CSC 427: Data Structures and Algorithm Analysis

Fall 2006

Dynamic programming

- top-down vs. bottom up
- caching
- dynamic programming vs. divide & conquer
- examples: making change, Fibonacci sequence, binomial coefficient
- problem-solving approaches summary

1

ACM programming contest problem

November 13, 2004 *ACM North Central North America Regional Programming Contest* *Problem 1*

Problem 1: Breaking a Dollar

Using only the U. S. coins worth 1, 5, 10, 25, 50, and 100 cents, there are exactly 293 ways in which one U. S. dollar can be represented. Canada has no coin with a value of 50 cents, so there are only 243 ways in which one Canadian dollar can be represented. Suppose you are given a new set of denominations for the coins (each of which we will assume represents some integral number of cents less than or equal to 100, but greater than 0). In how many ways could 100 cents be represented?

Input

The input will contain multiple cases. The input for each case will begin with an integer *N* (at least 1, but no more than 10) that indicates the number of unique coin denominations. By *unique* it is meant that there will not be two (or more) different coins with the same value. The value of *N* will be followed by *N* integers giving the denominations of the coins.

Input for the last case will be followed by a single integer -1.

Output

For each case, display the case number (they start with 1 and increase sequentially) and the number of different combinations of those coins that total 100 cents. Separate the output for consecutive cases with a blank line.

Sample Input	Output for the Sample Input
6 1 5 10 25 50 100 5 1 5 10 25 100 -1	Case 1: 293 combinations of coins Case 2: 243 combinations of coins

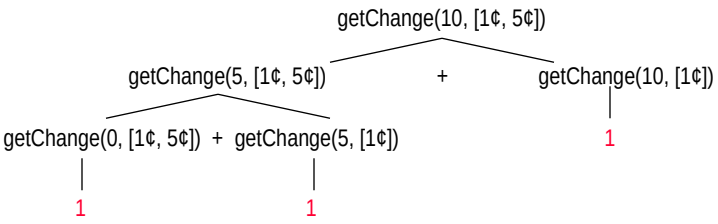
2

Divide & conquer solution

let `getChange(amount, coinList)` represent the number of ways to get an amount using the specified list of coins

```
getChange(amount, coinList) =
   getChange(amount-biggestCoinValue, coinList) // # of ways that use at least
   +                                            //   one of the biggest coin
   getChange(amount, coinList-biggestCoin)      // # of ways that don't
                                                //   involve the biggest coin
```

e.g., suppose want to get 10¢ using only pennies and nickels



3

Divide & conquer code

could implement as a `ChangeMaker` class

- when constructing, specify a sorted list of available coins (why sorted?)
- recursive helper method works with a possibly restricted coin list

```
public class ChangeMaker {
    private List<Integer> coins;

    public ChangeMaker(List<Integer> coins) {
        this.coins = coins;
    }

    public int getChange(int amount) {
        return this.getChange(amount, coins.size()-1);
    }

    private int getChange(int amount, int maxCoinIndex) {
        if (amount < 0 || maxCoinIndex < 0) {
            return 0;
        }
        else if (amount == 0) {
            return 1;
        }
        else {
            return this.getChange(amount-this.coins.get(maxCoinIndex), maxCoinIndex) +
                   this.getChange(amount, maxCoinIndex-1);
        }
    }
}
```

base case: if amount or max coin index becomes negative, then can't be done

base case: if amount is zero, then have made exact change

recursive case: count how many ways using a largest coin + how many ways not using a largest coin

4

Divide & conquer code

processing the input data sets and producing the proper output can be done by a main method

- could be placed in a separate driver class, or in `ChangeMaker` itself

```
public static void main(String[] args) {
     Scanner input = new Scanner(System.in);

     int caseNum = 0;
     int numCoins = input.nextInt();

     while (numCoins != -1) {
         caseNum++;
         ArrayList<Integer> coins = new ArrayList<Integer>();
         for (int i = 0; i < numCoins; i++) {
             coins.add(input.nextInt());
         }
         Collections.sort(coins);

         ChangeMaker cm = new ChangeMaker(coins);

         if (caseNum > 1) {
             System.out.println();
         }
         System.out.println("Case " + caseNum + ": " + cm.getChange(100) +
                            " combinations of coins");

         numCoins = input.nextInt();
     }
 }
```

