Dynamic Programming

adapted from David Luebke

Algorithm types

- Algorithm types we will consider include:
  - Simple recursive algorithms
  - Backtracking algorithms
  - Divide and conquer algorithms
  - Dynamic programming algorithms
    - difference from divide and conquer algorithms
  - Greedy algorithms
  - Branch and bound algorithms
  - Brute force algorithms
  - Randomized algorithms

Counting coins

- To find the minimum number of US coins to make any amount, the greedy method always works
  - At each step, just choose the largest coin that does not overshoot the desired amount: 31¢=25
- The greedy method would not work if we did not have 5¢ coins
  - For 31 cents, the greedy method gives seven coins (25+1+1+1+1+1+1), but we can do it with four (10+10+10+1)
- The greedy method also would not work if we had a 21¢ coin
  - For 63 cents, the greedy method gives six coins (25+25+10+1+1+1), but we can do it with three (21+21+21)
- How can we find the minimum number of coins for any given coin set?

Coin set for examples

- For the following examples, we will assume coins in the following denominations:
  1¢ 5¢ 10¢ 21¢ 25¢
- We'll use 22¢ and 63¢ as our goal

- This example is taken from:
  Data Structures & Problem Solving using Java *by* Mark Allen Weiss

A simple solution

- We always need a 1¢ coin, otherwise no solution exists for making one cent
- To make K cents:
  - If there is a K-cent coin, then that one coin is the minimum
  - Otherwise, for each value i < K,
    - Find the minimum number of coins needed to make i cents
    - Find the minimum number of coins needed to make K - i cents
  - Choose the i that minimizes this sum
- This algorithm can be viewed as divide-and-conquer, or as brute force
  - This solution is very recursive
  - It requires exponential work

A simple solution

Example: collect 7 cents

Simple code for the coin problem

```
public static void main(String[] args) {
    int[] coins = { 1, 5, 10, 21, 25 };
    int solveForThisAmount = 63;
    int[] solution;
    // try simple solution
    solution = makeChange1(coins, solveForThisAmount);
    System.out.println("solution: " +
                       Arrays.toString(solution));
    // try dynamic programming solution
    solution = makeChange2(coins, solveForThisAmount);
    System.out.println("solution: " +
                       Arrays.toString(solution));
}
```

makeChange1, part 1

```java
/**
 * Find the minimum number of coins required.
 * @param coins The available kinds of coins.
 * @param n The desired total.
 * @return An array of how many of each coin.
 */
private static int[] makeChange1(int[] coins, int n) {
    int numberOfDifferentCoins = coins.length;
    int[] solution;
    // if there is a single coin with value n, use it
    for (int i = 0; i < numberOfDifferentCoins; i += 1) {
        if (coins[i] == n) {
            solution = new int[numberOfDifferentCoins];
            solution[i] = 1;
            return solution;
        }
    }
    // else try all combinations of i and n-i coins
    ...
}
```

makeChange1, part 2

```
    // else try all combinations of i and n-i coins
    solution = new int[numberOfDifferentCoins];
    int leastNumberOfCoins = Integer.MAX_VALUE;
    for (int i = 1; i < n; i += 1) {
        int[] solution1 = makeChange1(coins, i);
        int[] solution2 = makeChange1(coins, n - i);
        int newCoinCount =
            totalCoinCount(solution1, solution2);
        if (newCoinCount < leastNumberOfCoins) {
            leastNumberOfCoins = newCoinCount;
            solution = arraySum(solution1, solution2);
        }
    }
    return solution;
}
```

Another solution

- We can reduce the problem recursively by choosing the first coin, and solving for the amount that is left
- For 63¢:
  - One 1¢ coin plus the best solution for 62¢
  - One 5¢ coin plus the best solution for 58¢
  - One 10¢ coin plus the best solution for 53¢
  - One 21¢ coin plus the best solution for 42¢
  - One 25¢ coin plus the best solution for 38¢
- Choose the best solution from among the 5 given above
- Instead of solving 62 recursive problems, we solve 5
- This is still a very expensive algorithm

A dynamic programming solution

- Idea: Solve first for one cent, then two cents, then three cents, etc., up to the desired amount
  - *Save each answer in an array !*
- For each new amount N, compute all the possible pairs of previous answers which sum to N
  - For example, to find the solution for 13¢,
    - First, solve for all of 1¢, 2¢, 3¢, ..., 12¢
    - Next, choose the best solution among:
      - Solution for 1¢ + solution for 12¢
      - Solution for 2¢ + solution for 11¢
      - Solution for 3¢ + solution for 10¢
      - Solution for 4¢ + solution for 9¢
      - Solution for 5¢ + solution for 8¢
      - Solution for 6¢ + solution for 7¢

