0$	X										
$s = 1$			X		X						
$s = 2$							X		X		X
$s = 3$											

- **Start state**: We use no money, so mark $T(0, 0)$
- **Transition** $(s \rightarrow s + 1)$:
  - Loop: $i = 0 \rightarrow m$
  - If $T(s, i)$ is marked:
    - Loop: $j = 0 \rightarrow |K_c|$
    - Mark $T(s + 1, i + K_{c,j})$
- **Solution**: Maximum marked column of the last row.
- **Note**: Other solutions are possible!

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
$s = 0$	X										
$s = 1$			X		X						
$s = 2$							X		X		X
$s = 3$								X	X	X	X

- **Start state**: We use no money, so mark $T(0, 0)$
- **Transition** $(s \to s + 1)$:
  - Loop: $i = 0 \to m$
  - If $T(s, i)$ is marked:
    - Loop: $j = 0 \to |K_c|$
    - Mark $T(s + 1, i + K_{c,j})$
- **Solution**: Maximum marked column of the last row.
- **Note**: Other solutions are possible!

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
$s = 0$	X										
$s = 1$			X		X						
$s = 2$							X		X		X
$s = 3$								X	X	X	X

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

for g in (0 to C-1)
  for i in (0 to M-1):
    if table[g][i] == 1:
       for k in (0 to K[g]-1):
          table[g + 1][i + cost[g][k]] = 1
            ## omitted out of bounds check!
```

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
$s = 0$	X										
$s = 1$			X		X						
$s = 2$							X		X		X
$s = 3$								X	X	X	X

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

for g in (0 to C-1)
  for i in (0 to M-1):
    if table[g][i] == 1:
       for k in (0 to K[g]-1):
          table[g + 1][i + cost[g][k]] = 1
            ## omitted out of bounds check!
```

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
$s = 0$	X										
$s = 1$			X		X						
$s = 2$							X		X		X
$s = 3$								X	X	X	X

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

for g in (0 to C-1)
  for i in (0 to M-1):
    if table[g][i] == 1:
       for k in (0 to K[g]-1):
          table[g + 1][i + cost[g][k]] = 1
            ## omitted out of bounds check!
```

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
$s = 0$	X										
$s = 1$			X		X						
$s = 2$							X		X		X
$s = 3$								X	X	X	X

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

for g in (0 to C-1)
  for i in (0 to M-1):
    if table[g][i] == 1:
       for k in (0 to K[g]-1):
          table[g + 1][i + cost[g][k]] = 1
            ## omitted out of bounds check!
```

DP: 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 (especially python!). Hard to reduce the size of the DP table.

Bottom-Up

Pros:
Faster if you will visit the entire DP table anyway. It is possible to save memory by discarding old rows.

Cons:
Can be harder to create the algorithm. If the DP table is sparse, the loop will visit every state.

Finding the Decision Set with DP

In the first example, the solution is only the final money. However, in other examples, you also need to know the **decisions** used to reach that result.

Example: Print the path with the smallest cost;

It is not very hard to solve this problem. It just requires TWO tables:

- The DP table that we already know;
- The "Parent" table, which indicate which cell led to the current one;

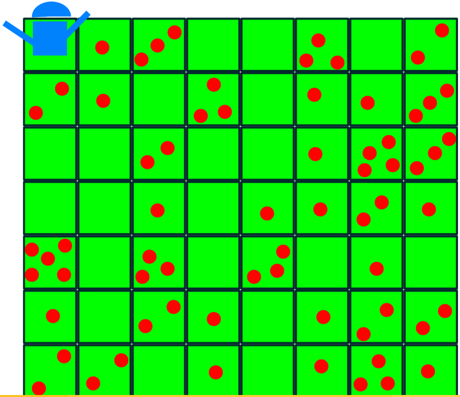
The next example will show the use of the "Parent" table.

Be careful how the problem break ties between identical solutions (lexographic order, first found, any order, etc.)

Example 2: Apple Field

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

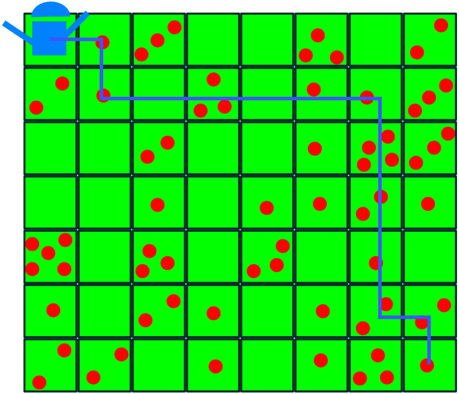
For each cell in the field, you know how many apples the robot can pick. Find the path that maximizes the number of apples the robot picks.