5

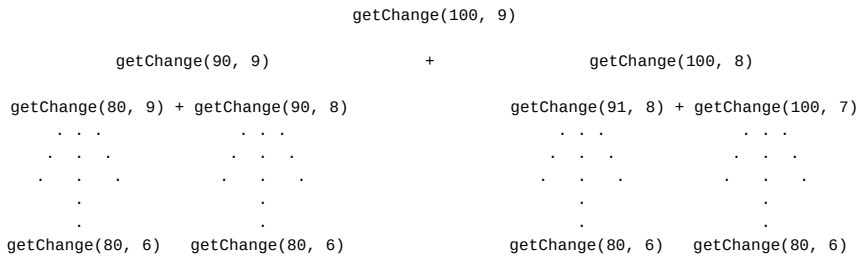
Will this solution work?

certainly, it will produce the correct answer -- but, how quickly?

- at most 10 coins
- worst case: 1 2 3 4 5 6 7 8 9 10

- TAKES A LOOOOOOOOOOOOONG TIME!

the problem is duplication of effort



6

Caching

a simple solution is to augment the program to *cache* results

- create a table in which to store results as they are computed

```
int[][] remember = new int[MAX_AMOUNT+1][MAX_COINS];

for (int a = 0; a < MAX_AMOUNT+1; a++) {
    for (int c = 0; c < MAX_COINS; c++) {
        remember[a][c] = -1;
    }
}
```

- when `getChange` is called, first check to see if answer has already been cached
  if so (`entry != -1`), then simply report the answer without recomputing
  if not (`entry == -1`), compute the answer and store in the table before returning

Caching version

```
public class ChangeMaker {
    private List<Integer> coins;

    private static final int MAX_AMOUNT = 100;
    private static final int MAX_COINS = 10;
    private int[][] remember;

    public ChangeMaker(List<Integer> coins) {
        this.coins = coins;

        remember = new int[ChangeMaker.MAX_AMOUNT+1][ChangeMaker.MAX_COINS];
        for (int r = 0; r < ChangeMaker.MAX_AMOUNT+1; r++) {
            for (int c = 0; c < ChangeMaker.MAX_COINS; c++) {
                remember[r][c] = -1;
            }
        }
    }

    public int getChange(int amount) {
        return this.getChange(amount, coins.size()-1);
    }

    private int getChange(int amount, int maxCoinIndex) {
        if (maxCoinIndex < 0 || amount < 0) {
            return 0;
        }

        if (amount == 0) {
            remember[amount][maxCoinIndex] =  1;
        }
        else {
            remember[amount][maxCoinIndex] =
                this.getChange(amount-this.coins.get(maxCoinIndex), maxCoinIndex) +
                this.getChange(amount, maxCoinIndex-1);
        }
        return remember[amount][maxCoinIndex];
    }
}
```

with caching, even worst case answer is fast:

6,292,069 combinations

Dynamic programming

caching solutions to smaller problems, building up to the goal is known as *dynamic programming*

- applicable to same types of problems as divide & conquer
- bottom-up, as opposed to divide & conquer which is top-down
- usually more effective than top-down if the parts are not completely independent (thus leading to redundancy)

silly Fibonacci example: top-down divide & conquer

```
public static int fib(int n) {
    if (n <= 1) {
        return 1;
    }
    else {
        return fib(n-1) + fib(n-2);
    }
}
```

smarter Fibonacci : bottom-up dynamic programming

```
public static int fib(int n) {
    int prev = 1, current = 1;
    for (int i = 1; i < n; i++) {
        int next = prev + current;
        prev = current;
        current = next;
    }
    return current;
}
```

9

Example: binary coefficient

binomial coefficient C(n, k) is relevant to a large number of problems

- the number of ways can you select k lottery balls out of n
- the number of ways to get k heads when a coin is tossed n times,
- the number of birth orders possible in a family of n children where k are sons
- the number of acyclic paths connecting 2 corners of an k×(n-k) grid

- the coefficient of the $x^k y^{n-k}$ term in the polynomial expansion of $(x + y)^n$
- the entry at the nth row and kth column of Pascal's triangle

$$C(n,k) \equiv \binom{n}{k} \equiv \frac{n!}{k!(n-k)!}$$

10

Example: binary coefficient

while easy to define, a binomial coefficient is difficult to compute

e.g, 6 number lottery with 49 balls → 49!/6!43!

