5: Dynamic Programming - More techniques

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

Combinatronics

Combinatronics *is a branch of Discrete Mathematics concerned with the study of* countable *discrete structures, such as integers, graphs/trees or logic*

Combinatronics

Problems involving combinatronics are usually entitled:

- How Many?
- Count this...
- etc

Combinatronics

- The solution code is usually small but deriving the formula takes time
- If you encounter one of this type of problems, ask the team member who is stronger in math to derive the formula

Combinatronics

We'll look in detail into three combinatronics problems:

1. Fibonacci Numbers
2. Binomial Coefficients
3. Catalan Numbers

Fibonacci Numbers

Lenoardo Fibonacci's numbers definition:

$$\text{fib(n)} = \begin{cases} 0 & \text{for } n = 0 \\ 1 & \text{for } n = 1 \\ \text{fib(n-1)} + \text{fib(n-2)} & \text{for } n \geq 2 \end{cases}$$

Numbers:

$$0, 1, 1, 2, 3, 5, 8, 21, 34, 55, 89, \ldots$$

Fibonacci Numbers - Properties

Theorem (Zeckendorf theorem)

Every positive integer can be written in a unique way as a sum of one or more distinct Fibonacci numbers such that the sum does not include any two consecutive Fibonacci numbers

Fibonacci Numbers - Properties

For example, to find a representation that satisfies Zeckendorf's theorem (1), we could implement a greedy algorithm that chooses the largest possible Fibonacci number on each step:

$$100 = 89 + 8 + 3$$

$$77 = 55 + 21 + 1$$

$$18 = 13 + 5$$

Fibonacci Numbers - Properties

Theorem (Pisano Period)

The last one/last two/last three/last four digit(s) of a Fibonacci number repeats with a period of 60/300/1500/15000 *respectively*

Fibonacci Numbers

- Some problems might require the use of Fibonacci's numbers while not stating explicitly the word *Fibonacci*
- Possible implementations:
  - Recurrence: very slow
  - $O(n)$ DP technique
  - $O(\log n)$ using the matrix power: recommended when n is large
  - $O(1)$ using Binet's formula: recommended when n is not too large

Fibonacci Numbers

Binet's formula:

$$\mathrm{fib(n)} = \frac{\Phi^n - (-\Phi)^{-n}}{\sqrt{5}}$$

where Φ is the golden ratio:

$$\Phi = (1 + \sqrt{5})/2 \approx 1.618$$

Binomial Coefficients

- $C(n,k)$, ${}^{n}C_{k}$, $\binom{n}{k}$
- Problem: find the coefficients of the algebraic expansion of powers of a binomial
- These coefficients are the number of ways of choosing k unordered elements from n items

$$C(n,k) = \frac{n!}{k! \cdot (n-k)!} \tag{1}$$

- Computing $C(n,k)$ when n and k are large can be quite difficult

Binomial Coefficients

For example,

$$(x+y)^3 = 1x^3 + 3x^2y + 3xy^2 + 1y^3$$

$\{1, 3, 3, 1\}$ are the binomial coefficients of $n = 3$ with $k = \{0, 1, 2, 3\}$ respectively

In other words, the number of ways of choosing $k = \{0, 1, 2, 3\}$ elements from $n = 3$ items

Binomial Coefficients

For example, in how many ways can your select 4 elements from a set of 10 items?

$$\binom{n}{k} = \binom{10}{4} = \frac{10!}{4! \cdot (10-4)!}$$

Binomial Coefficients

- We can compute $C(n, k)$ with equation 1, however, it can be a challenge when n or k are large values

Binomial Coefficients

- We can compute $C(n,k)$ with equation 1, however, it can be a challenge when n or k are large values
- If we have to compute many but not all values of $C(n,k)$ for different n and k, it is better to use top-down Dynamic Programming:
    - $C(n,0) = C(n,n) = 1$
    - $C(n,k) = C(n-1,k-1) + C(n-1,k)$

