Competitive Programming 2

Week 3: Dynamic Programming 1

Announcement

- ICPC Qualifiers Contest -- next Wednesday at LWSN B151 8pm

Dynamic Programming

“Method for solving complex problems by breaking them down into simpler subproblems” - Wikipedia

- Historical term from the 40s and 50s
    - programming: creating a schedule for tasks
    - dynamic: used on time-varying problems

Example 1 - Climbing stairs

You are climbing a staircase.

It takes n steps to reach to the top.

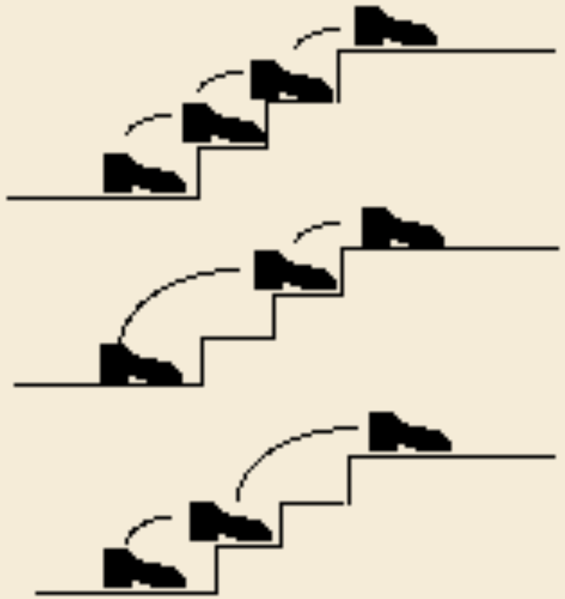
Each time you can either climb 1 or 2 steps. In how many distinct ways can you climb to the top?

Example 1 - Climbing stairs

- Input: n = 1
  - Output: 1
  - There is only one way to climb 1 stair

- Input: n = 2
  - Output: 2
  - There are two ways: (1, 1) and (2)

- Input: n = 4
  - Output: 5
  - (1, 1, 1, 1), (1, 1, 2), (2, 1, 1), (1, 2, 1), (2, 2)

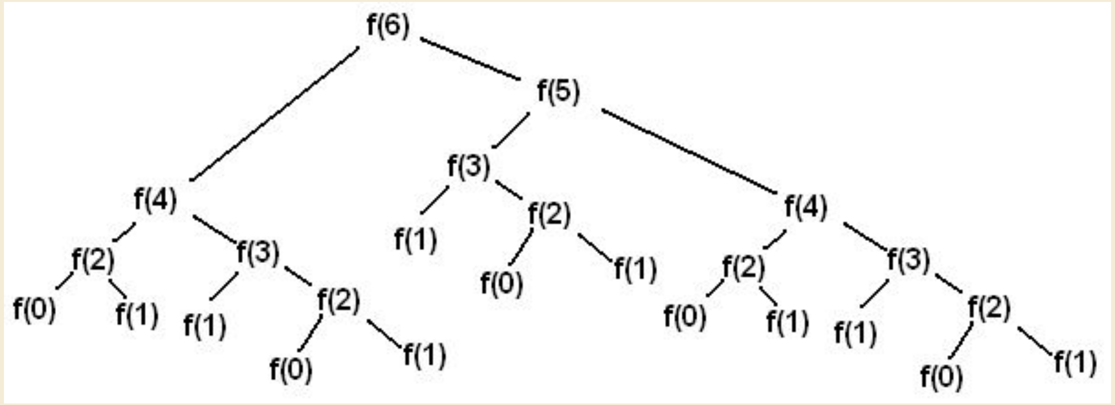


Example 1 - Climbing stairs

We can model this recursively.

The # of ways to climb n steps is equal to the number of ways to climb n - 1 steps (where we then climb one step) plus the ways of climbing n - 2 steps (where we then climb 2 steps)

So, f(n) = f(n - 1) + f(n - 2)



Example 1 - Climbing stairs

Basic definition using recursion:

```
int f(int n) {
        if (n <= 2) return n;
        return f(n - 1) + f(n - 2);
}
```

Problem: we recalculate multiple values!

So What is Dynamic Programming?

Break problem down into subproblems

Useful when problem has...