Example

- Suppose coins are 1¢, 3¢, and 4¢
  - There's only one way to make 1¢ (one coin)
  - To make 2¢, try 1¢+1¢ (one coin + one coin = 2 coins)
  - To make 3¢, just use the 3¢ coin (one coin)
  - To make 4¢, just use the 4¢ coin (one coin)
  - To make 5¢, try
    - 1¢ + 4¢ (1 coin + 1 coin = 2 coins)
    - 2¢ + 3¢ (2 coins + 1 coin = 3 coins)
    - The first solution is better, so best solution is 2 coins
  - To make 6¢, try
    - 1¢ + 5¢ (1 coin + 2 coins = 3 coins)
    - 2¢ + 4¢ (2 coins + 1 coin = 3 coins)
    - 3¢ + 3¢ (1 coin + 1 coin = 2 coins) – best solution
  - Etc.

makeChange2

```
/**
 * Find the minimum number of coins required.
 * @param coins The available kinds of coins.
 * @param desiredTotal The desired total.
 * @return An array of how many of each coin.
 */
public static int[] makeChange2(int[] coins,
                                int desiredTotal) {
    int numberOfDifferentCoins = coins.length;
    int[][] solutions = new int[desiredTotal + 1][];
    int[] best = new int[desiredTotal + 1];
    solutions[0] = new int[numberOfDifferentCoins];
    best[0] = 0;
    for (int n = 1; n <= desiredTotal; n++) {
        solveFor2(n, coins, solutions, best);
    }
    return solutions[desiredTotal];
}
```

solveFor2, part 1

```
/**
 * Finds the minimum number of coins for each value <= n.
 * @param n The largest amount to be solved for.
 * @param coins The available kinds of coins.
 * @param solutions Arrays of how many of each kind of coin.
 * @param best The number of coins in each solution array.
 */
private static void solveFor2(int n, int[] coins,
                              int[][] solutions, int[] best) {
    int numberOfDifferentCoins = coins.length;
    // if there is a single coin with the value n, use it
    for (int i = 0; i < numberOfDifferentCoins; i += 1) {
        if (coins[i] == n) {
            solutions[n] =
                new int[numberOfDifferentCoins];
            solutions[n][i] = 1;
            best[i] = 1;
            return;
        }
    }
    // else try all combinations of i and n-i coins
    ...
}
```

solveFor2, part 2

```
    // else try all combinations of i and n-i coins
    int leastNumberOfCoins = Integer.MAX_VALUE;
    for (int i = 1; i < n; i += 1) {
        int coinCount = best[i] + best[n - i];
        if (coinCount < leastNumberOfCoins) {
            leastNumberOfCoins = coinCount;
            best[n] = coinCount;
            solutions[n] = arraySum(solutions[i],
                                    solutions[n - i]);
        }
    }
}
```

How good is the algorithm?

- The first algorithm is recursive, with a branching factor of up to 62
  - Possibly the average branching factor is somewhere around half of that (31)
  - The algorithm takes exponential time, with a large base
- The second algorithm is much better—it has a branching factor of 5
  - This is exponential time, with base 5
- The dynamic programming algorithm is O(N*K), where N is the desired amount and K is the number of different kinds of coins

Comparison with divide-and-conquer

- Divide-and-conquer algorithms split a problem into separate subproblems, solve the subproblems, and combine the results for a solution to the original problem
  - Example: Quicksort
  - Example: Mergesort
  - Example: Binary search
- Divide-and-conquer algorithms can be thought of as top-down algorithms
- In contrast, a dynamic programming algorithm proceeds by solving small problems, remembering the results, then combining them to find the solution to larger problems
- Dynamic programming can be thought of as bottom-up

Comparison with divide-and-conquer

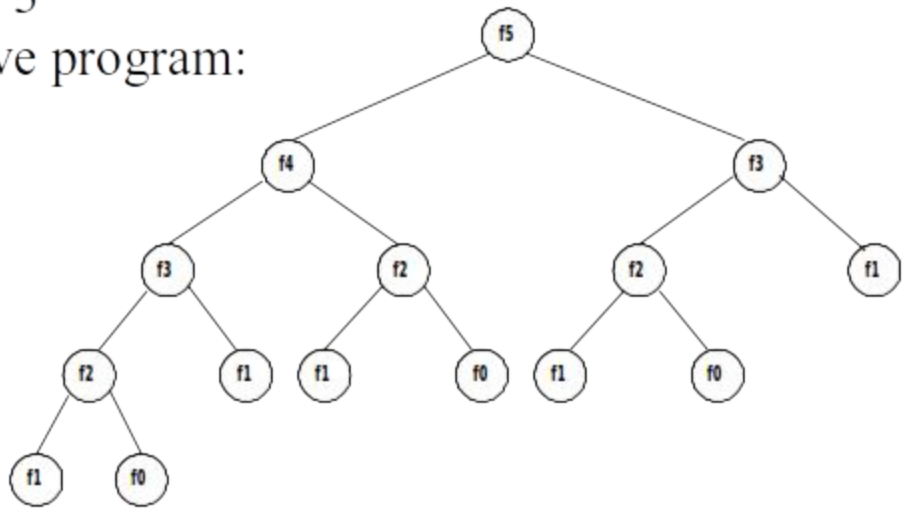
- Like divide-and-conquer, solve problem by combining the solutions to sub-problems.
- Differences between divide-and-conquer and DP:
  - Independent sub-problems, solve sub-problems independently and recursively, (so same sub(sub)problems solved repeatedly)
  - Sub-problems are dependent, i.e., sub-problems share sub-sub-problems, every sub(sub)problem solved just once, solutions to sub(sub)problems are stored in a table and used for solving higher level sub-problems.
- DP reduces computation by
  - Solving subproblems in a bottom-up fashion.
  - Storing solution to a subproblem the first time it is solved.
  - Looking up the solution when subproblem is encountered again.