Binomial Coefficients

- We can compute $C(n, k)$ with equation 1, however, it can be a challenge when n or k are large values
- If we have to compute many but not all values of $C(n, k)$ for different n and k, it is better to use top-down Dynamic Programming:
  - $C(n, 0) = C(n, n) = 1$
  - $C(n, k) = C(n-1, k-1) + C(n-1, k)$
- If we have to compute all values of $C(n, k)$ from $n = 0$ up to a certain value of n, we can build Pascal's Triangle

```
n = 0            1
n = 1          1   1
n = 2        1   2   1
n = 3      1   3   3   1  <- as shown above
            \ / \ / \ /
n = 4    1   4   6   4   1 ... and so on
```

Catalan Numbers

The n-th Catalan number is defined as:

$$\mathrm{Cat}(n) = \frac{C\binom{2n}{n}}{n+1}$$

$\mathrm{Cat}(0) = 1$

Catalan Numbers

For example,

$$\mathrm{Cat}(3) = \frac{C\binom{2\cdot 3}{3}}{3+1} = \frac{20}{4} = 5$$

Catalan Numbers

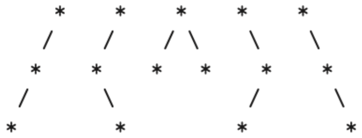
If we were asked to compute n values of Catalan numbers we could use a bottom-up Dynamic Programming approach: if we know $\mathrm{Cat}(n)$ we could compute $\mathrm{Cat}(n+1)$ as follows:

$$\mathrm{Cat}(n+1) = \frac{(2n+2)\cdot(2n+1)}{(n+2)\cdot(n+1)} \cdot \mathrm{Cat}(n)$$

Catalan Numbers

Catalan numbers are found in various combinatorial problems:

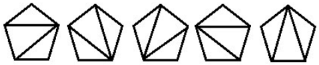
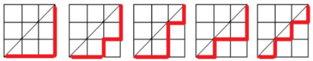
1. Cat(n) counts the number of distinct binary trees with n vertices. For example, $n = 3$



2. Cat(n) counts the number of expressions containing n pairs of parentheses which are correctly matched. For example, $n = 3$:
()()(), ()(()), (())(), ((())), (()())

Catalan Numbers

3. Cat(n) counts the number of ways a convex polygon of $n + 2$ sides can be triangulated. A convex polygon is one in which all of its internal angles are equal or less than 180° (left image)
4. Cat(n) counts the number of monotonic paths along the edges of an n$\times n$ grid, which do not pass above the diagonal. A monotonic path is one which starts in the lower left corner, finishes in the upper right corner, and consists entirely of edges pointing rightwards or upwards (right image)

Combinatronics in Programming Contests

- In online programming contests, if Internet access is allowed, one can generate small instances of a sequence and then search that sequence in the Online Encyclopedia of Integer Sequences (OEIS): http://oeis.org/

Combinatronics Problems

- UVa - Combinatronics Problems
- If site is down, please check [Halim]: page 207-209 and browse for each problem at CP Book's Method to Solve

Table of Contents

Probability Theory

> *Branch of mathematics dealing with the analysis of random phenomena*

For certain random phenomena (such as toin cossing), the sequence of random events will exhibit -in the long term- a certain statistical pattern. For example, the probability of a head in a toss coin -in the long term- will be $1/2$

Probability Theory

In programming contests, problems involving probability will be solvable with:

- **Closed-form formula**: for these type of problems, we need to derive a formula (usually $O(1)$)
- **Exploration of the search space to count the number of events over the countable sample space**: these type of problems may involve some sort of Combinatronics, Complete Search of Dynamic Programming

Probability Theory Problems

- UVa - Probability Theory
- If site is down, please check [Halim]: page 222 and then browse for the problem's PDF here

Table of Contents

String Processing with Dynamic Programming

Now, we'll study some string processing problems which solution can be found using Dynamic Programming techniques previously seen.

Classical string processing problems every competitive programmer must know:

1. String Alignment
2. Longest Common Subsequence

After studying these two problems, we'll study more problems that contain slight modifications of these two.