- optimal substructure: optimal overall solution can be found by combining optimal solutions to subproblems (think: recursive definition)
- subproblems are overlapping: a recursive solution will be faced with the same subproblems over and over (think: computing Fibonacci numbers)

Idea: solve each subproblem only once

Steps for Dynamic Programming

1. Define subproblems (states)

2. Define the recurrence relationship that connects subproblems
    a. transfer equation
3. Recognize and solve the base cases
    a. Set up initial values
4. Look for the target

◭ Each step is very important!

Steps for Dynamic Programming

Top-down: use recursion, but

- Remember already-computed subproblems
- "Memoization table" maps arguments to solutions
- On each recursive call:
  - Check memoization table first; use it
  - Else compute value and store in table

Bottom-up: compute subproblems first

- Need to figure out what the simplest subproblems are
- Build more complex subproblems on top of simpler subproblems

Example 1 - Climbing stairs

```
int[] dp = new int[n + 1];

Arrays.fill(dp, -1);

return f(dp, n);

int f(int[] dp, int n) {

        if (dp[n] != -1) return dp[n];

        if (n <= 2) return dp[n] = n;

        return dp[n] = f(n - 1) + f(n - 2);

}
```

Top-down (start at n, end at 0):

Calculate values only once, but have a large stack due to repeated recursive calls

Example 1 - Climbing stairs

```
public int f(int n) {

    int[] dp = new int[n + 1];

    dp[0] = 0;      dp[1] = 1;    dp[2] = 2;

    if (n <= 2) return dp[n];

    for(int i = 3; i <= n; i++) {

        dp[i] = dp[i - 1] + dp[i - 2];

    }

    return dp[n];

}
```

Bottom-up (start at 0, end at n):
O(n) space

Note: we never need to calculate the same values multiple times

Example 2 - Maximum Sum Subarray

Given an array A of non-zero integers, find the maximum sum between two indices i and j.

I.e. Given A = [−2, 1, −3, 4, −1, 2, 1, −5, 4], the largest contiguous subarray sum is [4, −1, 2, 1] with sum 6.

Note that if all numbers are positive, we can just return the sum of the entire array.

Example 2 - Maximum Sum Subarray

The naive method is to run two loops. The outer loop picks the beginning element, the inner loop finds the maximum sum with first element picked by outer loop and compares this maximum with the overall maximum. Finally return the overall maximum.

The time complexity of the Naive method is $O(n^2)$.

Example 2 - Maximum Sum Subarray

With some realization, we can divide this up into multiple subproblems!

Viewing this as DP, at each step we have 2 choices. We can either leverage the previously accumulated maximum sum or begin a new range.

The DP variable dp[i] represents the maximum sum of a range of integers that ends with element A[i].

Thus, the final answer is the maximum over all dp[i] for i in [0..n-1].

Example 2 - Maximum Sum Subarray

```
public int maxSubArray(int[] A) {

    int n = A.length, max = max(0, A[0]);

    int[] dp = new int[n + 1];

    dp[0] = A[0];

    for (int i = 1; i < n; i++) {

        if ( dp[i - 1] > 0) dp[i] = max( A[i], dp[i - 1] + A[i]);

        else dp[i] = A[i];

        if (dp[i] > max) max = dp[i];

    }

    return max;
```

Bottom-up (start at 0, end at n):
O(n) space

Note: we never need to calculate the same values multiple times

Example 2 - Maximum Sum Subarray

Even better: **we don’t even need to build the table!**

```
int sum = 0, ans = A[0];

for (int i = 0; i < n; i++) {

    sum += A[i];

    ans = Math.max(ans, sum);

    if (sum < 0) sum = 0;

}

return ans;
```

O(n) time, O(1) space

The intuition here is that we just need to keep a running sum of the integers seen so far, then greedily reset to 0 if the sum dips below 0.

