4. Dynamic programming 1 (basic ideas)

CPCFI

UNAM's School of Engineering

2021

Based on: Halim S., Halim F.*Competitive Programming 3*. Handbook for ACM ICPC and IOI Contestants. 2013

Table of Contents

Index

CPCFI

Dynamic Programming - Background

The most challenging algorithmic problems involve optimization, where we seek to find a solution that maximizes or minimizes and objective function

CPCFI

Dynamic Programming - Background

Algorithms for optimization problems require proof that they always return the best possible solution.

- Greedy algorithms that make best local decisions at each step are efficient but do not guarantee global optimality

Dynamic Programming - Background

Algorithms for optimization problems require proof that they always return the best possible solution.

- Greedy algorithms that make best local decisions at each step are efficient but do not guarantee global optimality
- Complete Search algorithms always produce the optimal solution but suffer from great time complexity

CPCFI

Dynamic Programming - Background

Algorithms for optimization problems require proof that they always return the best possible solution.

- Greedy algorithms that make best local decisions at each step are efficient but do not guarantee global optimality
- Complete Search algorithms always produce the optimal solution but suffer from great time complexity

Dynamic Programming gives us a way to design custom algorithms that **systematically search** all possibilities (guaranteeing correctness) while **storing** intermediate results to avoid recomputing (efficiency)

Dynamic Programming - Background

- Dynamic Programming is the fourth paradigm of algorithm construction (CS, Greedy, D&Q) occupied for efficiently implementing a recursive algorithm by sorting partial results.

Dynamic Programming - Background

- Dynamic Programming is the fourth paradigm of algorithm construction (CS, Greedy, D&Q) occupied for efficiently implementing a recursive algorithm by sorting partial results.
- It requires searching for a recursive structure that computes the same sub-problems repeatedly. Once we identify this structure, the idea is to store the answer for each sub-problem in a table to further look up the answer instead of computing again the same sub-problem.

Prerequisites for DP

1. The problem has optimal sub-structures
2. The problem has overlapping sub-problems

- In the next subsection (3.5.1 Illustration - DP Approach), we'll see an example (UVa - 11450) where we look into detail the definition of this two prerequisites

CPCFI

How to identify a DP problem?

- DP is primarily used to solve *optimization* problems and *counting* problems
- Most DP problems will begin with "minimize this", "maximize that" or "count the ways to do that"

Index

CPCFI

Table of Contents

CPCFI

UVa 11450 - Wedding Shopping

Problem Description - UVa 11450

Given different options for each garment (e.g. 3 shirt models, 2 belt models, 4 shoe models, ...) and a certain limited budget, our task is to buy one model of each garment. We cannot spend more money than the given budget, but we want to spend the maximum possible amount.

CPCFI

UVa 11450 - Wedding Shopping

Input:

- Budget M: $1 \leq M \leq 200$
- Number of garments C: $1 \leq C \leq 20$
- For each garment g_i: $g_i \in [0, \ldots, C-1]$:
  - Number of models K: $1 \leq K \leq 20$
  - Price for each model p_i: $p_i \in [1, \ldots, K]$

Output: one integer that indicates the maximum amount of money we can spend purchasing one of each garment without exceeding the budget. It is possible that there is no solution to some test cases.

CPCFI

UVa 11450 - Wedding Shopping

Test case A: $M = 20$ and $C = 3$:

- $K_0 = 3 \rightarrow \{6, 4, 8\}$
- $K_1 = 2 \rightarrow \{5, 10\}$
- $K_2 = 4 \rightarrow \{1, 5, 3, 5\}$

UVa 11450 - Wedding Shopping

Test case A: $M = 20$ and $C = 3$:

- $K_0 = 3 \rightarrow \{6, 4, 8\}$
- $K_1 = 2 \rightarrow \{5, 10\}$
- $K_2 = 4 \rightarrow \{1, 5, 3, 5\}$

Solution:

The answer is 19 which may be formed with the following combinations of garments' purchases:

1. $8 + 10 + 1$
2. $6 + 10 + 3$
3. $4 + 10 + 5$

UVa 11450 - Wedding Shopping

Test case B: $M = 9$ and $C = 3$:

- $K_0 = 3 \rightarrow \{6, 4, 8\}$
- $K_1 = 2 \rightarrow \{5, 10\}$
- $K_2 = 4 \rightarrow \{1, 5, 3, 5\}$

UVa 11450 - Wedding Shopping

Test case B: $M = 9$ and $C = 3$:

- $K_0 = 3 \rightarrow \{6, 4, 8\}$
- $K_1 = 2 \rightarrow \{5, 10\}$
- $K_2 = 4 \rightarrow \{1, 5, 3, 5\}$

Solution:
The answer is *no solution*

UVa 11450 - Wedding Shopping

Now, let's explore some approaches to solve this problem...

1. Greedy - **WA** verdict
2. Complete Search - **TLE** verdict
3. Divide and Conquer - **Not possible**
4. Top-Down DP - **AC** verdict
5. Bottom-Up DP - **AC** verdict

UVa 11450: Greedy Approach

Since we want to maximize the budget spent, one greedy idea (there are other greedy approaches—which also produces *WA* verdict) is to take the most expensive model for each garment g_i which still fits our budget.

This greedy idea works for test cases A and B and both produce the optimal solution, however, if we look into test case C, this strategy won't work.