Fibonacci sequence (1/4)

- Fibonacci sequence: 1 , 1 , 2 , 3 , 5 , 8 , 13 , 21 , …

  $F_i = 1$ if $i == 1$ or $i == 2$ (assume $F_0 = 0$)

  $F_i = F_{i-1} + F_{i-2}$ if $i \geq 3$
- Solved by a recursive program:



- Much replicated computation is done.
- It should be solved by a simple loop.

Fibonacci Sequence (2/4)

- Recursive logic:
  - $F_1 = F_2 = 1$
  - If $i > 2$ then $F_i = F_{i-1} + F_{i-2}$
- Directly translates into a recursive algorithm F:

```
F(i) {
    if (i = 1) or (i = 2)
      x ← 1
    else
      x ← F(i-1) + F(i-2)
    return x
    }
```

- F's call tree is **exponential**; F is recomputed many times for the same input value!

- We can speed things up by storing output values of F in an array A_F

```
F(i) {
     if (A_F != NULL) return
A_F
    if (i = 1) or (i = 2)
      x ← 1
    else
      x ← F(i-1) + F(i-2)
    A_F ← x
    return x
    }
```

- Since there are *n* cells in A_F and each cell takes $O(1)$ time to compute, this is $O(n)$!

Example 2: Binomial Coefficients

- $(x + y)^2 = x^2 + 2xy + y^2$, coefficients are 1,2,1
- $(x + y)^3 = x^3 + 3x^2y + 3xy^2 + y^3$, coefficients are 1,3,3,1
- $(x + y)^4 = x^4 + 4x^3y + 6x^2y^2 + 4xy^3 + y^4$,
  coefficients are 1,4,6,4,1
- $(x + y)^5 = x^5 + 5x^4y + 10x^3y^2 + 10x^2y^3 + 5xy^4 + y^5$,
  coefficients are 1,5,10,10,5,1
- The n+1 coefficients can be computed for $(x + y)^n$ according to the formula c(n, i) = n! / (i! * (n - i)!)
  for each of i = 0..n
- The repeated computation of all the factorials gets to be expensive
- We can use dynamic programming to save the factorials as we go

Solution by dynamic programming

n	c(n,0)	c(n,1)	c(n,2)	c(n,3)	c(n,4)	c(n,5)	c(n,6)
0	1						
1	1	1					
2	1	2	1				
3	1	3	3	1			
4	1	4	6	4	1		
5	1	5	10	10	5	1	
6	1	6	15	20	15	6	1

Each row depends only on the preceding row

Only linear space and quadratic time are needed

This algorithm is known as Pascal's Triangle

$c(n,k) \rightarrow (1+x)^n \quad .. \; c \; x^k$

Solution by dynamic programming

n	c(n,0)	c(n,1)	c(n,2)	c(n,3)	c(n,4)	c(n,5)	c(n,6)
0	1						
1	1	1					
2	1	2	1				
3	1	3	3	1			
4	1	4	6	4	1		

Each row depends only on the preceding row

Only linear space and quadratic time are needed

This algorithm is known as Pascal's Triangle

$c(n,k) \rightarrow (1+x)^n \quad .. \ c\ x^k$

$c(n,k) \ = c(n-1,k-1)+c(n-1,k)$

$c(n,0) = c(n,1) = 1$

Solution by dynamic programming

Optimal Substructure

Let's work it out for n=4 and k=2.