String Processing with Dynamic Programming

Note: for various DP problems on string, we usually manipulate the integer indices of the strings and not the actual strings (or substrings) themselves. Passing substrings as parameters is very slow and hard to memoize

String Alignment (Edit Distance)

Problem description

Align two strings A with B with the maximum alignment score (or minimum number of edit operations):

1. Character $A[i]$ and $B[i]$ match: **+2 points**
2. Character$A[i]$ and $B[i]$ mismatch and we replace $A[i]$ with $B[i]$: **-1 point**
3. We insert a space in $A[i]$: **-1 point**
4. We delete a letter from $A[i]$: **-1 point**

- `C++ code: ch6_03_str_align.cpp`

String Alignment (Edit Distance)

For example (non optimal alignment score),

A = 'ACAATCC' $\rightarrow$ A_CAATCC
B = 'AGCATGC' $\rightarrow$ AGCATGC_
Alignment score $= 4 * 2 + 4 * (-1) = 4$

String Alignment (Edit Distance)

The solution for this problem is the Needleman-Wunsch's Bottom Up DP algorithm $O(nm)$:

- Consider two strings `A[1..n]` and `B[1..i]`
- $V(i, j)$: score of the optimal alignment between prefix `A[1..i]` and `B[1..j]`
- $score(C1, C2)$: function that returns the score if the character $C1$ is aligned with character $C2$

String Alignment (Edit Distance)

We define the following recurrences:

Bases cases:

- $V(0,0) = 0$: no score for matching two empty strings

String Alignment (Edit Distance)

We define the following recurrences:

Bases cases:

- $V(0,0) = 0$: no score for matching two empty strings
- $V(i,0) = i \times \text{score}(A[i], \_)$: '_' represents an empty space, delete substring `A[1..i]` to make the alignment $i > 0$

String Alignment (Edit Distance)

We define the following recurrences:

Bases cases:

- $V(0,0) = 0$: no score for matching two empty strings
- $V(i,0) = i \times \text{score}(A[i], \_)$: '_' represents an empty space, delete substring `A[1..i]` to make the alignment $i > 0$
- $V(0,j) = j \times \text{score}(\_, B[j])$: insert spaces `B[1..j]` to make the alignment $i > 0$

String Alignment (Edit Distance)

We define the following recurrences:

Bases cases:

- $V(0,0) = 0$: no score for matching two empty strings
- $V(i,0) = i \times \text{score}(A[i], \_)$: '_' represents an empty space, delete substring `A[1..i]` to make the alignment $i > 0$
- $V(0,j) = j \times \text{score}(\_, B[j])$: insert spaces `B[1..j]` to make the alignment $i > 0$

Recurrences for $i > 0$ and $j > 0$:

String Alignment (Edit Distance)

We define the following recurrences:

Bases cases:

- $V(0, 0) = 0$: no score for matching two empty strings
- $V(i, 0) = i \times \text{score}(A[i], \_)$: '_' represents an empty space, delete substring `A[1..i]` to make the alignment $i > 0$
- $V(0, j) = j \times \text{score}(\_, B[j])$: insert spaces `B[1..j]` to make the alignment $i > 0$

Recurrences for $i > 0$ and $j > 0$:

- $V(i, j) = \max(\text{option1}, \text{option2}, \text{option3})$ where

String Alignment (Edit Distance)

We define the following recurrences:

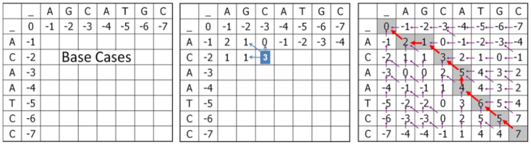
Bases cases:

- $V(0,0) = 0$: no score for matching two empty strings
- $V(i,0) = i \times \text{score}(A[i], \_)$: '_' represents an empty space, delete substring `A[1..i]` to make the alignment $i > 0$
- $V(0,j) = j \times \text{score}(\_, B[j])$: insert spaces `B[1..j]` to make the alignment $i > 0$