Example 2: Apple Field

Complete Search

One possible solution (not-maximum)



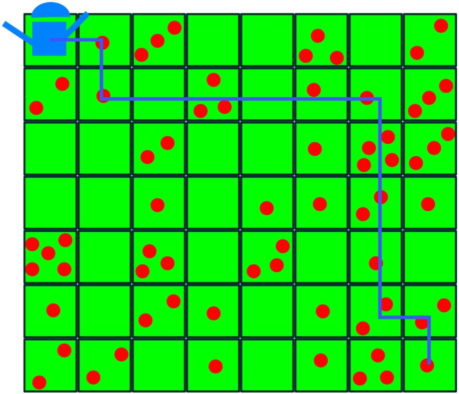
How many different paths are possible? Answer: $\binom{2n}{n} = \frac{(2n)!}{n!n!}$.

Is it necessary to check every path? Or overlapping states exist?

Example 2: Apple Field

Overlapping Solutions

One possible (non-optimal) solution



For any (x, y), the maximum path $(x, y) \to (n, n)$ does not depend on the path $(0, 0) \to (x, y)$. This let us create a DP table based on the current position of the robot.

Example 2: Apple Field

Bottom-up DP

- **DP table and Parent table:**
  - The DP table is a $n+1 \times n+1$ table. At every position, we have the maximum number of apples from $(0,0) \rightarrow (x,y)$.
  - The Parent table is a $n+1 \times n+1$ table. At every position, we store the last-1 cell (up or right) of $(0,0) \rightarrow (x,y)$.
- **Initial Condition:** (DP table only)
  - To avoid special treatment of the first row and first column, we include a "boundary" at the top and right of the table. Every cell at the boundary has "0" apples
- **Transition:**
  - We double loop over the DP table (row $\rightarrow$ column, or vice-versa). For every cell (x,y):
    $DP[x][y] = \text{apple}[x][y] + \max(DP[x-1][y], DP[x][y-1])$
    $\text{Parent}[x][y] = (DP[x-1][y] > DP[x][y-1]?\leftarrow:\uparrow)$

Example 2: Apple Field

Pseudocode

```
int apple[m+1][n+1];
// Input Data. Requires some preprocessing:
//   - Valid data is put 1 to m, 1 to n.

int DP[m+1][n+1];          // Intialized to -1
DP[0][1] = 0;              // Initial state;
int parent[m+1][n+1][2]; // store (x,y) of parent

for (int i = 1; i < m+1; i++) {
  for (int j = 1; j < n+1; j++) {
    DP[m][n] = apple[m][n] + max(DP[m][n-1], DP[m-1][n]);
    if (DP[m][n-1] > DP[m-1][n]):
       parent[m][n][0] = m; parent[m][n][1] = n-1;
    else:
       parent[m][n][0] = m-1; parent[m][n][1] = n;
  }
}
```

Example 2: Apple Field

Simulating the algorithm

Input Table

	1		2	1		1	
	2	1			2		4
		3		1	1		

DP Table

0							

Classical DP Problems

There are some categories of problems that are considered to be "classical DP problems".

- 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.

QUIZ: For each problem, after the problem is explained please spend 10 minutes finding the **DP table**, and the **transition**.

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
```

The 1D Range Sum Problem

Complete Search

Complete Search

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

```
maxindex = []
maxsum = 0
for i in (0:n):
   for j in (i:n):
      sum = 0
      for k in (i:j):
         sum += k
      if sum > maxsum:
         maxsum = sum
         maxindex = [i,j]
```

Because of three loops, this approach is $O(n^3)$. For large values of N (for example 10.000), this is not feasible.

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 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]}

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;

# Be careful with index 0! A[] will begin with 1 here!
```

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

Using a mix of the Sum Table, and a greedy approach, it is possible to sove the Range Sum problem in $O(n)$

```
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 transition)?

Maximum Range Sum 2D

Complete Search

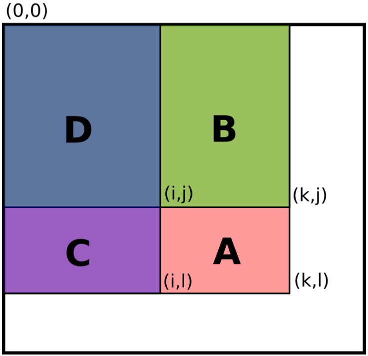
The complete search approach needs 6 loops (2 for horizontal axis, 2 for vertical axis, 2 for calculating the sum). So the total complexity is $O(n^6)$.