UVa 11450: Greedy Approach

Test case C: $M = 12, C = 3$:

- $K_0 = 3 \to \{6, 4, 8\}$
- $K_1 = 2 \to \{5, 10\}$
- $K_2 = 4 \to \{1, 5, 3, 5\}$

CPCFI

UVa 11450: Greedy Approach

Test case C: $M = 12, C = 3$:

- $K_0 = 3 \rightarrow \{6, 4, 8\}$
- $K_1 = 2 \rightarrow \{5, 10\}$
- $K_2 = 4 \rightarrow \{1, 5, 3, 5\}$

Solution: the greedy strategy will select for K_0 price 8, for K_1 price 10 and for K_2 price 5 which results in a total price of $23 >> M$. Therefore, producing a *no solution* answer which is incorrect.

CPCFI

UVa 11450: Complete Search Approach

- Let's start with a function `shop(money, g)` where the pair `(money, g)` is a state of the problem

UVa 11450: Complete Search Approach

- Let's start with a function `shop(money, g)` where the pair `(money, g)` is a state of the problem
- We start with `money=M` and `g=0`

UVa 11450: Complete Search Approach

- Let's start with a function `shop(money, g)` where the pair `(money, g)` is a state of the problem
- We start with `money=M` and `g=0`
- We try all possible models in garment $g = 0$ and if model i is chosen, we subtract model i's price from `money`

UVa 11450: Complete Search Approach

- Let's start with a function `shop(money, g)` where the pair `(money, g)` is a state of the problem
- We start with `money=M` and `g=0`
- We try all possible models in garment $g = 0$ and if model i is chosen, we subtract model i's price from `money`
- We repeat this process recursively and stop in either the last garment or when `money < 0` before reaching the last garment model

CPCFI

UVa 11450: Complete Search Approach

- Let's start with a function `shop(money, g)` where the pair `(money, g)` is a state of the problem
- We start with `money=M` and `g=0`
- We try all possible models in garment $g = 0$ and if model i is chosen, we subtract model i's price from `money`
- We repeat this process recursively and stop in either the last garment or when `money < 0` before reaching the last garment model
- Among all valid combinations, we can then pick the one that results in the smallest non-negative `money`

UVa 11450: Complete Search Approach

We can define the recurrences formally as follows:

1. If `money < 0`:
   - `shop(money, g) =` $-\infty$

UVa 11450: Complete Search Approach

We can define the recurrences formally as follows:

1. If `money < 0`:
   - `shop(money, g) =` $-\infty$
2. If a model from the last garment has been bought, i.e. $g = C$:
   - `shop(money, g)= M - money`

UVa 11450: Complete Search Approach

We can define the recurrences formally as follows:

1. If `money < 0`:
   - `shop(money, g) =` $-\infty$
2. If a model from the last garment has been bought, i.e. $g = C$:
   - `shop(money, g)= M - money`
3. $\forall$ garment model $g_i \in [1, \ldots, K]$:
   - `shop(money,g)=max(shop(money-price[g][model],g+1))`

UVa 11450: Complete Search Approach

- This solution works correctly but it's **very slow**
- In the largest test case, garment $g = 0$ has up to 20 models, garment $g = 1$ also has up to 20 models and all garments including the last garment $g = 19$ also have up to 20 models
- Therefore, this Complete Search solution runs in $20 \times 20 \times \ldots \times 20$ operations, i.e. $20^{20} = 1.04 \times 10^{26}$ which produces a **Time Limit Exceeded TLE** verdict

UVa 11450: Divide and Conquer Approach

- This problem is not solvable using the Divide and Conquer paradigm, since the sub-problems are not independent
- The sub-problems are the problem's states described above in the Complete Search approach

UVa 11450: Top-Down DP Approach

UVa 11450 satisfies the two prerequisites for DP to be applicable:

1. Optimal sub-structures
   - The solution for the sub-problem is part of the solution of the original problem
   - If we select model i for garment $g = 0$, for our final selection to be optimal, our choice for garments $g = 1$ and above must also be the optimal choice for a reduced budget of `M - price`, where `price` refers to the price of model i

UVa 11450: Top-Down DP Approach

UVa 11450 satisfies the two prerequisites for DP to be applicable:

1. Optimal sub-structures
   - The solution for the sub-problem is part of the solution of the original problem
   - If we select model i for garment $g = 0$, for our final selection to be optimal, our choice for garments $g = 1$ and above must also be the optimal choice for a reduced budget of `M - price`, where `price` refers to the price of model i
2. Overlapping sub-problems
   - This is the key characteristic of DP
   - The search space of this problem is not as big as the rough 20^{20} bound obtained earlier because **many** sub-problems are overlapping

CPCFI

UVa 11450: Top-Down DP Approach

Are there any **repeated** (overlapping) sub-problems?

UVa 11450: Top-Down DP Approach

Are there any **repeated** (overlapping) sub-problems?

This happens if some combination of `money` and chosen model's price causes `money1 - p1 = money2 - p2` for the same garment g.

UVa 11450: Top-Down DP Approach

So, how many distinct sub-problems are there? Considering that `money` $\in [0, \ldots, 200]$ and the number of garments $\in [0, \ldots, 19]$.