Recurrences for $i > 0$ and $j > 0$:

- $V(i,j) = \max(\text{option1}, \text{option2}, \text{option3})$ where
- $\text{option1} = V(i-1, j-1) + \text{score}(A[i], B[j])$: score of match or mismatch
- $\text{option2} = V(i-1, j) + \text{score}(A[i], \_)$: delete A_i
- $\text{option3} = V(i, j-1) + \text{score}(\_, B[j])$: insert B_j

String Alignment (Edit Distance)



	_	A	G	C	A	T	G	C
_	0	-1	-2	-3	-4	-5	-6	-7
A	-1							
C	-2	Base Cases						
A	-3							
A	-4							
T	-5							
C	-6							
C	-7							

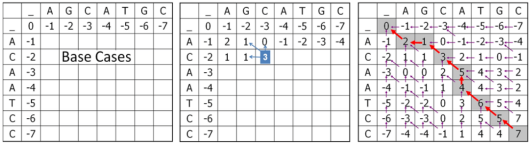
	_	A	G	C	A	T	G	C
_	0	-1	-2	-3	-4	-5	-6	-7
A	-1	2	1	0	-1	-2	-3	-4
C	-2	1	1	3				
A	-3							
A	-4							
T	-5							
C	-6							
C	-7							

	_	A	G	C	A	T	G	C
_	0	-1	-2	-3	-4	-5	-6	-7
A	-1	2	1	0	-1	-2	-3	-4
C	-2	1	1	3	2	1	0	-1
A	-3	0	0	2	5	4	3	2
A	-4	-1	-1	1	4	4	3	2
T	-5	-2	-2	0	3	6	5	4
C	-6	-3	-3	0	2	5	5	7
C	-7	-4	-4	-1	1	4	4	7

Figure 6.1: Example: `A = 'ACAATCC'` and `B = 'AGCATGC'` (alignment score = 7)

- Initially, only the base cases are known
- Then, we can fill the values row by row, left to right. To fill in $V(i,j)$for$i,j > 0$, we just need three other values: $V(i-1,j-1)$, $V(i-1,j)$, and $V(i,j-1)$

String Alignment (Edit Distance)



	_	A	G	C	A	T	G	C
_	0	-1	-2	-3	-4	-5	-6	-7
A	-1							
C	-2		Base Cases					
A	-3							
A	-4							
T	-5							
C	-6							
C	-7							

	_	A	G	C	A	T	G	C
_	0	-1	-2	-3	-4	-5	-6	-7
A	-1	2	1	0	-1	-2	-3	-4
C	-2	1	1	3				
A	-3							
A	-4							
T	-5							
C	-6							
C	-7							

	_	A	G	C	A	T	G	C
_	0	-1	-2	-3	-4	-5	-6	-7
A	-1	2	1	0	-1	-2	-3	-4
C	-2	1	1	3	2	1	0	-1
A	-3	0	0	2	5	4	3	2
A	-4	-1	-1	1	4	4	3	2
T	-5	-2	-2	0	3	6	5	4
C	-6	-3	-3	0	2	5	5	7
C	-7	-4	-4	-1	1	4	4	7

Figure 6.1: Example: `A = ‘ACAATCC’` and `B = ‘AGCATGC’` (alignment score = 7)

- Initially, only the base cases are known
- Then, we can fill the values row by row, left to right. To fill in $V(i,j)$for$i,j > 0$, we just need three other values: $V(i-1,j-1)$, $V(i-1,j)$, and $V(i,j-1)$

Optimal alignment:

`A = ’A\_CAAT[C]C’`

`B = ’AGC\_AT[G]C’`

Alignment score $= 5 * 2 + 3 * (-1) = 7$

Longest Common Subsequence

Problem description

Given two strings A and B, determine the longest common subsequence between them

Longest Common Subsequence

Problem description

Given two strings A and B, determine the longest common subsequence between them

For example, `A = ‘ACAATCC’` and `B = ‘AGCATGC’` have LCS of length 5, i.e. `‘ACATC’`

Longest Common Subsequence

This LCS problem can be reduced to the String Alignment problem by setting the cost for mismatch as negative infinity, cost for insertion and deletion as 0, and the cost for match as 1. This makes the Needleman-Wunsch's algorithm for String Alignment to never consider mismatches.

