# CSC 427: Data Structures and Algorithm Analysis

## Fall 2007

Dynamic programming

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



# Divide and conquer

divide and conquer is a *top-down* approach to problem solving

- start with the problem to be solved (i.e., the top)
- break that problem down into smaller pieces and solve
- continue breaking down until reach base/trivial case (i.e., the bottom)

divide and conquer works 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, …

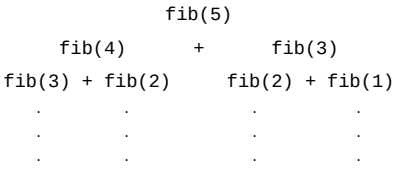
```
public static int fib(int n) {
    if (n <= 1) {
        return 1;
    }
    else {
        return fib(n-1) + fib(n-2);
    }
}
```

## 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);
    }
}
```




### 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 & 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×(n-k) grid
- the coefficient of the $x^k y^{n-k}$ term in the polynomial expansion of $(x + y)^n$

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

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 long time or exceed memory due to redundant work

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

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

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

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

## 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 rquire 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?

8

## Example: 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<br>5 1 5 10 25 100<br>-1 | Case 1: 293 combinations of coins<br><br>Case 2: 243 combinations of coins |

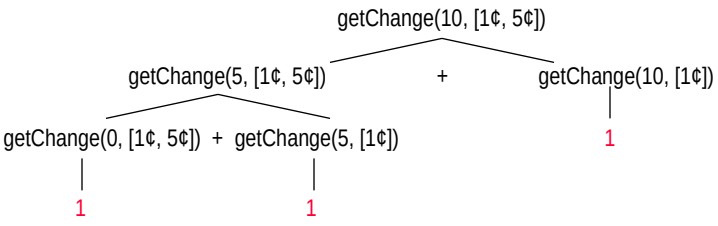
9

## 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




10

## 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

11

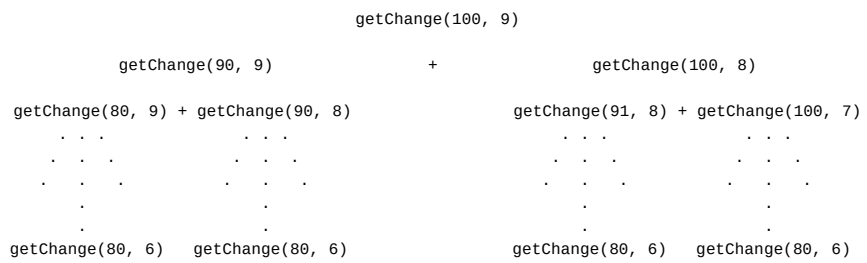
## Will this solution work?

### 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




12

## 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 divide 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)

13

## 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

with caching, even the worst case is fast:

6,292,069 combinations

```
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];
    }
}
```

14

## HW 6: Minimal Edit Distance

suppose you are building an autocorrect feature into a word processor

- given a mistyped word, want to find the "nearest match" in a dictionary

helbo could be "hello" with 'b' substituted for 'l'
could be "elbow" with an extraneous 'h' and a missing 'w'

we can associate a cost with each editing operation

sCost = cost associated with subsituting one letter for another
iCost = cost associated with inserting a letter at any point
dCost = cost associated with deleting any letter

helbo → hello substitute 'l' for 'b', so cost is (1×sCost)

helbo → elbow delete 'h', insert 'w', so cost is (1×dCost + 1×iCost)

hello → flow ???

there can be multiple editing sequences that transform words – we want the minimal cost of a transformation (i.e., minimal edit distance)

15

## HW 6: Divide & conquer pseudocode

for HW6, implement a solution based on the following pseudocode

- note: you will need to add caching in order to handle longer words

```
public int distance(String word1, String word2) {
    if (word1.length() == 0) {
        return COST OF ADDING LETTERS TO word2;
    }
    else if (word2.length() == 0) {
        return COST OF REMOVING LETTERS FROM word1;
    }
    else {
        int distIfSubstFirst = COST IF TRANSFORM FIRST LETTER OF word1 VIA SUBSTITUTION;
        int distIfInsertFirst = COST IF TRANSFORM FIRST LETTER OF word1 VIA INSERTION;
        int distIfDeleteFirst = COST IF TRANSFORM FIRST LETTER OF word1 VIA DELETION;

        return Math.min(distIfSubstFirst, Math.min(distIfInsertFirst, distIfDeleteFirst));
    }
}
```

e.g., helpo → flow

- if substitution first: cost of substituting 'f' for 'h' + distance between "elpo" and "low" (note: if first letters match, then there is no substitution required)
- if insertion first: cost of inserting 'f' + distance between "helpo" and "low"
- if deletion first: cost of deleting 'h' + distance between "elpo" and "flow"

16

## 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, storing 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 implement by adding caching to top-down recursion

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

17