UVa 11450: Top-Down DP Approach

So, how many distinct sub-problems are there? Considering that `money` $\in [0, \ldots, 200]$ and the number of garments $\in [0, \ldots, 19]$.

$$201 \times 20 = 4020 \text{ states}$$

CPCFI

UVa 11450: Top-Down DP Approach

So, how many distinct sub-problems are there? Considering that `money` $\in [0, \ldots, 200]$ and the number of garments $\in [0, \ldots, 19]$.

$$201 \times 20 = 4020 \text{ states}$$

Each sub-problem must be computed only once and by ensuring this, we could solve the problem much faster.

UVa 11450: Top-Down DP Approach

In order to implement the Top-Down DP solution we must add the following steps:

1. Initialize a DP table with dummy values that are not used in the problem
   - This table must have dimensions accordingly to the problem states

UVa 11450: Top-Down DP Approach

In order to implement the Top-Down DP solution we must add the following steps:

1. Initialize a DP table with dummy values that are not used in the problem
   - This table must have dimensions accordingly to the problem states
2. At the start of the recursive function, check if this state has been computed before
   - If it has, return the value from the DP table $O(1)$
   - If it has not, perform the computation and then store the computed value in the DP table

UVa 11450: Top-Down DP Approach

DP analysis:

- If the problem has M states, then the program will require $O(M)$ memory space
- If computing one state requires $O(k)$ steps, the the overall time complexity is $O(kM)$
- UVa 11450 has $M = 4020$ states and $k = 20$ (at most 20 models for each garment g)
- Therefore, $80,400$ operations per test case[1] which is ideal

* `C++ code: ch3_02_UVa11450_td.cpp`

[1] Recall that modern computers can perform $10^8 = 100M$ operations per second

UVa 11450: Bottom-Up DP Approach

Steps to build a bottom-up DP solution:

1. Determine the required set of parameters that uniquely describe the problem (state)

UVa 11450: Bottom-Up DP Approach

Steps to build a bottom-up DP solution:

1. Determine the required set of parameters that uniquely describe the problem (state)
2. If there are N parameters required to represent the states, prepare an N dimensional DP table, with one entry per state
   - In Bottom-Up DP, we only need to initialize some cells of the DP table with known initial values (the base cases)
   - In Top-Down DP, we initialize the memo table completely with dummy values (usually -1) to indicate that we have not yet computed the values

CPCFI

UVa 11450: Bottom-Up DP Approach

Steps to build a bottom-up DP solution:

1. Determine the required set of parameters that uniquely describe the problem (state)
2. If there are N parameters required to represent the states, prepare an N dimensional DP table, with one entry per state
   - In Bottom-Up DP, we only need to initialize some cells of the DP table with known initial values (the base cases)
   - In Top-Down DP, we initialize the memo table completely with dummy values (usually -1) to indicate that we have not yet computed the values
3. Determine the cells/states that can be filled next (the transitions)
4. Repeat (iterate) until the DP table is complete

UVa 11450: Bottom-Up DP Approach

Bottom-Up DP for UVa 11450:

1. Parameters: current garment g and current `money`
2. Initialize a 2D boolean matrix (DP table): `reachable[g][money]` of size 20×201
3. Cells/states when $g = 0$ are marked as true (using test case A's parameters: $M = 20$, $C = 3$)
   - `reachable[0][M - 6] = 1`
   - `reachable[0][M - 4] = 1`
   - `reachable[0][M - 8] = 1`

UVa 11450: Bottom-Up DP Approach

Bottom-Up DP for UVa 11450:

1. Parameters: current garment g and current `money`
2. Initialize a 2D boolean matrix (DP table): `reachable[g][money]` of size 20×201
3. Cells/states when $g = 0$ are marked as true (using test case A's parameters: $M = 20$, $C = 3$)
   - `reachable[0][M - 6] = 1`
   - `reachable[0][M - 4] = 1`
   - `reachable[0][M - 8] = 1`

money =>

g =>	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	0	1	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Figure: DP table for $g = 0$

UVa 11450: Bottom-Up DP Approach

4. Iterate with the remaining garments ($g = \{1, 2\}$) as follows:
   - If `reachable[g-1][money]` is true, then `reachable[g][money - p]` will be set to true as long as `money` is not negative[2]
   - For example, `reachable[0][16]` propagates to `reachable[1][16-5]` and `reachable[1][16-10]`

money =>

g =>	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	0	1	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	0	1	0	0	0	0
1	0	0	1	0	1	0	1	1	0	1	0	1	0	0	0	0	0	0	0	0	0
2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	0	1	0	0	0	0
1	0	0	1	0	1	0	1	1	0	1	0	1	0	0	0	0	0	0	0	0	0
2	0	1	1	1	1	1	1	0	1	0	1	0	0	0	0	0	0	0	0	0	0

Figure: DP table when $g = 1$ and $g = 2$

[2] p is the price for garment g_i

UVa 11450: Bottom-Up DP Approach

- The answer can be found in the last row when `g = C-1`
- Find the state in that row that is both nearest to index 0 and reachable
- By looking at the DP table, we can see that the answer lies in `reachable[2][1]`
- Meaning that we can reach state `money = 1` by buying some combination of the various garment models
- The final answer is `M - money = 20 - 1 = 19`
- `C++ code: ch3_03_UVa11450_bu.cpp`