Non Classical String Processing with DP

- UVa 11151 - Longest Palindrome

Problem description
Given a string of up to $n = 1000$ characters, determine the length of the longest palindrome that you can make from it by deleting zero or more characters

Non Classical String Processing with DP

- UVa 11151 - Longest Palindrome

Problem description
Given a string of up to $n = 1000$ characters, determine the length of the longest palindrome that you can make from it by deleting zero or more characters

For example:

- `ADAM` $\rightarrow$ `ADA`: length 3, delete `M`
- `MADAM` $\rightarrow$ `MADAM`: length 5, delete nothing

UVa 11151

DP solution $O(n^2)$: let $len(l, r)$ be the length of the longest palindrome from string `A[l..r]`

UVa 11151

DP solution $O(n^2)$: let $len(l, r)$ be the length of the longest palindrome from string `A[l..r]`

Base cases:

- If `l == r`, then `len(l,r) = 1`, odd-length palindrome
- If `l + 1 == r`:
  - If `A[l] == A[r]`, then `len(l,r) = 2`
  - Otherwise, `1`, even-length palindrome

UVa 11151

DP solution $O(n^2)$: let $len(l, r)$ be the length of the longest palindrome from string `A[l..r]`

Base cases:

- If `l == r`, then `len(l,r) = 1`, odd-length palindrome
- If `l + 1 == r`:
    - If `A[l] == A[r]`, then `len(l,r) = 2`
    - Otherwise, 1, even-length palindrome

Recurrences:

- If `A[l] == A[r]`, then `len(l,r) = 2 + len(l+1, r-1)`, both corner characters are the same
- Otherwise, `len(l,r) = max(len(l,r-1), len(l+1,r))`, increase left side or decrease right side

Problems

- UVa - String Processing with DP: Classic
- UVa - String Processing with DP: Non Classic
- If site is down, please check [Halim]: page 248

Table of Contents

More Advanced DP Techniques

Now, we'll look into more advanced DP techniques:

- DP with Bitmask
- Compilation of common DP parameters
- Offset Technique
- Balanced BST as Memo Table

8.3.1

DP with Bitmask

DP with Bitmask

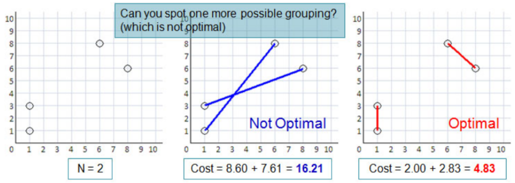
- Some of the modern DP problems require a (small) set of Boolean as one of the parameters of the DP state
- Bitmask technique can be useful as one of the parameters in the DP table

UVa 10911 - Forming Quiz Teams

- `C++ code: ch8_02_10911.cpp`

Problem description

Let (x, y) be the coordinates of a student's house on a 2D plane. There are $2N$ students and we want to pair them into N groups. Let d_i be the distance between the houses of 2 students in group i. Form N groups such that $cost = \sum_{i=1}^{N} d_i$ is minimized. Output the minimum cost. Constraints: $1 \leq N \leq 8$ and $0 \leq x, y \leq 1000$



UVa 10911 - Forming Quiz Teams

*This problem is formally called: "minimum weight perfect matching on a small generated weighted graph". This is a **hard** problem but if $M \leq 20$ then DP bitmask solution can be used

UVa 10911 - Forming Quiz Teams

For example, when $M = 6$, $(M = 2N)$:

- We start with a state where nothing is matched (`bitmask = 000000`)

UVa 10911 - Forming Quiz Teams

For example, when $M = 6$, $(M = 2N)$:

- We start with a state where nothing is matched (`bitmask = 000000`)
- If item 0 and item 2 are matched, we can turn on two bits (bit 0 and bit 2), thus the state becomes `bitmask = 000101`

UVa 10911 - Forming Quiz Teams

For example, when $M = 6$, $(M = 2N)$:

- We start with a state where nothing is matched (`bitmask = 000000`)
- If item 0 and item 2 are matched, we can turn on two bits (bit 0 and bit 2), thus the state becomes `bitmask = 000101`
- If from this state, item 1 and item 5 are matched next, the state will become `bitmask = 100111`

UVa 10911 - Forming Quiz Teams

For example, when $M = 6$, ($M = 2N$):

- We start with a state where nothing is matched (`bitmask = 000000`)
- If item 0 and item 2 are matched, we can turn on two bits (bit 0 and bit 2), thus the state becomes `bitmask = 000101`
- If from this state, item 1 and item 5 are matched next, the state will become `bitmask = 100111`
- The perfect matching is obtained when the state is all '1's, in this case: `bitmask=111111`

UVa 10911 - Forming Quiz Teams

Solution

- There are only $O(2^M)$ distinct states. (Each number in the bitmask is a different state)

UVa 10911 - Forming Quiz Teams

Solution

- There are only $O(2^M)$ distinct states. (Each number in the bitmask is a different state)
- For each state, we store the minimum weight of previous matchings that must be done in order to reach this state

UVa 10911 - Forming Quiz Teams

Solution

- There are only $O(2^M)$ distinct states. (Each number in the bitmask is a different state)
- For each state, we store the minimum weight of previous matchings that must be done in order to reach this state
- First, we find one 'off' bit i using one $O(M)$ loop. Then, we find the best other 'off' bit j from `[i+1..M-1]` using another $O(M)$ loop and recursively match i and j

UVa 10911 - Forming Quiz Teams

Solution

- There are only $O(2^M)$ distinct states. (Each number in the bitmask is a different state)
- For each state, we store the minimum weight of previous matchings that must be done in order to reach this state
- First, we find one 'off' bit i using one $O(M)$ loop. Then, we find the best other 'off' bit j from `[i+1..M-1]` using another $O(M)$ loop and recursively match i and j
- This algorithm runs in $O(M \times 2^M)$

8.3.2

Compilation of Common DP Parameters

Compilation of Common DP Parameters

1. **Parameter:** Index i in array $[x_0, x_1, \ldots, x_i, \ldots]$

8.3.3

Handling Negative Parameter Values with Offset Technique

Handling Negative Parameter Values with Offset Technique

- In some cases, the DP parameter can go negative which causes an issue since most of the times we use this parameter as index in a DP table
- This can be solved using the **offset** technique

Handling Negative Parameter Values with Offset Technique

- UVa 1238 - Free Parentheses

Problem description

Given an arithmetic expression which consists of only addition and subtraction operators, put any parentheses to the expression (as long as it's still valid), **how many different numbers can you make?**

UVa 1238 - Free Parentheses

For example, $1 - 2 + 3 - 4 - 5$:

- $1 - 2 + 3 - 4 - 5 = -7$
- $1 - (2 + 3) - 4 - 5 = -13$
- $1 - (2 + 3 - 4) - 5 = -5$
- $1 - (2 + 3 - 4 - 5) = 5$
- $1 - 2 + 3 - (4 - 5) = 3$
- $1 - (2 + 3) - (4 - 5) = -3$

UVa 1238 - Free Parentheses

The expression consists of only $2 \leq N \leq 30$ non-negative numbers less than 100 separated by addition or subtraction operators and there is no operator before the first and after the last number.

