Dynamic Programming

9/27/2017

These slides will be put on:
github.com/AuburnACM/Competitive-Programming

Agenda

- Announcements
- What is Dynamic Programming?
- Common DP problems
  - Maximum Subsequence
  - Longest Increasing Subsequence
  - Knapsack
  - Egg Drop
- Your Turn!

Announcements - Last Week's Mock

September 24 Mock

Rank	Team	Solved	Time	A	B	C	D	E	F	G	H	I	J
1	Mitch Price	5	336	1 13				1 15	1 166	1 35		4 47 (+60)	
2	Amy Cheng	3	188					1 17		1 49		1 122	
3	Matt Bonsall	3	231	1 -- (+20)				1 17		1 51		4 103 (+60)	
4	Turner Atwood	3	250					1 8		1 22		8 80 (+140)	
5	Chengyu Tang	3	292			1 -- (+20)		5 30 (+80)		1 46		2 116 (+20)	
6	Henry Rice	2	140					1 49		2 71 (+20)		1 -- (+20)	
7	Nirmit Patel	1	137					1 137		3 -- (+60)			
8	Robby March	1	204	13 -- (+260)				7 84 (+120)		1 -- (+20)			
9	Nicholas Tkalych	1	226					8 86 (+140)		8 -- (+160)			
10	Andrew McGehee	0	0					1 -- (+20)					

Last Week's Mock

● A Classy Problem	User - Defined Comparator
● Amazing Race	TSP - like DP $O(N^2 2^N)$
● Bundles of Joy	DPish or Tree Solution
● Flipping Cards	Union Find (Stack Depth Issues)
● I've Been Everywhere, Man	Data Structure
● Matrix Keypad	Pattern Matching
● Popular Vote	Implementation (Be careful with edge cases!)
● Rubik's Revenge	Search (optional bit manipulation)
● Scaling Recipes	Implementation (Floating Point Issues)
● Space Junk	Algebra / Geometry (Floating Point issues)
● The Magical 3	Number Theory, (Didn't appear in the mock).

Announcements - This Week’s Mock

- Sunday October 1st, at 2:00 PM in this room (seminar room)
- Will focus on DP, Union Find, and Prefix Trees
- More guided format

Other ACM Announcements

- Tea Time at Momma Goldberg's (on Thach) tomorrow at 6:00!
- For any questions / comments / concerns / or just to get in contact with other ACM members, feel free to join us on slack!
  - AuburnACM.slack.com
- Workiva will be giving a Tech Talk on October 24th on Cloud Computing
- If you have any ideas for / would like to give a tech talk, let us know!

What is Dynamic Programming?

