CSC 421: Algorithm Design & Analysis

Spring 2014

Dynamic programming

- top-down vs. bottom-up

- divide & conquer vs. dynamic programming

- examples: Fibonacci sequence, binomial coefficient

- examples: World Series puzzle, Floyd's algorithm

- top-down with caching

- example: making change

- problem-solving approaches summary

1

Divide and conquer

divide/decrease & conquer are *top-down* approaches to problem solving
- start with the problem to be solved (i.e., the top)
- break that problem down into smaller piece(s) and solve
- continue breaking down until reach base/trivial case (i.e., the bottom)

they work well when the pieces can be solved independently
- e.g., merge sort – sorting each half can be done independently, no overlap

what about Fibonacci numbers? 1, 1, 2, 3, 5, 8, 13, 21, …

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

2

1

Top-down vs. bottom-up

divide and conquer is a horrible way of finding Fibonacci numbers
- the recursive calls are NOT independent; redundencies build up

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

```
                    fib(5)

        fib(4)        +        fib(3)

fib(3) + fib(2)          fib(2) + fib(1)
   .        .              .            .
   .        .              .            .
   .        .              .            .
```

in this case, a bottom-up solution makes more sense
- start at the base cases (the bottom) and work up to the desired number
- requires remembering the previous two numbers in the sequence

```
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;
}
```

3

Dynamic programming

dynamic programming is a bottom-up approach to solving problems
- start with smaller problems and build up to the goal, storing intermediate solutions as needed

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

example: binomial coefficient C(n, k) is relevant to many problems
- the number of ways can you select k lottery balls out of n
- 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 $k \times (n-k)$ grid
- the coefficient of the $x^k y^{n-k}$ term in the polynomial expansion of $(x + y)^n$

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

4

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)

5

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 <u>long</u> time or exceed memory due to redundant work

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

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

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

6

Dynamic programming solution

could instead work bottom-up, filling a
table starting with the base cases
(when k = 0 and n = k)

	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]
    }
}
```

7

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?

DIVIDE & CONQUER SOLUTION? DYNAMIC PROGRAMMING?

8

World Series puzzle code

```
private static double winningsTopDown(int numWins, int numLosses) {
    if (numWins == GAMES_TO_WIN) {
        return FINAL_WORTH;
    }
    else if (numLosses == GAMES_TO_WIN) {
        return -FINAL_WORTH;
    }
    else {
        return (winningsTopDown(numWins+1, numLosses) +
                winningsTopDown(numWins, numLosses+1))/2;
    }
}
```

divide & conquer solution