Let's look at the following remarks:

- We only need to put a parentheses after a subtraction operator '-'
- We can only put X closing parentheses if we have used X open parentheses
- Maximum value: $100 + 100 + \ldots + 100 = 3000$
- Minimum value: $0 - 100 - \ldots - 100 = -2900$

UVa 1238 - Free Parentheses

Solution

DP parameters:

1. `idx`: current position in the expression being processed
2. `open`: number of open parentheses (we need this parameter to build a valid expression)
3. `val`: total value of the current expression

UVa 1238 - Free Parentheses

Solution

DP parameters:

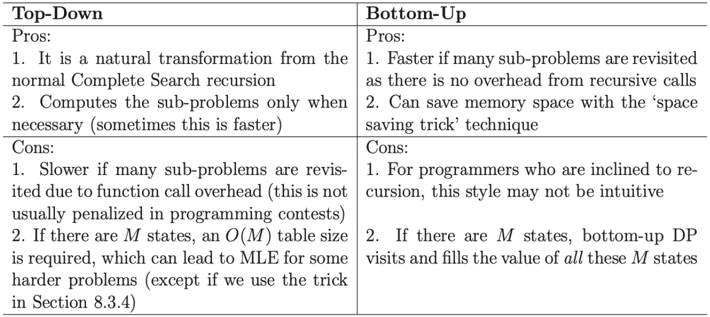
1. `idx`: current position in the expression being processed
2. `open`: number of open parentheses (we need this parameter to build a valid expression)
3. `val`: total value of the current expression

*The memo table will consist of a 3D array `visited[idx][open][idx+3000]` where the offset technique is represented by adding the maximum value to the current index `idx`

8.3.4
MLE? Balanced BST as a Memo Table

MLE? Balanced BST as a Memo Table

Let's revisit the differences between Top-Down DP approach and 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

MLE? Balanced BST as a Memo Table

- In previous slides, we've seen the Knapsack 0-1 problem where the state is `(id, remW)`
- Parameter `id` has a range of `[0..n-1]` and parameter `remW` a range of `[0..S]`
- If the problem states that $n \times S$ is too large, the DP table of size $n \times S$ might result in a MLE (Memory Limit Exceeded) verdict

MLE? Balanced BST as a Memo Table

- If we run a Top-Down DP on this problem, not all of the states will be visited (in comparison to a Bottom-Up approach)
- We can trade runtime for smaller space by using a balanced BST (`C++ STL map`) as the memo table
- This balanced BST will only store the states visited by the Top-Down DP

Thus, if there are only k visited states, we'll only use $O(k)$ space instead of $O(n \times S)$

8.3.5
MLE/TLE? Use Better State Representation

MLE/TLE? Use Better State Representation

If we coded a correct DP solution but the OJ still gives a MLE or TLE, we have no option but to use a better DP state representation in order to reduce the DP table size and speed the overall time complexity

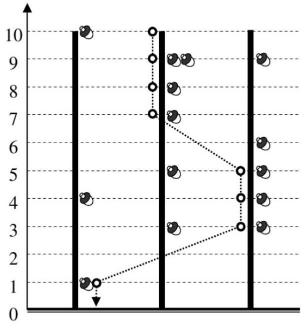
MLE/TLE? Use Better State Representation

- UVa 1231 - ACORN
- `C++ code: ch8_03_UVa1231.cpp`

Problem description
Given t oak trees, the height h of **all** trees, the height f that Jayjay the squirrel loses when it flies from one tree to another, $1 \leq t, h \leq 2000$, $1 \leq f \leq 500$, and the positions of acorns on each of the oak trees: `acorn[tree][height]`, determine the max number of acorns that Jayjay can collect in one single descent.

UVa 1231 - ACORN

For example, $t = 3$, $h = 10$, $f = 2$, the best descent has a total of 8 acorns:



UVa 1231 - ACORN

Naive DP Solution:

- Use a table `total[tree][height]` that stores the best possible acorns collected when Jayjay is on a certain tree at certain height
- Jayjay recursively tries to either go down (-1) unit on the same oak tree or flies $(-f)$ unit(s) to $t-1$ other oak trees from this position
- On the largest test case, this requires $2000 \times 2000 = 4M$ states and $4M \times 2000 = 8B$ operations

*This approach is clearly TLE

UVa 1231 - ACORN

Best DP Solution:

- We can actually ignore the information: “On which tree Jayjay is currently at” as just memoizing the best among them is sufficient