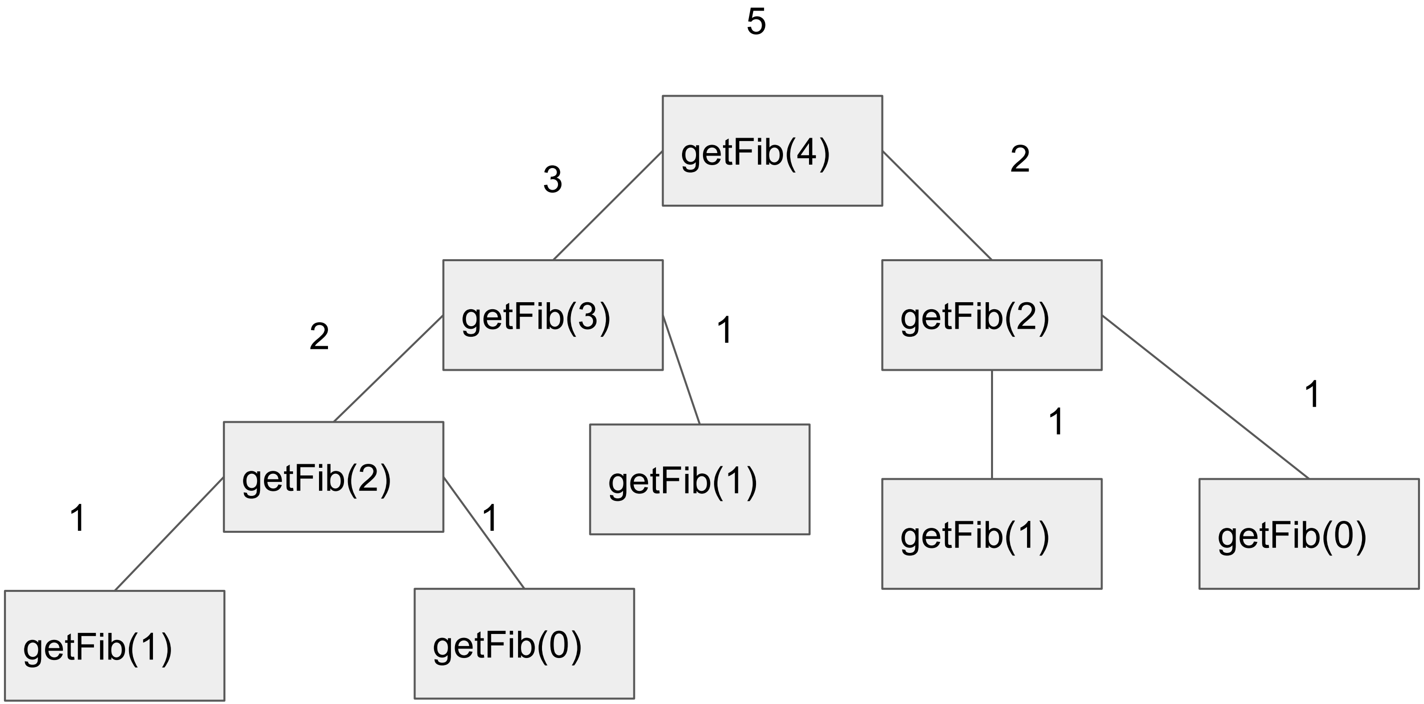
- Ignore the name (it is intentionally scary)
- Basically, it just means you remember the result of a previous computation, rather than having to re-calculate it again!
- Example: write a function that calculates the Nth fibonacci number
- 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, ….
- Fib (0) = Fib(1) = 1, otherwise Fib(N) = Fib(N-1) + Fib(N-2)

```
static int getFib(int N)
```

One Possible Solution

```
static int getFib(int N) {
    if (N == 0 || N == 1) {
        return 1;
    }
    return getFib(N-1) + getFib(N-2);
}
```