```
private static double winningsBottomUp(int numWins, int numLosses) {
    double winGrid[][] = new double[GAMES_TO_WIN+1][GAMES_TO_WIN+1];
    for (int w = GAMES_TO_WIN; w >= numWins; w--) {
        for (int l = GAMES_TO_WIN; l >= numLosses; l--) {
            if (w == GAMES_TO_WIN) {
                winGrid[w][l] = FINAL_WORTH;
            }
            else if (l == GAMES_TO_WIN) {
                winGrid[w][l] = -FINAL_WORTH;
            }
            else {
                winGrid[w][l] = (winGrid[w+1][l]+winGrid[w][l+1])/2;
            }
        }
    }
    return winGrid[numWins][numLosses];
}
```

dynamic programming solution

9

All-pairs shortest path

recall Dijkstra's algorithm for finding the shortest path between two vertices in a graph
- simple implementation is $O(|V|^2)$
- trickier implementaion is $O((|E| + |V|) \log |V|)$

suppose we wanted to find the shortest paths between all pairs of vertices
- e.g., oracleofbacon.org, baseball-reference.com/oracle/

brute force solution: run Dijkstra's algorithm for every pair
- $O(|V|^2)$ pairs x $O(|V|^2)$ → $O(|V|^4)$
- $O(|V|^2)$ pairs x $O((|E| + |V|) \log |V|)$ → $O((|E||V|^2 + |V|^3) \log |V|)$

we can do much better

10

Floyd's algorithm

Floyd's algorithm uses a dynamic programming approach to find all shortest path is $O(|V|^3)$

- computes a series of distance matrices $D^{(0)}, D^{(1)}, \ldots, D^{(|V|)}$

- $D^{(0)}$ is is the weight matrix for the graph ($D^{(0)}[i][j]$ is weight of edge from v_i to v_j) represents the minimum distances if no intermediate vertices are allowed
- $D^{(1)}$ allows paths with v_1 as intermediate vertex
- $D^{(2)}$ allows paths with v_1, v_2 as intermediate vertices
- …
- $D^{(|V|)}$ allows paths with $v_1, v_2, \ldots, v_{|V|}$ as intermediate vertices (i.e., all paths)

when calculating the entries in $D^{(k)}$

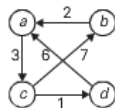- shortest path connecting v_i and v_j either uses v_k or it doesn't

$$D^{(k)}[i][j] = \min(D^{(k-1)}[i][j], D^{(k-1)}[i][k] + D^{(k-1)}[k][j])$$

11

Floyd's example

$D^{(0)}[i][j] = w_{ij}$
$D^{(k)}[i][j] = \min(D^{(k-1)}[i][j], D^{(k-1)}[i][k] + D^{(k-1)}[k][j])$

$$D^{(0)} = \begin{array}{c|cccc} & a & b & c & d \\ \hline a & 0 & \infty & 3 & \infty \\ b & 2 & 0 & \infty & \infty \\ c & \infty & 7 & 0 & 1 \\ d & 6 & \infty & \infty & 0 \end{array}$$

Lengths of the shortest paths with no intermediate vertices ($D^{(0)}$ is simply the weight matrix).

$$D^{(1)} = \begin{array}{c|cccc} & a & b & c & d \\ \hline a & 0 & \infty & 3 & \infty \\ b & 2 & 0 & 5 & \infty \\ c & \infty & 7 & 0 & 1 \\ d & 6 & \infty & 9 & 0 \end{array}$$

Lengths of the shortest paths with intermediate vertices numbered not higher than 1, i.e., just a (note two new shortest paths from b to c and from d to c).

$$D^{(2)} = \begin{array}{c|cccc} & a & b & c & d \\ \hline a & 0 & \infty & 3 & \infty \\ b & 2 & 0 & 5 & \infty \\ c & 9 & 7 & 0 & 1 \\ d & 6 & \infty & 9 & 0 \end{array}$$

Lengths of the shortest paths with intermediate vertices numbered not higher than 2, i.e., a and b (note a new shortest path from c to a).

$$D^{(3)} = \begin{array}{c|cccc} & a & b & c & d \\ \hline a & 0 & 10 & 3 & 4 \\ b & 2 & 0 & 5 & 6 \\ c & 9 & 7 & 0 & 1 \\ d & 6 & 16 & 9 & 0 \end{array}$$

Lengths of the shortest paths with intermediate vertices numbered not higher than 3, i.e., a, b, and c (note four new shortest paths from a to b, from a to d, from b to d, and from d to b).

$$D^{(4)} = \begin{array}{c|cccc} & a & b & c & d \\ \hline a & 0 & 10 & 3 & 4 \\ b & 2 & 0 & 5 & 6 \\ c & 7 & 7 & 0 & 1 \\ d & 6 & 16 & 9 & 0 \end{array}$$

Lengths of the shortest paths with intermediate vertices numbered not higher than 4, i.e., a, b, c, and d (note a new shortest path from c to a).

12

Floyd's algorithm

bottom-up algorithm statement:

```
D = weight matrix
for k from 1 to |V|:
    for i from 1 to |V|:
        for j from 1 to |V|:
            D[i][j] = min( D[i][j], D[i][k] + D[k][j] );
```

clearly, this is $O(|V|^3)$

could be formulated top-down (requires 3 dimensions)

solve(0, i, j) ← weight[i][j]

solve(k, i, j) ← min(solve(k-1, i, j), solve(k-1, i, k) + solve(k-1, k, j))

Dynamic programming & caching

when calculating C(n,k), the entire table must be filled in (up to the nth row and kth column)
- → working bottom-up from the base cases does not waste any work

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

for many problems, this is not the case
- solving a problem may require only a subset of smaller problems to be solved
- constructing a table and exhaustively working up from the base cases could do lots of wasted work

- can dynamic programming still be applied?

Example: programming contest problem

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.

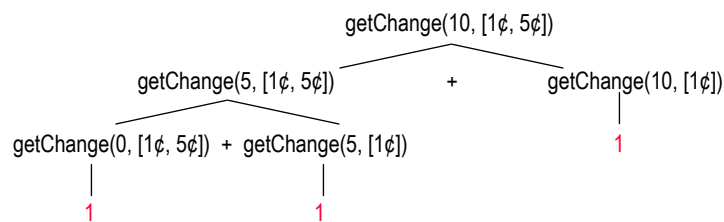| Sample Input	Output for the Sample Input
6 1 5 10 25 50 100	Case 1: 293 combinations of coins
5 1 5 10 25 100	
-1	Case 2: 243 combinations of coins

15

Divide & conquer approach

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

getChange(10, [1¢, 5¢])

getChange(5, [1¢, 5¢])   +   getChange(10, [1¢])

getChange(0, [1¢, 5¢]) + getChange(5, [1¢])   1

1   1

16

8

Divide & conquer solution

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, this.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
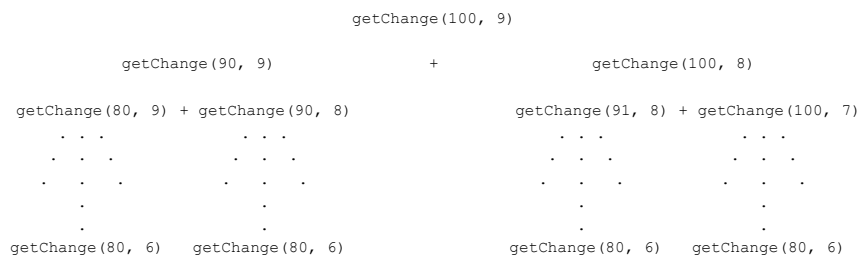
17

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

- 6,292,069 combinations → depending on your CPU, this can take a while
- # of combinations will explode if more than 10 coins allowed

the problem is duplication of effort

```
                          getChange(100, 9)

      getChange(90, 9)              +              getChange(100, 8)

 getChange(80, 9) + getChange(90, 8)        getChange(91, 8) + getChange(100, 7)
    . . .            . . .                     . . .            . . .
   .   .  .         .   .  .                   .   .  .         .   .  .
  .   .   .        .   .   .                   .   .   .        .   .   .
     .                .                            .                .
     .                .                            .                .
 getChange(80, 6)   getChange(80, 6)           getChange(80, 6)   getChange(80, 6)
```

18

9

Caching

we could use dynamic programming and solve the problem bottom up
- however, consider getChange(100, [1¢, 5¢, 10¢, 25¢])
- would we ever need to know getChange(99, [1¢, 5¢, 10¢, 25¢]) ?
  getChange(98, [1¢, 5¢, 10¢, 25¢]) ?
  getChange(73, [1¢, 5¢]) ?

when exhaustive bottom-up would yield too many wasted cases, dynamic programming can instead utilize top-down with *caching*
- create a table (as in the exhaustive bottom-up approach)
- however, fill the table in using a top-down approach
   that is, execute a top-down decrease and conquer solution, but store the solutions to subproblems in the table as they are computed
- before recursively solving a new subproblem, first check to see if its solution has already been cached

- avoids the duplication of pure top-down
- avoids the waste of exhaustive bottom-up (only solves relevant subproblems)

19

ChangeMaker with caching

as each subproblem is solved, its solution is stored in a table

each call to getChange checks the table first before recursing

```
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;

        this.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++) {
                this.remember[r][c] = -1;
            }
        }
    }

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

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

with caching, even the worst case is fast:

6,292,069 combinations

20

10

Algorithmic approaches summary

brute force: sometimes the straightforward approach suffices

transform & conquer: sometimes the solution to a simpler variant suffices

divide/decrease & conquer: tackles a complex problem by breaking it into smaller piece(s), solving each piece (often w/ recursion), and combining into an overall solution
- applicable for any application that can be divided into independent parts

dynamic: bottom-up implementation of divide/decrease & conquer – start with the base cases and build up to the desired solution, storing results to avoid redundancy
- usually more effective than top-down recursion if the parts are not completely independent
- can implement by adding caching to top-down recursion

greedy: makes a sequence of choices/actions, choose whichever looks best at the moment
- applicable when a solution is a sequence of moves & perfect knowledge is available

backtracking: makes 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
- branch & bound variant cuts off search at some level and backtracks

21