Answer: 6

We know:

- $C(n, k) = C(n-1, k-1) + C(n-1, k)$
- $C(n, 0) = C(n, n) = 1$

$C(4, 2)$ = Sum of

- $C(3, 1)$ = Sum of
  - $C(2, 0) = 1$
  - $C(2, 1)$ = Sum of
    - $C(1, 0) = 1$
    - $C(1, 1) = 1$
- $C(3, 2)$ = Sum of
  - $C(2, 1)$ = Sum of
    - $C(1, 1) = 1$
    - $C(1, 0) = 1$
  - $C(2, 2) = 1$

Recursive Solution

```
// Returns value of Binomial Coefficient C(n, k)
int binomialCoeff(int n, int k)
{
  // Base Cases
  if (k==0 || k==n)
    return 1;

  // Recur
  return  binomialCoeff(n-1, k-1) + binomialCoeff(n-1, k);
}
```

```
                     C(5, 2)
                /                     \
     C(4, 1)                           C(4, 2)
       /   \                          /           \
C(3, 0)   C(3, 1)             C(3, 1)               C(3, 2)
            /    \             /    \                /    \
  C(2, 0)      C(2, 1)     C(2, 0) C(2, 1)       C(2, 1)  C(2, 2)
                /       \            /   \         /    \
      C(1, 0)  C(1, 1)      C(1, 0)  C(1, 1)  C(1, 0)  C(1, 1)
```

Dynamic Programming

Construct a temporary array C[][] in a bottom-up manner

```
// Returns value of Binomial Coefficient C(n, k)
int binomialCoeff(int n, int k)
{
    int C[n+1][k+1];
    int i, j;

    // Caculate value of Binomial Coefficient in bottom up manner
    for (i = 0; i <= n; i++)
    {
        for (j = 0; j <= min(i, k); j++)
        {
            // Base Cases
            if (j == 0 || j == i)
                C[i][j] = 1;

            // Calculate value using previosly stored values
            else
                C[i][j] = C[i-1][j-1] + C[i-1][j];
        }
    }

    return C[n][k];
}
```

The algorithm in Java

```java
public static int binom(int n, int m) {
    int[ ] b = new int[n + 1];
    b[0] = 1;
    for (int i = 1; i <= n; i++) {
        b[i] = 1;
        for (int j = i - 1; j > 0; j--) {
            b[j] += b[j - 1];
        }
    }
    return b[m];
}
```

- Source: Data Structures and Algorithms with Object-Oriented Design Patterns in Java *by* Bruno R. Preiss

The principle of optimality, I

- Dynamic programming is a technique for finding an *optimal* solution
- The principle of optimality applies if the optimal solution to a problem always contains optimal solutions to all subproblems
- Example: Consider the problem of making N¢ with the fewest number of coins
  - Either there is an N¢ coin, or
  - The set of coins making up an optimal solution for N¢ can be divided into two nonempty subsets, n_1¢ and n_2¢
    - If either subset, n_1¢ or n_2¢, can be made with fewer coins, then clearly N¢ can be made with fewer coins, hence solution was *not* optimal

The principle of optimality, II

- The principle of optimality holds if
  - Every optimal solution to a problem contains...
  - ...optimal solutions to all subproblems
- The principle of optimality does *not* say
  - If you have optimal solutions to all subproblems...
  - ...then you can combine them to get an optimal solution
- Example: In US coinage,
  - The optimal solution to 7¢ is 5¢ + 1¢ + 1¢, *and*
  - The optimal solution to 6¢ is 5¢ + 1¢, *but*
  - The optimal solution to 13¢ is *not* 5¢ + 1¢ + 1¢ + 5¢ + 1¢
- But there is *some* way of dividing up 13¢ into subsets with optimal solutions (say, 11¢ + 2¢) that will give an optimal solution for 13¢
  - Hence, the principle of optimality holds for this problem

The 0-1 knapsack problem

- A thief breaks into a house, carrying a knapsack...
  - He can carry up to 25 pounds of loot
  - He has to choose which of N items to steal
    - Each item has some weight and some value
    - "0-1" because each item is stolen (1) or not stolen (0)
  - He has to select the items to steal in order to maximize the value of his loot, but cannot exceed 25 pounds
- A greedy algorithm does not find an optimal solution
- A dynamic programming algorithm works well
- This is similar to, but not identical to, the coins problem
  - In the coins problem, we had to make an *exact* amount of change
  - In the 0-1 knapsack problem, we can't *exceed* the weight limit, but the optimal solution may be *less* than the weight limit
  - The dynamic programming solution is similar to that of the coins problem

Comments

- Dynamic programming relies on working "from the bottom up" and saving the results of solving simpler problems
  - These solutions to simpler problems are then used to compute the solution to more complex problems
- Dynamic programming solutions can often be quite complex and tricky
- Dynamic programming is used for optimization problems, especially ones that would otherwise take exponential time
  - Only problems that satisfy the principle of optimality are suitable for dynamic programming solutions
- Since exponential time is unacceptable for all but the smallest problems, dynamic programming is sometimes essential

The End