Call Structure

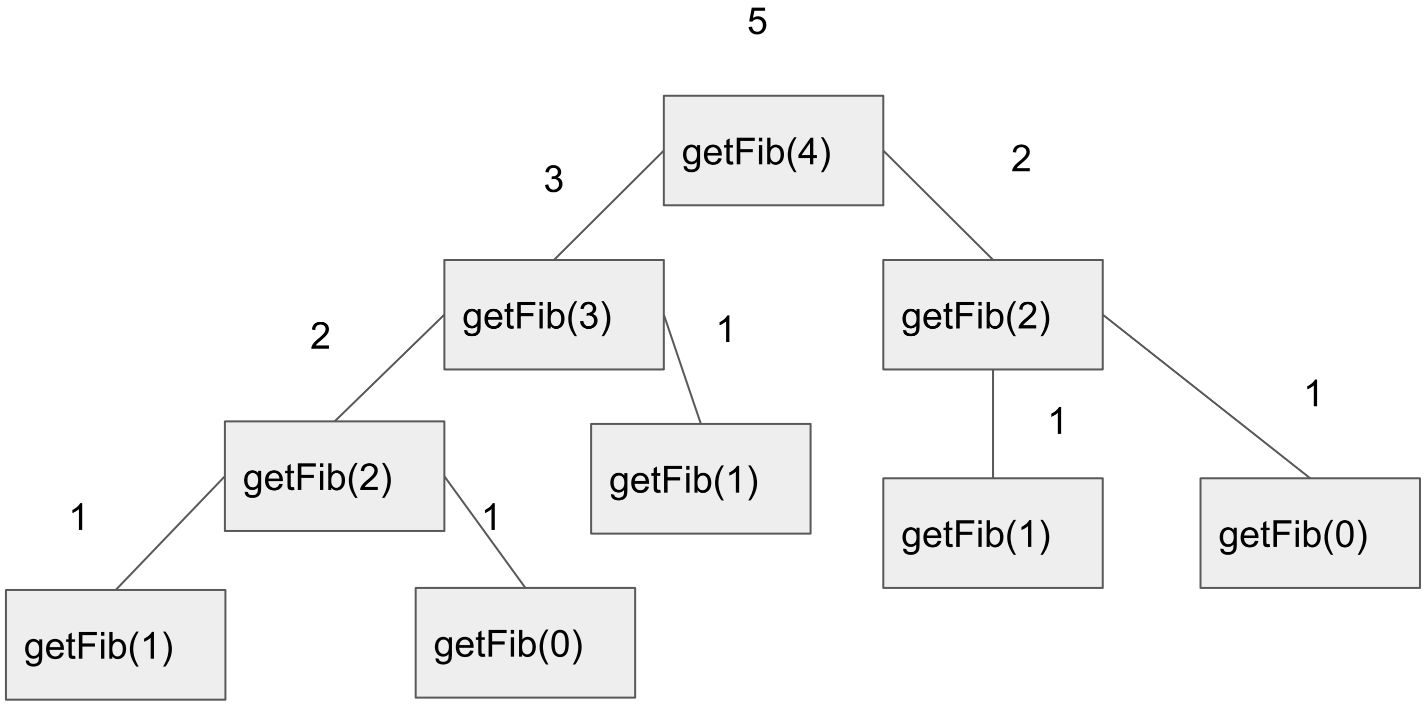


An Improvement

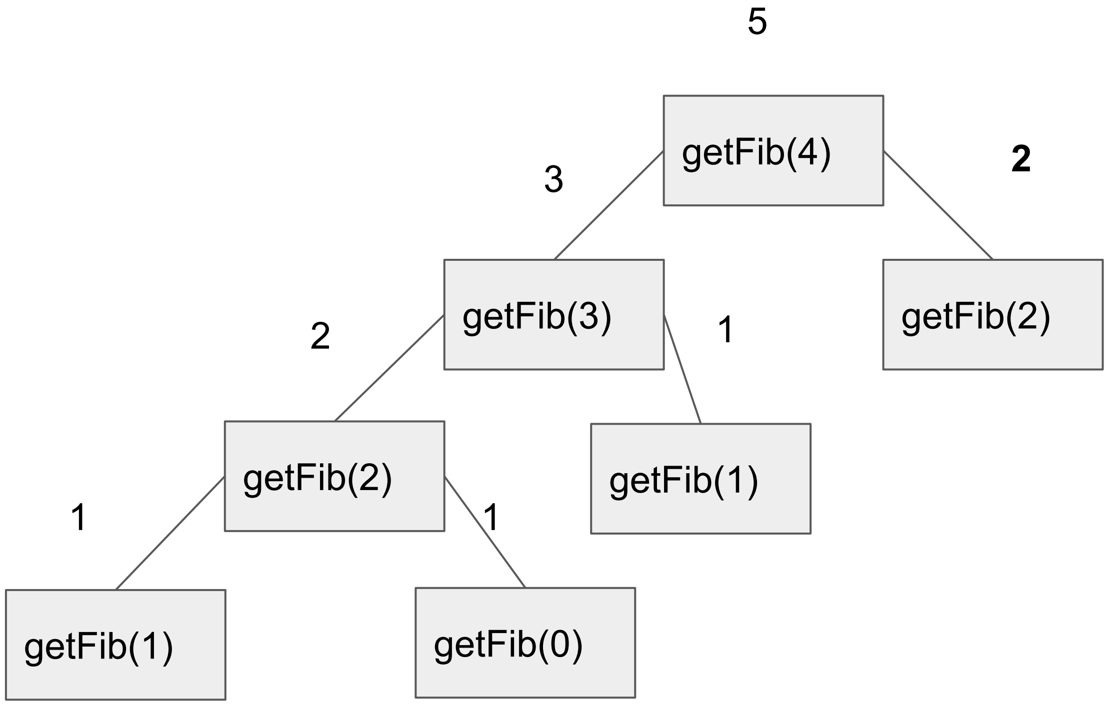
```
int[] solutions = new int[100];

static int getFib(int N) {
    if (N == 0 || N == 1) {
        return 1;
    }
    if (solutions[N] == 0) {
        solutions[N] = getFib(N-1) + getFib(N-2);
    }
    return solutions[N]
}
```