```
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 we need to 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):                               // Creating the Sum Table
   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]
                                              // Avoid double count
       if (i > 0 && j > 0) ST[i][j] -= ST[i-1][j-1]

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 Transition) 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) {           // Found a longer subset
    max_len = len; S_max = S;
} }
```

DP for Longest Increasing Subsequence

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

Transition

For every element A[i], that element is either:

- The beginning of a new partial LIS;
- Added to the end of an existing partial LIS;

So for each element, we only need to know which partial LIS this item should be added to.

Table

- **Parent**: Indicate the previous element of the longest partial LIS this element is a member of;
- **LIS**: Indicate the current size of the longest partial LIS this element is a member of;

DP for Longest Increasing Subsequence

Example

```
A      = [ -7, 10, 9, 2, 3, 8, 8, 1 ]
parent = [ -1,  0, 0, 0, 3, 4, 4, 0 ]
LIS    = [  1,  2, 2, 2, 3, 4, 4, 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. I'll leave that one for you to find by yourself!

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* where the sum of sizes is under *M*, and the sum of values is as high as possible.

```
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

Because we already have a recursive function, it is very easy to modify *value(id,size)* to use a DP table as memory. Let's see an example:

```
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														

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.

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?
- What is the DP Table and Transition?

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 + change(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.
- For Bottom-UP DP, it is easier to use a table indexed on COINS

Bottom-UP DP

```
boolean DP[c][v] = FALSE;       // Can we reach v with c coins?

i = 0; DP[0][0] = TRUE;         // Start condition
while (TRUE):
  i+=1; possible = FALSE        // Start the loop
  for j = 0 to V:
    if (DP[i-1][j]):            // For each reachable value of V
      possible = TRUE           // We can continue
      if (j == V): return c-1   // Found a solution, go back!
      for k in (coins):         // update all coins
        DP[i][j+k] = TRUE       // Mark new reachable values
  if (!possible): return -1     // No solution found
```

DP for Coin Change

Simulation

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

	0	1	2	3	4	5	6	7
0	T							
1								
2								
3								
4								

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.

Summary

Dynamic Programming is a search technique that uses memoization to **avoid recalculation overlapping partial solutions.**

There are two main types of solutions:

- Top-down DP: Add memory to a recursive full search;
- Bottom-up DP: Fill the DP table using a for loop;

To create a DP, you need to decide the **DP table** and the **Transition rules**.

DP problems are very common in programming competitions. If you are good at DP, you will be able to get a good (but not best) rank in several contests.

Problem Discussion – At a Glance

- Wedding Shopping – Explained in Class
- Jill Rides Again – Range Sum (1D)
- Largest Submatrix – Range Sum (2D)
- Is Bigger Smarter? – Longest Increasing Subsequence
- Murcia's Skyline – Longest Increasing Subsequence
- Trouble of 13 Dots – 0-1 Knapsack
- Exact Change – Coin Change
- Unidirectional TSP – Pathfinding

Problem Hints

Largest Submatrix

Find the largest patch of **ones** inside a matrix of 1s and 0s.

Hints:

- Do a range sum to find the rectangle with biggest sum (biggest number of 1).
- **Key Idea**: How do you avoid adding zeroes?

This kind of problem sometimes appears as the initial part of a more complex problem, to calculate valid territory.

Problem Hints

Is bigger Smarter?

You have the "weight" and "intelligence" value of a set of elephants. Find the largest subset where:

- A - Intelligence is decreasing, and;
- B - Weight is increasing

Hints:

- Think about "Dragon of Loowater" from last lesson.

Problem Hints

Murcia Skyline

Compare the size of the Longest **Increasing** skyline and the longest **Decreasing** skyline.

Hints:

- "Longest Increasing Subsequence" in this problem is modified by the **building width.**

Problem Hints

Trouble of 13 dots

Find the subset of items that:

- Mazimize flavor;
- Is inside the price budget; You can get a discount;

Hints

- 1-0 knapsack problem:
- Be careful with special rule: **the knapsack change size if price > 2000!**

Problem Hints

Exact Change

Find the smallest amount of overpay that you can do, with the smallest number of coins.

Hints:

- Variation of the Coin Change problem discussed in Class;
- Calculate all possible changes above the desired value, and find the smallest;
- Order by smallest number of coins necessary;
- Bottom Up algorithm is probably best;

Problem Hints 6

Unidirectional TSP

Find the minimal path from left to right. Up and down are connected!

- Very similar to the “apple robot” problem;
- Note that when two paths have the same weight, the smaller index is best!

About these Slides

These slides were made by Claus Aranha, 2020. You are welcome to copy, re-use and modify this material.

Individual images in some slides might have been made by other authors. Please see the references in each slide for those cases.

Image Credits I

[Page 5] Wedding Dress Image CC-By-2.0 by `https://www.flickr.com/photos/vancouver125/5634967507`