CPCFI

Top-Down vs. Bottom-Up DP

Top-Down	Bottom-Up
Pros: 1. It is a natural transformation from the normal Complete Search recursion 2. Computes the sub-problems only when necessary (sometimes this is faster)	Pros: 1. Faster if many sub-problems are revisited as there is no overhead from recursive calls 2. Can save memory space with the 'space saving trick' technique
Cons: 1. Slower if many sub-problems are revisited due to function call overhead (this is not usually penalized in programming contests) 2. If there are M states, an $O(M)$ table size is required, which can lead to MLE for some harder problems (except if we use the trick in Section 8.3.4)	Cons: 1. For programmers who are inclined to recursion, this style may not be intuitive 2. If there are M states, bottom-up DP visits and fills the value of *all* these M states

Figure: Top-Down vs. Bottom-Up DP, [Halim]:page 102

Index

CPCFI

Table of Contents

DP Classical Problems

- UVa 11450 was an example of a non-classical DP problem
- However, there exists a list of classical DP problems (6) where the states and transitions are *well-known*
- These problems should be mastered to perform well in ICPC in addition to its variants

DP Classical Problems

1. Max 1D Range Sum
2. Max 2D Range Sum
3. Longest Increasing Subsequence (LIS)
4. 0-1 Knapsack (Subset Sum)
5. Coin Change (CC) - General Version
6. Traveling Salesman Problem (TSP)

1. Max 1D Range Sum

- UVa 507 - Jill Rides Again
- `C++ code: ch3_04_Max1DRangeSum.cpp`

Problem Description:
Given an integer array A containing $n \leq 20K$ non-zero integers, determine the maximum (1D) range sum of A. In other words, find the maximum Range Sum Query (RSQ) between two indices $i, j \in [0, \ldots, n-1]$, that is:

$$A[i] + A[i+1] + A[i+2] + \ldots + A[j]$$

CPCFI

1. Max 1D Range Sum

DP strategy

- Pre-process array `A` by computing `A[i]+=A[i-1]` $\forall i \in [1, \ldots, n-1]$ so that `A[i]` contains the sum of integers in subarray `A[0,...,i]`
- We can now compute `RSQ(i,j)` in $O(1)$ as follows: `RSQ(0,j)=A[j]` and `RSQ(i,j)=A[j]-A[i-1]` $\forall i > 0$

1. Max 1D Range Sum

Kadane's $O(n)$ Algorithm

- Keeps a sum of the integers seen so far and greedily reset that to 0 if the running sum dips below 0
- Restarting from 0 is better than continuing from a negative sum
- At each step we have two choices:
  1. Keep using the accumulated maximum sum
  2. Begin a new range
- `ch3_04_Max1DRangeSum.cpp` uses the space saving trick

2. Max 2D Range Sum

- UVa 108 - Maximum Sum
- `C++ code: ch3_05_UVa108.cpp`

Problem Description

Given an $n \times n$ $(1 \leq n \leq 100)$ square matrix of integers A where each integer ranges from $[-127, \ldots, 127]$, find a sub-matrix of A with the maximum sum.

CPCFI

2. Max 2D Range Sum

For example, the 4×4 matrix ($n = 4$) below has a 3×2 sub-matrix on the lower-left with maximum sum of $9 + 2 - 4 + 1 - 1 + 8 = 15$.

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

2. Max 2D Range Sum

DP strategy

- The solution for the Max 1D Range Sum can be extended to two (or more) dimensions and we'll be dealing now with cumulative $n \times n$ sub-matrices
- `A[i][j]` no longer contains its own value, but the sum of all items within sub-matrix $(0, 0)$ to (i, j)

CPCFI

2. Max 2D Range Sum

The code shown below turns the input square matrix into a cumulative sum matrix:

```
scanf("\%d", &n);
for (int i = 0; i < n; i++) for (int j = 0; j < n; j++) {
  scanf("\%d", &A[i][j]);
  if (i > 0) A[i][j] += A[i - 1][j];
  if (j > 0) A[i][j] += A[i][j - 1];
  if (i > 0 && j > 0) A[i][j] -= A[i - 1][j - 1];
}
```

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

B	0	-2	-9	-9
	9	9	-4	2
	5	6	-11	-8
	4	13	-4	-3

2. Max 2D Range Sum

To obtain the sum of a sub-matrix going from (i, j) to (k, l) in $O(1)$, we do the following:

- `sum = A[k][l]`
- If $i > 0$, `sum -= A[i-1][l]`
- If $j > 0$, `sum -= A[k][j-1]`
- If $i > 0$ and $j > 0$, `sum += A[i-1][j-1]`

2. Max 2D Range Sum

For example, let's compute the sum of $(1, 2)$ to $(3, 3)$:

- `sum = A[3][3] = -3`
- If $i > 0$, `sum -= A[0][3] = -9`
- If $j > 0$, `sum -= A[3][1] = 13`
- If $i > 0$ and $j > 0$, `sum += A[0][1] = -2`
- `sum = -9`

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

B				
	0	-2	-9	-9
	9	9	-4	2
	5	6	-11	-8
	4	13	-4	-3

C				
	0	-2	-9	-9
	9	9	-4	2
	5	6	-11	-8
	4	13	-4	-3

3. Longest Increasing Subsequence (LIS)

- `C++ code: ch3_06_LIS.cpp`

Problem Description:
Given a sequence $\{A[0], A[1], \ldots, A[n-1]\}$, determine its Longest Increasing Subsequence (LIS). Note that these *subsequences* are not necessarily contiguous.