Example 3 - # of Paths in a Matrix

Given m and n, representing the width and height of a matrix, count the number of paths from the top left cell to the bottom right with the constraint that you can only move down + right.

paths(m, 0) = 1

paths(0, n) = 1

paths(m, n) = paths(m - 1, n) + paths(m, n - 1)

Example 3 - # of Paths in a Matrix

```
int numberOfPaths(int m, int n)

{

    // Create a 2D table to store results of subproblems

    int count[m][n];

    // Initialize number paths to reach any cell in first column/row to 1

    for (int i = 0; i < m; i++) count[i][0] = 1;

    for (int j = 0; j < n; j++) count[0][j] = 1;
```

Example 3 - # of Paths in a Matrix

```
    // Calculate number of paths for other cells in bottom-up manner using the recursive dynamic programming equations
    for (int i = 1; i < m; i++)
    {
        for (int j = 1; j < n; j++)
            // Question: how should we change the dynamic programming equation to allow for diagonal movement?
            count[i][j] = count[i-1][j] + count[i][j-1];
    }
    return count[m-1][n-1];
}
```

Example 3 - # of Paths in a Matrix

```
    // Calculate number of paths for other cells in bottom-up manner using the recursive dynamic programming equations
    for (int i = 1; i < m; i++)
    {
        for (int j = 1; j < n; j++)
            // Answer: we need to add in the number of paths from the cell diagonally above the new entry!
            count[i][j] = count[i-1][j] + count[i][j-1];
    }
    return count[m-1][n-1];
}
```

Example 3 - # of Paths in a Matrix

Follow-ups:

- How would you handle the problem if certain cells are blocked off?
- Can you reduce the space requirements?
    - This currently costs O(nm)
    - O(n) or O(m) is possible

Going Forward

Basic DP versions

1. **Parameter: Index i in an array - ex: LIS**
2. **Parameter: Subarray (i, j) of an array - ex: Matrix Chain multiplication**
3. Parameter: Indices (i, j) in two arrays - ex: Edit Distance, LCS
4. Parameter: Knapsack-Style Parameter - ex: Knapsack, Coin change etc…

We will see examples of (3) and (4) in next class

Longest Increasing Subsequence

Given an array of size N, return the **length** of a longest increasing subsequence.

E.x.

0, 8, 4, 12, 2, 10, 6, 14, 1, 9

One of the longest increasing subsequence is 0, 2, 6, 9, so you need to return 4.

Longest Increasing Subsequence

Let f[i] be the length of longest subsequence that ends at position i.

f[i] = max(f[i], f[j] + 1) for 0 <= j < i and A[j] < A[i]

Base state: f[i] = 1 for 0 <= i < N

Answer: max(f[i]) for 0 <= i < N

Longest Increasing Subsequence

```
for i = 0 to n - 1:
    f[i] = 1
for i = 0 to n - 1:
    for j = 0 to i - 1:
        if A[j] < A[i]:
            f[i] = max(f[i], f[j] + 1)
    answer = max(answer, f[i])
```

Longest Increasing Subsequence

- How to print one of the longest increasing subsequence?
  - Backtrace the table!

Matrix Chain multiplication

Given a sequence of matrices, the goal is to find the most efficient way to multiply these matrices.

You don’t have to do the multiplication. You only output the minimum number of simple operations.

E.x.

A.size = (10 x 30)   B.size = (30 x 5)   C.size = (5 x 60)

(AB)C needs $(10\times30\times5) + (10\times5\times60) = 1500 + 3000 = 4500$ operations, while

A(BC) needs $(30\times5\times60) + (10\times30\times60) = 9000 + 18000 = 27000$ operations.

Matrix Chain multiplication

Let f(i, j) be the minimum number of operations after multiplying matrices from i to j.

f(i, j) = min(f(i, j), f(i, k) + f(k + 1, j) + A[i][0] * A[k][1] * A[j][1]), for i <= k < j

Base state: f(i, i) = 0

Answer: f(0, N - 1)

Matrix Chain multiplication

```
for i = 0 to N - 1:
    f(i, i) = 0
for len = 2 to N:
    for i = 0 to N - len:
        j = i + len - 1
        for k = i to j - 1:
            f(i, j) = min(f(i, j), f(i, k) + f(k + 1, j) + A[i][0] * A[k][1] * A[j][1])
```

Additional DP resources

Cool problems: Minimum Path Sum, Leetcode DP Problem List

Beginner Guides:

- Stanford DP Slides
- Topcoder DP

Advanced:

- DP on Trees
- DP on Graphs

Announcements

- Next week: Dynamic Programming 2
- Sign the attendance sheet before you leave!