Call Structure - Before



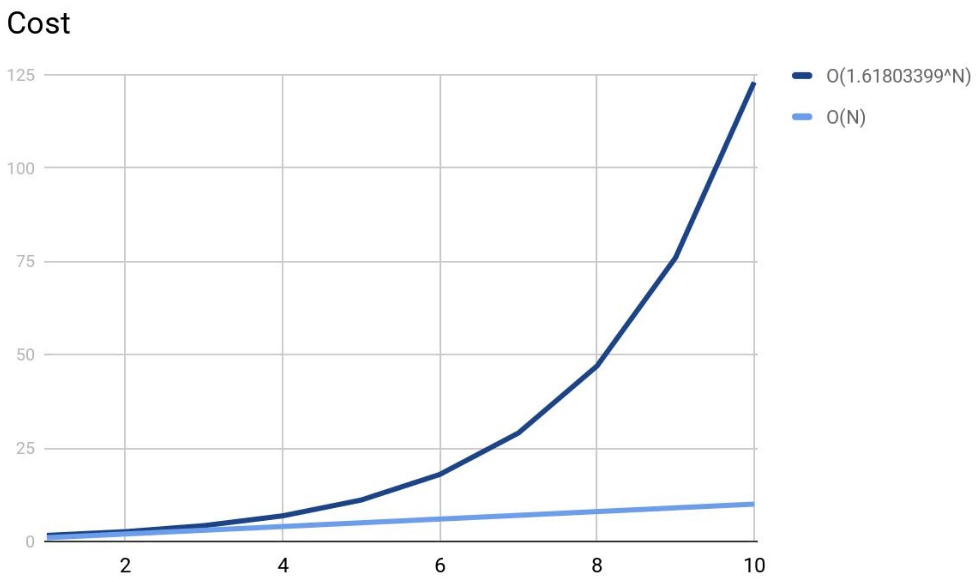
Call Structure - After



Improvement

Before - O (1.61803399^N)

After - O(N)



Further Changes

Recursive calls have an overhead, so we can replace it with:

```
static int getFib(int N) {
    if (N < 2) return 1;
    int[] fibs = new int[N + 1];
    fibs[0] = fibs[1] = 1;
    for (int i = 2; i <= N; i++) {
        fibs[i] = fibs[i-1] + fibs[i-2];
    }
    return fibs[N];
}
```

What is Dynamic Programming?

More formally, a dynamic programming problem aims to solve a problem with 2 properties:

Optimal Substructure (In other words, we can get the answer to a problem using the answer to other subproblems).

Overlapping Subproblems (These subproblems actually have something in common).

Maximum Subarray

We are given an array of numbers and we need to find a subarray (a range of consecutive elements in the array) that sums to the largest value:

−2, 1, −3, **4, −1, 2, 1**, −5, 4

The maximum subarray is 4, -1, 2, 1, which sums to 6.

Maximum Subarray

Optimal Substructure:

maxSum(A) = max(0, maxSumEndingAt(A, 0) … maxSumEndingAt(A, N-1))

maxSumEndingAt(A, i) = max(maxSumEndingAt(A, i-1) + A[i], A[i])

Overlapping Subproblems:

Rather than recalculate maxSumEndingAt, we can simply store it in a variable as we iterate over the array.

Maximum Subarray

```
int maxSum(int[] A) {
    int runningTotal = 0, max = 0;
    for (int i = 0; i < A.length; i++) {
        runningTotal = Math.max(A[i], runningTotal + A[i]);
        max = Math.max(max, runningTotal);
    }
    return max;
}
```

Longest Increasing Subsequence

a i e m **c** k **g** o b **j** f n d **l** h **p**

Given a list of elements, find the length of the longest subsequence in those elements that is monotonically increasing.

Longest Increasing Subsequence

Optimal substructure:

Let smallestNthElement(A, n, i) be the smallest element of A[:i] that terminates an increasing subsequence of length n in A[:i]. Then:

```
lengthLongestIncreasing(A, i) =
    lengthLongestIncreasing(A, i-1) + 1 if A[i] >
        smallestNthElement(A, lengthLongestIncreasing(A, i-1), i-1)
    lengthLongestIncreasing(A, i-1) otherwise.
```

Longest Increasing Subsequence

Overlapping subproblems:

We can keep a mapping of N to smallestNthElement, updating it as we iterate over i.