For example, $n = 8$, $A = \{-7, 10, 9, 2, 3, 8, 8, 1\}$. The length-4 LIS is $\{-7, 2, 3, 8\}$

3. Longest Increasing Subsequence (LIS)

Solution

- Let `LIS(i)` be the longest increasing subsequence ending at index `i`
- Therefore, we can model this problem with one parameter: `i`

3. Longest Increasing Subsequence (LIS)

Solution

- Let `LIS(i)` be the longest increasing subsequence ending at index `i`
- Therefore, we can model this problem with one parameter: `i`
- `LIS(0) = 1` since the first number in `A` is a subsequence

CPCFI

3. Longest Increasing Subsequence (LIS)

Solution

- Let `LIS(i)` be the longest increasing subsequence ending at index `i`
- Therefore, we can model this problem with one parameter: `i`
- `LIS(0) = 1` since the first number in `A` is a subsequence
- For `LIS(i)` given that $i \geq 1$ we need to do the following:

3. Longest Increasing Subsequence (LIS)

Solution

- Let `LIS(i)` be the longest increasing subsequence ending at index `i`
- Therefore, we can model this problem with one parameter: `i`
- `LIS(0) = 1` since the first number in `A` is a subsequence
- For `LIS(i)` given that $i \geq 1$ we need to do the following:
  - Find an index `j` such that $j < i$

3. Longest Increasing Subsequence (LIS)

Solution

- Let `LIS(i)` be the longest increasing subsequence ending at index `i`
- Therefore, we can model this problem with one parameter: `i`
- `LIS(0) = 1` since the first number in `A` is a subsequence
- For `LIS(i)` given that $i \geq 1$ we need to do the following:
  - Find an index `j` such that $j < i$
  - `A[j] < A[i]`

3. Longest Increasing Subsequence (LIS)

Solution

- Let `LIS(i)` be the longest increasing subsequence ending at index `i`
- Therefore, we can model this problem with one parameter: `i`
- `LIS(0) = 1` since the first number in `A` is a subsequence
- For `LIS(i)` given that $i \geq 1$ we need to do the following:
    - Find an index `j` such that $j < i$
    - `A[j] < A[i]`
    - `A[j]` must be the largest

3. Longest Increasing Subsequence (LIS)

Solution

- Let `LIS(i)` be the longest increasing subsequence ending at index `i`
- Therefore, we can model this problem with one parameter: `i`
- `LIS(0) = 1` since the first number in `A` is a subsequence
- For `LIS(i)` given that $i \geq 1$ we need to do the following:
  - Find an index `j` such that $j < i$
  - `A[j] < A[i]`
  - `A[j]` must be the largest
- Once we found index `j`, `LIS(i) = LIS(j) + 1`

Index	0	1	2	3	4	5	6	7
A	-7	10	9	2	3	8	8	1
LIS(i)	1	2	2	2	3	4	4	2

3. Longest Increasing Subsequence (LIS)

We can write this recurrence formally as:

1. `LIS(0) = 1`, the base case

3. Longest Increasing Subsequence (LIS)

We can write this recurrence formally as:

1. `LIS(0) = 1`, the base case
2. `LIS(i) = max(LIS(j)+1)`, $\forall j \in [0, \ldots, i-1]$ and `A[j] < A[i]`, the recursive case

3. Longest Increasing Subsequence (LIS)

We can write this recurrence formally as:

1. `LIS(0) = 1`, the base case
2. `LIS(i) = max(LIS(j)+1)`, $\forall j \in [0, \ldots, i-1]$ and `A[j] < A[i]`, the recursive case

The answer is the largest value of `LIS(k)` $\forall k \in [0, \ldots, n-1]$

Index	0	1	2	3	4	5	6	7
A	-7	10	9	2	3	8	8	1
LIS(i)	1	2	2	2	3	4	4	2

3. Longest Increasing Subsequence (LIS)

- Using the DP strategy described above, the algorithm will run in $O(n^2)$ time since we need to make a loop for each index i
- In addition to this, there are many overlapping sub-problems due to the same reason
- To further improve this algorithm, we could store the predecessor information and trace the green arrows from index `k` that contains the largest value of `LIS(k)`

3. Longest Increasing Subsequence (LIS)

However, there is a solution using the Greedy and D&C paradigms that runs in $O(n \log k)$:

- Let array `L` be an array such that `L(i)` represents the smallest ending value of all length-i LISs found so far

3. Longest Increasing Subsequence (LIS)

However, there is a solution using the Greedy and D&C paradigms that runs in $O(n \log k)$:

- Let array `L` be an array such that `L(i)` represents the smallest ending value of all length-i LISs found so far
- `L(i-i)` will always be smaller than `L(i)`

3. Longest Increasing Subsequence (LIS)

However, there is a solution using the Greedy and D&C paradigms that runs in $O(n \log k)$:

- Let array `L` be an array such that `L(i)` represents the smallest ending value of all length-i LISs found so far
- `L(i-i)` will always be smaller than `L(i)`
- Finally, we can binary search array `L` to determine the longest possible subsequence we can create by appending the current element `A[i]`

4. 0-1 Knapsack (Subset Sum)

- C++ code: `ch3_07_UVa10130.cpp`

Problem Description:
Given n items, each with its own value V_i and weight W_i, $\forall i \in [0, \ldots, n-1]$, and a maximum knapsack size S, compute the maximum value of the items that we can carry, if we can either ignore or take a particular item (hence the term 0-1 for ignore/take)

4. 0-1 Knapsack (Subset Sum)

For example, $n = 4$, $V = \{100, 70, 50, 10\}$, $W = \{10, 4, 6, 12\}$, $S = 12$.