49! = 608,281,864,034,267,560,872,252,163,321,295,376,887,552,831,379,210,240,000,000,000

could try to get fancy by canceling terms from numerator & denominator

- can still can end up with individual terms that exceed integer limits

a computationally easier approach makes use of the following recursive relationship

$$\binom{n}{k} \equiv \binom{n-1}{k-1} + \binom{n-1}{k}$$

e.g., to select 6 lottery balls out of 49, partition into:

selections that include 1
(must select 5 out of remaining 48)
+
selections that don't include 1
(must select 6 out of remaining 48)

11

Example: binomial coefficient

could use straight divide&conquer to compute based on this relation

```
/**
 * Calculates n choose k (using divide-and-conquer)
 *   @param n the total number to choose from (n > 0)
 *   @param k the number to choose (0 <= k <= n)
 *   @return n choose k (the binomial coefficient)
 */
public static int binomial(int n, int k) {
    if (k == 0 || n == k) {
        return 1;
    }
    else {
        return binomial(n-1, k-1) + binomial(n-1, k);
    }
}
```

however, this will take a long time or exceed memory due to redundant work

$$\binom{49}{6}$$

$$\binom{48}{5} \quad + \quad \binom{48}{6}$$

$$\binom{47}{4} + \binom{47}{5} \qquad \binom{47}{5} + \binom{47}{6}$$

12

Dynamic programming solution

could use caching to store answers, or build a solution entirely from the bottom-up (starting at base cases)

	0	1	2	3	…	k
0	1					
1	1	1				
2	1		1			
3	1			1		
…	…				…	
n	1					1

```
/**
 * Calculates n choose k (using dynamic programming)
 *   @param n the total number to choose from (n > 0)
 *   @param k the number to choose (0 <= k <= n)
 *   @return n choose k (the binomial coefficient)
 */
public static int binomial(int n, int k) {
    if (n < 2) {
        return 1;
    }
    else {
        int bin[][] = new int[n+1][n+1];        // CONSTRUCT A TABLE TO STORE

        for (int r = 0; r <= n; r++) {          // COMPUTED VALUES
            for (int c = 0; c <= r && c <= k; c++) {
                if (c == 0 || c == r) {
                    bin[r][c] = 1;              // ENTER 1 IF BASE CASE
                }
                else {                          // OTHERWISE, USE FORMULA
                    bin[r][c] = bin[r-1][c-1] + bin[r-1][c];
                }
            }
        }
        return bin[n][k];                       // ANSWER IS AT bin[n][k]
    }
}
```


World series puzzle

Consider the following puzzle:

At the start of the world series (best-of-7), you must pick the team you want to win and then bet on games so that

- if your team wins the series, you win exactly $1,000
- if your team loses the series, you lose exactly $1,000

You may bet different amounts on different games, and can even bet $0 if you wish.

QUESTION: how much should you bet on the first game?

Algorithmic approaches summary

divide & conquer: tackles a complex problem by breaking it into smaller pieces, solving each piece, and combining them into an overall solution

- often involves recursion, if the pieces are similar in form to the original problem
- applicable for any application that can be divided into independent parts

dynamic: bottom-up implementation of divide & conquer – start with the base cases and build up to the desired solution, caching results to avoid redundant computation

- usually more effective than top-down recursion if the parts are not completely independent (thus leading to redundancy)
- note: can approximate using top-down recursion with caching

greedy: involves making a sequence of choices/actions, each of which simply looks best at the moment

- applicable when a solution is a sequence of moves & perfect knowledge is available

backtracking: involves making a sequence of choices/actions (similar to greedy), but stores alternatives so that they can be attempted if the current choices lead to failure

- more costly in terms of time and memory than greedy, but general-purpose
- worst case: will have to try every alternative and exhaust the search space

15