For each iteration, we can find the first element in the mapping >= A[i] and replace that element with A[i].

Longest Increasing Subsequence

```
static int longestIncreasingSubsequence(int[] A) {
      int[] smallest = new int[A.length];
      Arrays.fill(smallest, Integer.MAX_VALUE);
      int length = 0;
      for (int i = 0; i < A.length; i++) {
            int insertAt = 0;
            for (int j = 0; j < A.length; j++) {
                  if (smallest[j] >= A[i]) {
                        insertAt = j;
                        break;
                  }
            }
            smallest[insertAt] = A[i];
            length = Math.max(length, insertAt + 1);
      }
      return length;
}
```

Longest Increasing Subsequence

But wait, there's more!

Because smallest is monotonically increasing, we can use a binary search or another log(N) find operation to improve our performance even further!

Knapsack

Imagine this:

You break into a house, carrying a knapsack to hold your loot, which can hold up to W pounds.

Knowing the weight / value of every item in the house, maximize the value of items your knapsack can hold without breaking. We will assume integer weights / values.

- 0/1
- Bounded
- Unbounded (What we will talk about)

Unbounded Knapsack

Optimal Substructure:

Let's say I can hold up to W pounds. There are N items numbered 0...N-1, where weight(i) is the weight of the ith item and value(i) is the value of the ith item.

```
MaxLoot(W) =
    max for i in 0...N-1
        value(i) + MaxLoot(W-weight(i)),
    MaxLoot(W-1)
```

Overlapping Subproblems:

Since MaxLoot(W) only depends on previous values of MaxLoot, we can store these answers rather than recomputing them.

Unbounded Knapsack

```
static int maxValue(Item[] items, int maxWeight) {
    int[] best = new int[maxWeight+1];
    for (int w = 1; w <= maxWeight; w++) {
        best[w] = best[w-1];
        for (Item item : items) {
            if (w - item.weight >= 0) {
                best[w] = Math.max(best[w-item.weight] + item.value, best[w]);
            }
        }
    }
    return best[maxWeight];
}
```

Egg Drop Problem

Let's say you want to find the highest floor that you can drop an egg from without it cracking. (With floors numbered 1 to N)

If you only have 1 egg, you have to test floor 1, floor 2, floor 3…. Etc.

If you have N or more eggs, we can binary search… so we only need the ceiling of $\log_2 N$ drops or less

What if we have K eggs, where $1 < K < N$? Can we determine the maximum # of drops needed if we use an optimal strategy?

Egg Drop Problem

Optimal substructure:

Let FirstDrop(N, K, F) mean the worst-case # of drops if I am using an optimal strategy for N floors and K eggs after an initial first drop from floor F.

Now FirstDrop(N, K, F) = 1 + max(EggDrop(F-1, K-1), EggDrop(N-F, K))

We know EggDrop(N, 1) = N, EggDrop(0, K) = 0, and EggDrop(1, K) = 1.

Finally, EggDrop(N, K) = min(FirstDrop(N, K, F) for F in 0...K-1)

Egg Drop Problem

```
static int eggDrop(int N, int K) {
      int[][] memo = new int[N+1][K+1];
      for (int i = 1; i <= K; i++) {
            memo[1][i] = 1;
            memo[0][i] = 0;
      }
      for (int j = 1; j <=k; j++) memo[j][1] = j;
      for (int eggs = 2; eggs <= K; eggs++) {
            for (int floor = 2; floor <= N; floor++) {
                  memo[floor][eggs] = Integer.MAX_VALUE;
                  for (int firstDrop = 1; firstDrop <= floor; firstDrop++) {
                        memo[floor][eggs] = Math.min(memo[floor][eggs], 1 +
                              Math.max(memo[firstDrop-1][eggs-1], memo[floor-firstDrop][eggs]));
                  }
            }
      }
      return memo[N][K]
}
```

Your Turn!

- Maximum Subarray - Profits
  - auacm.com/problem/profits
- Unbounded Knapsack - Candy Store
  - auacm.com/problem/candystore
- Egg Drop (with a twist) - Mailbox Manufacturer’s Problem
  - open.kattis.com/problems/mailbox
- Roll Your Own - Digit Sum
  - open.kattis.com/problems/digitsum