- If we select item 0 with weight 10 and value 100, we cannot take any other item. Not optimal
- If we select item 3 with weight 12 and value 10, we cannot take any other item. Not optimal
- If we select item 1 and 2, we have total weight 10 and total value 120. **This is the maximum**

CPCFI

4. 0-1 Knapsack (Subset Sum)

Let's look at the following Complete Search recurrences for `val(id, remW)` where `id` is the index of the current item to be considered and `remw` the remaining weight in the knapsack:

1. `val(id, 0) = 0`, if `remW = 0` we cannot take anything else

4. 0-1 Knapsack (Subset Sum)

Let's look at the following Complete Search recurrences for `val(id, remW)` where `id` is the index of the current item to be considered and `remw` the remaining weight in the knapsack:

1. `val(id, 0) = 0`, if `remW = 0` we cannot take anything else
2. `val(n, remW) = 0` , if `id = n` we have considered all items

4. 0-1 Knapsack (Subset Sum)

Let's look at the following Complete Search recurrences for `val(id, remW)` where `id` is the index of the current item to be considered and `remw` the remaining weight in the knapsack:

1. `val(id, 0) = 0`, if `remW = 0` we cannot take anything else
2. `val(n, remW) = 0` , if `id = n` we have considered all items
3. `if W[id] > remW`, we have to ignore this item
   - `val(id, remW) = val(id + 1, remW)`

4. 0-1 Knapsack (Subset Sum)

Let's look at the following Complete Search recurrences for `val(id, remW)` where `id` is the index of the current item to be considered and `remw` the remaining weight in the knapsack:

1. `val(id, 0) = 0`, if `remW = 0` we cannot take anything else
2. `val(n, remW) = 0` , if `id = n` we have considered all items
3. `if W[id] > remW`, we have to ignore this item
   - `val(id, remW) = val(id + 1, remW)`
4. `if W[id] <= remW`, we have two choices, either ignore the item or take it. We take the one that gives us the maximum value:

```
val(id, remW) =
  max( val(id+1,remW), V[id]+val(id+1,remW-W[id]))
```

CPCFI

5. Coin Change (CC) - General Version

- C++ code: `ch3_08_UVa674.cpp`

Problem Description:
Given a target amount V cents and a list of denominations for n coins, i.e. we have `coinValue[i]` (in cents) for coin types $i \in [0, \ldots, n-1]$, what is the minimum number of coins that we must use to represent V ? Assume that we have unlimited supply of coins of any type

5. Coin Change (CC) - General Version

Let's look at the following Complete Search recurrences for `change(value)` where `value` is the remaining amount of cents that we need to represent in coins:

1. `change(0) = 0`, we need 0 coins to produce 0 cents

5. Coin Change (CC) - General Version

Let's look at the following Complete Search recurrences for `change(value)` where `value` is the remaining amount of cents that we need to represent in coins:

1. `change(0) = 0`, we need 0 coins to produce 0 cents
2. `change(< 0) =` ∞ , we can return a large positive value

5. Coin Change (CC) - General Version

Let's look at the following Complete Search recurrences for `change(value)` where `value` is the remaining amount of cents that we need to represent in coins:

1. `change(0) = 0`, we need 0 coins to produce 0 cents
2. `change(< 0) =` ∞ , we can return a large positive value
3. `change(value) = 1+min(change(value-coinValue[i]))`, $\forall i \in [0, \ldots, n-1]$

<0	0	1	2	3	4	5	6	7	8	9	10
∞	0	1	2	3	4	1	2	3	4	5	2

V = 10, N = 2, coinValue= {1, 5}

5. Coin Change (CC) - Alternative Version

A variant of this problem is to count the number of possible (canonical) ways to get value V cents using a list of denominations of n coins.

For example, for $V = 10$ above, the answer is $3 : \{1+1+1+1+1+1+1+1+1+1, 5+1+1+1+1+1, 5+5\}$

5. Coin Change (CC) - Alternative Version

Complete Search recurrences for `ways(type, value)` where `type` is the index of the coin type that we are currently considering:

1. `ways(type, 0) = 1`, one way, just use nothing
2. `ways(type, <0) = 0`, no way since we cannot reach a negative value
3. `ways(n, value) = 0`, no way since we have considered all coin types $\in [0, \ldots, n-1]$
4. `ways(type,value) = ways(type+1,value) + ways(type,value-coinValue[type])`, if we ignore this coin type plus if we use this coin type

CPCFI

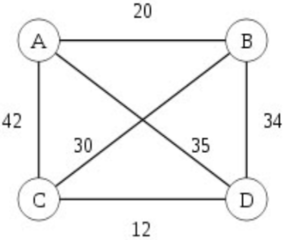
6. Traveling Salesman Problem (TSP)

- UVa 10496 - Collecting Beepers
- `C++ code: ch3_09_UVa10496.cpp`

Problem Description:
Given n cities and their pairwise distances in the form of a matrix `dist` of size $n \times n$, compute the cost of making a tour that starts from any city s, goes through all the other $n - 1$ cities exactly once, and finally returns to the starting city s.