UVa 1231 - ACORN

Best DP Solution:

- We can actually ignore the information: "On which tree Jayjay is currently at" as just memoizing the best among them is sufficient
- This is because flying to any other $t - 1$ other oak trees decreases Jayjay's height in the same manner

UVa 1231 - ACORN

Best DP Solution:

- We can actually ignore the information: "On which tree Jayjay is currently at" as just memoizing the best among them is sufficient
- This is because flying to any other $t-1$ other oak trees decreases Jayjay's height in the same manner
- `dp[height]` will store the best possible acorns collected when Jayjay is at this height
- This Bottom-Up DP takes $2000 = 2K$ states and time complexity of $2000 \times 2000 = 4M$

`*C++ code ch8_03_UVa1231.cpp`

UVa 1231 - ACORN

When the size of naive DP states are too large that causes the overall DP time complexity to be not doable, think of another more efficient (but usually not obvious) way to represent the possible states

8.3.6
MLE/TLE? Drop One Parameter, Recover It From Others

MLE/TLE? Drop One Parameter, Recover It From Others

Another trick to reduce the memory space and speed up the solution is to drop one important parameter which can be recovered by using the other parameter(s)

MLE/TLE? Drop One Parameter, Recover It From Others

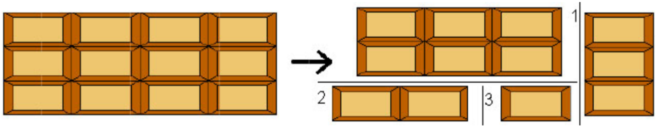
- UVa 1099 - Sharing Chocolate

Problem Description

Given a big chocolate bar of size $1 \leq w, h \leq 100, 1 \leq n \leq 15$ friends, and the size request of each friend. Can we break the chocolate by using horizontal and vertical cuts so that each friend gets one piece of chocolate bar of his chosen size?

UVa 1099 - Sharing Chocolate

For example, the size of the original chocolate bar is $w = 4$ and $h = 3$. If there are $n = 4$ friends, each requesting a chocolate piece of size $\{6, 3, 2, 1\}$, respectively, then we can break the chocolate into 4 parts using 3 cuts:



UVa 1099 - Sharing Chocolate

One possible state representation

- State: `(w, h, bitmask)`, where `bitmask` is the subset of friends that already have a chocolate piece of their chosen size
- However, to implement this solution we'll need a DP table of size $100 \times 2^15 = 327M$ which is **infeasible**

UVa 1099 - Sharing Chocolate

Better state representation

- Use only two parameters, either `(w,bitmask)` or `(h, bitmask)`

UVa 1099 - Sharing Chocolate

Better state representation

- Use only two parameters, either `(w,bitmask)` or `(h, bitmask)`
- If we use `(w,bitmask)`:
    - `h = sum(bitmask) / w`

UVa 1099 - Sharing Chocolate

Better state representation

- Use only two parameters, either `(w,bitmask)` or `(h, bitmask)`
- If we use `(w,bitmask)`:
  - `h = sum(bitmask) / w`
- DP table of size: $100 \times 2^15 = 3M$ which is **reasonable**

UVa 1099 - Sharing Chocolate

Base cases

- If `bitmask` only contains 1 'on' bit and the requested chocolate size of that person equals to $w \times h$, we have a solution
- Otherwise we do not have a solution.

General case

- If we have a chocolate piece of size $w \times h$ and a current set of satisfied friends `bitmask = bitmask1` U `bitmask2`
    - We can do either horizontal or vertical cut so that one piece is to serve friends in `bitmask1` and the other is to serve friends in `bitmask2`

UVa - Problems

- UVa - More Advanced DP Techniques
- If site is down, please refer to [Halim]: page 318-319 and browse for the PDF's here

References

- Halim S., Halim F., *Competitive Programming 3*, Handbook for ACM ICPC and IOI Contestants. 2013
- Skiena S. *The Algorithm Design Manual*. Springer. 2020