6. Traveling Salesman Problem (TSP)

For example, for $n = 4$ cities, we have $4! = 24$ possible tours (permutations of 4 cities). One of the minimum tours is `A-B-C-D-A` with a cost of $20 + 30 + 12 + 35 = 97$



	A	B	C	D
A	0	20	42	35
B	20	0	30	34
C	42	30	0	12
D	35	34	12	0

6. Traveling Salesman Problem (TSP)

- TSP presents various overlapping sub-problems
  - The tour `A-B-C-(n-3)` other cities overlaps with the tour `A-C-B-(n-3)` same other cities that also return to `A`
- If we can avoid re-computing various sub-tours we can save a lot of computation time
- Parameters for TSP:
  1. The last city visited: `pos`
  2. Subset of the visited cities

6. Traveling Salesman Problem (TSP)

How do we represent a set?

- We need the set to be lightweight since it'll be used as a parameter in a recursive function

6. Traveling Salesman Problem (TSP)

How do we represent a set?

- We need the set to be lightweight since it'll be used as a parameter in a recursive function
- Solution: **bitmask**
- If we have n cities, we could use an integer of length n

6. Traveling Salesman Problem (TSP)

How do we represent a set?

- We need the set to be lightweight since it'll be used as a parameter in a recursive function
- Solution: **bitmask**
- If we have n cities, we could use an integer of length n
- If the bit i is 1 (on), we say that the city with index i has been visited (inside the set)
- Otherwise, it has not been visited yet ($i = 0$, off)

6. Traveling Salesman Problem (TSP)

For example

- $\texttt{mask} = 18_{10} = 10010_2$
- Implies that cities **1** and **4** have been visited
- To check if a bit i is on or off: $\texttt{mask}\ \&(1 << i)$
- To set bit i: $\texttt{mask}\ | = (1 << i)$

6. Traveling Salesman Problem (TSP)

Complete Search recurrences for `tsp(pos, mask)`:

1. `tsp(pos,`2^n`-1) = dist[pos][0]`, all cities have been visited
2. `tsp(pos, mask) = min(dist[pos][nxt] + tsp(nxt, mask | (1 << nxt)))`, $\forall$ `nxt` $\in [0, \ldots, n-1]$, `nxt != pos` and `(mask & (1 << nxt))` is 0 (turned off). We try all possible next cities that have not been visited before at each step

6. Traveling Salesman Problem (TSP)

- There are only $O(n \times 2^n)$ distinct states because there are n cities and we remember up to 2^n other cities that have been visited in each tour

CPCFI

6. Traveling Salesman Problem (TSP)

- There are only $O(n \times 2^n)$ distinct states because there are n cities and we remember up to 2^n other cities that have been visited in each tour
- Each state can be computed in $O(n)$, thus the overall time complexity of this DP solution is$O(2^n \times n^2)$

6. Traveling Salesman Problem (TSP)

- There are only $O(n \times 2^n)$ distinct states because there are n cities and we remember up to 2^n other cities that have been visited in each tour
- Each state can be computed in $O(n)$, thus the overall time complexity of this DP solution is$O(2^n \times n^2)$
- This allows us to solve up to $n \approx 16$ as $16^2 \times 2^{16} \approx 17M$

6. Traveling Salesman Problem (TSP)

- There are only $O(n \times 2^n)$ distinct states because there are n cities and we remember up to 2^n other cities that have been visited in each tour
- Each state can be computed in $O(n)$, thus the overall time complexity of this DP solution is$O(2^n \times n^2)$
- This allows us to solve up to $n \approx 16$ as $16^2 \times 2^{16} \approx 17M$

This is not a huge improvement over the brute force solution but if the programming contest problem involving TSP has input size $11 \leq n \leq 16$, then DP is the solution, not brute force

CPCFI

Index

CPCFI

Table of Contents

DP Non-Classical Problems

- The classical DP problems in their pure forms usually never appear in modern ICPCs
- We'll discuss two more non-classical problems
  1. UVa 10943 - How do you add?
  2. UVa 10003 - Cutting Sticks

CPCFI

1. UVa 10943 - How do you add?

- UVa 10943 - How do you add?
- `C++ code: ch3_10_UVa10943.cpp`

Problem description

Given an integer n, how many ways can K non-negative integers less than or equal to n add up to n? Constraints: $1 \leq n, K \leq 100$

CPCFI

1. UVa 10943 - How do you add?

- UVa 10943 - How do you add?
- `C++ code: ch3_10_UVa10943.cpp`

Problem description

Given an integer n, how many ways can K non-negative integers less than or equal to n add up to n? Constraints: $1 \leq n, K \leq 100$

For example, for $n = 20$ and $K = 2$, there are 21 ways: $0 + 20, 1 + 19, 2 + 18, 3 + 17, \ldots, 20 + 0$

1. UVa 10943 - How do you add?

- The number of ways can be expressed as $\binom{n+k-1}{k-1}$ (Binomial Coefficient)
- However, we can solve this problem using DP techniques: parameters and transitions from one state to another given the base case or cases

1. UVa 10943 - How do you add?

- Parameters: tuple (n, K)
- **Base case**, when $K = 1$, there is only one number less than or equal to n to get n: n itself
- **General case**, at state (n, K) where $K > 1$, we can split n into $X \in [0, \ldots, n]$ and $n - X$, thus $n = X + (n - X)$
    - By doing this, we arrive at the sub-problem $(n - X, K - 1)$ which translates into "given a number $n - X$, how many ways can $K - 1$ numbers less than or equal to $n - X$ add up to $n - X$?"
    - We can sum all these ways

CPCFI

1. UVa 10943 - How do you add?

Complete Search recurrences for `ways(n,K)`:

1. `ways(n, 1) = 1`, since we can only use one number to add up to n, which is n itself
2. `ways(n, k) =` $\sum_{X=0}^{n}$ `ways(n - X, K - 1)`, sum all possible ways recursively

2. UVa 10003 - Cutting Sticks

- UVa 10003 - Cutting Sticks
- `C++ code: ch3_11_UVa10003.cpp`

Problem description

Given a stick of length $1 \leq l \leq 1000$ and $1 \leq n \leq 50$ cuts to be made to the stick (the cut coordinates, lying in the range $[0, \ldots, l]$, are given). The cost of a cut is determined by the length of the stick to be cut. Your task is to find a cutting sequence so that the overall cost is minimized.

2. UVa 10003 - Cutting Sticks

For example, $l = 100$, $n = 3$, and cut coordinates $A = \{25, 50, 75\}$

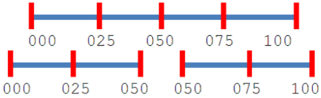
If we cut from left to right, we'll have a total cost of 225:

1. First cut is at coordinate 25 (at original stick of length $l = 100$), total cost so far $= 100$
2. Second cut is at coordinate 50 (at stick with length 75), total cost so far $= 100 + 75 = 175$
3. Third cut is at coordinate 75 (at stick with length 50), final total cost $= 175 + 50 = 225$

2. UVa 10003 - Cutting Sticks

However, the optimal answer is 200:

1. First cut is at coordinate 50, total cost so far $= 100$
2. Second cut is at coordinate 25, total cost so far $= 100 + 50 = 150$
3. Third cut is at coordinate 75, final total cost $= 150 + 50 = 200$



2. UVa 10003 - Cutting Sticks

One possible solution

- Add two more coordinates to $A = \{0, \text{original A}, l\}$ so that we can denote later a stick by the indices of its endpoints in A

[3] left and right are the indices of the current stick with respect to A

2. UVa 10003 - Cutting Sticks

One possible solution

- Add two more coordinates to $A = \{0, \text{original A}, l\}$ so that we can denote later a stick by the indices of its endpoints in A
- Define function `cut(left, right)` that will return the cost of cutting at indices[3] `left` and `right`

[3] left and right are the indices of the current stick with respect to A

CPCFI

2. UVa 10003 - Cutting Sticks

One possible solution

- Add two more coordinates to $A = \{0, \text{original A}, l\}$ so that we can denote later a stick by the indices of its endpoints in A
- Define function `cut(left, right)` that will return the cost of cutting at indices[3] `left` and `right`
- Originally, the stick is described by `left = 0` and `right =` n+1

[3] left and right are the indices of the current stick with respect to A

2. UVa 10003 - Cutting Sticks

Complete Search recurrences:

1. `cut(i-1, i) = 0`, $\forall i \in [1, \ldots, n+1]$, this stick segment does not need to be divided further since `left + 1 = right`
2. `cut(left,right) = min(cut(left,i) + cut(i,right) + (A[right]-A[left]))`, $\forall i \in$ [`left+1` , . . . , `right-1`], try all possible cutting points and pick the best (minimum)

*The cost of a cut is the length of the current stick captured in `(A[right]-A[left])`

Index

Table of Contents

DP in Programming Contests

Summary of the six classic DP problems:

	1D RSQ	2D RSQ	LIS	Knapsack	CC	TSP
State	`(i)`	`(i,j)`	`(i)`	`(id,remW)`	`(v)`	`(pos,mask)`
Space	$O(n)$	$O(n^2)$	$O(n)$	$O(nS)$	$O(V)$	$O(n2^n)$
Transition	subarray	submatrix	all $j < i$	take/ignore	all n coins	all n cities
Time	$O(1)$	$O(1)$	$O(n^2)$	$O(nS)$	$O(nV)$	$O(2^n n^2)$

DP in Programming Contests

- In addition to studying the non-classical examples, please check Top Coder for more DP tutorials
- Mastering Dynamic Programming problems is now a basic requirement
- Art of DP: determining the states and knowing how to fill up the DP table (either TP or BU)

CPCFI

Index

CPCFI

Table of Contents

CPCFI

DP - Problems

- UVa - DP problems
- If UVa's site is not available, please check [Halim]: page 115-117 for DP problems
- For problems' PDF's, please check here

CPCFI

References

Halim S., Halim F., *Competitive Programming 3*, Handbook for ACM ICPC and IOI Contestants. 2013

Skiena S. *The Algorithm Design Manual*. Springer. 2020