Universitas Esa Unggul

Smart, Creative and Entrepreneurial

www.esaunggul.ac.id

CMC 101 TOPIK DALAM PEMROGRAMAN

PERTEMUAN 13

PROGRAM STUDI MAGISTER ILMU KOMPUTER

FAKULTAS ILMU KOMPUTER

TOPIK DALAM PEMROGRAMAN Greedy Algorithms & Dynamic Programming

Pertemuan 13

TUJUAN PERKULIAHAN

- Mahasiswa memahami beberapa tipe persoalan yang penting.
  - Greedy Algorithms
  - Dynamic Programming

Greedy Algorithms

Optimization problems

- An optimization problem is one in which you want to find, not just *a* solution, but the *best* solution
- A "greedy algorithm" sometimes works well for optimization problems
- A greedy algorithm works in phases. At each phase:
  - You take the best you can get right now, without regard for future consequences
  - You hope that by choosing a *local* optimum at each step, you will end up at a *global* optimum

Example: Counting money

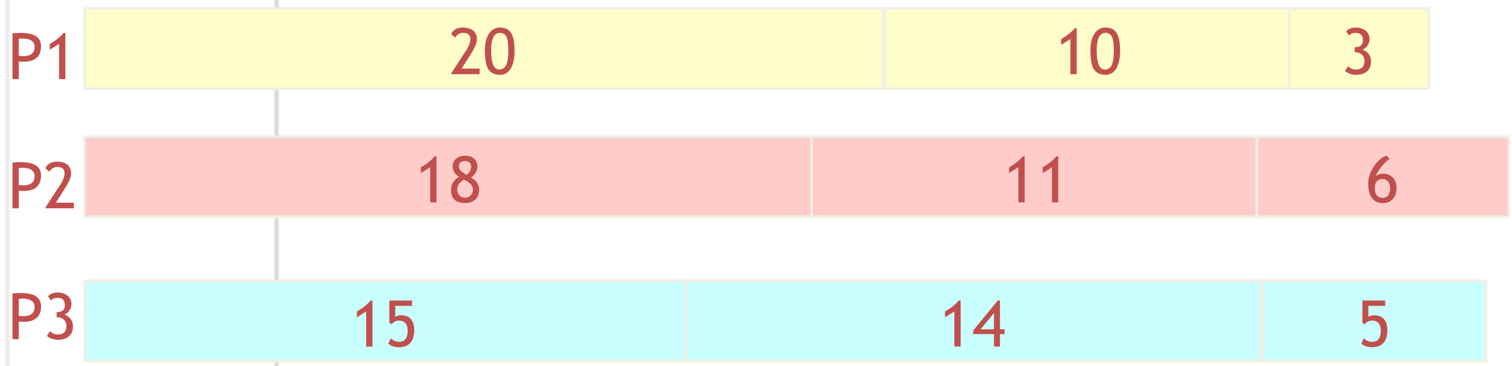
- Suppose you want to count out a certain amount of money, using the fewest possible bills and coins
- A greedy algorithm would do this would be: At each step, take the largest possible bill or coin that does not overshoot
  - Example: To make $6.39, you can choose:
    - a $5 bill
    - a $1 bill, to make $6
    - a 25¢ coin, to make $6.25
    - A 10¢ coin, to make $6.35
    - four 1¢ coins, to make $6.39
- For US money, the greedy algorithm always gives the optimum solution

A failure of the greedy algorithm

- In some (fictional) monetary system, "krons" come in 1 kron, 7 kron, and 10 kron coins
- Using a greedy algorithm to count out 15 krons, you would get
  - A 10 kron piece
  - Five 1 kron pieces, for a total of 15 krons
  - This requires six coins
- A better solution would be to use two 7 kron pieces and one 1 kron piece
  - This only requires three coins
- The greedy algorithm results in a solution, but not in an optimal solution

A scheduling problem

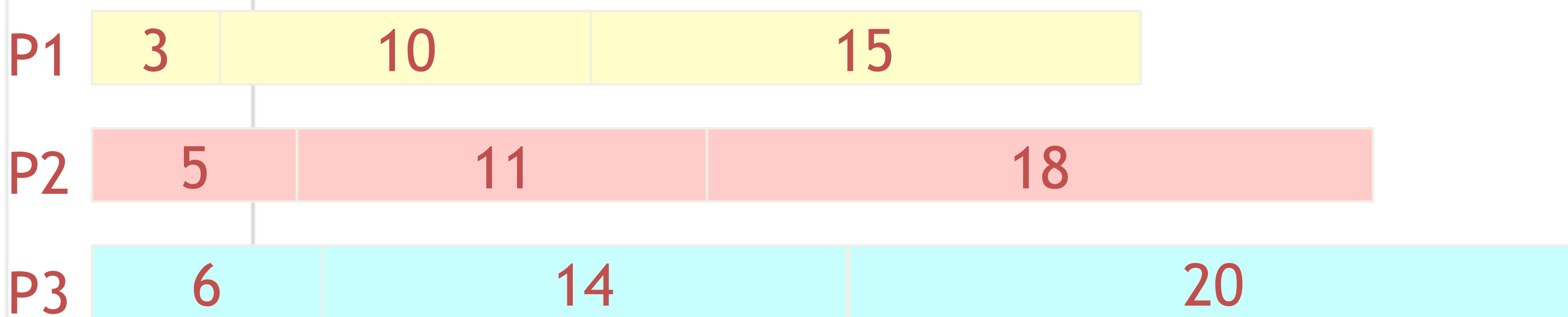
- You have to run nine jobs, with running times of 3, 5, 6, 10, 11, 14, 15, 18, and 20 minutes
- You have three processors on which you can run these jobs
- You decide to do the longest-running jobs first, on whatever processor is available



- Time to completion: 18 + 11 + 6 = 35 minutes
- This solution isn't bad, but we might be able to do better

Another approach

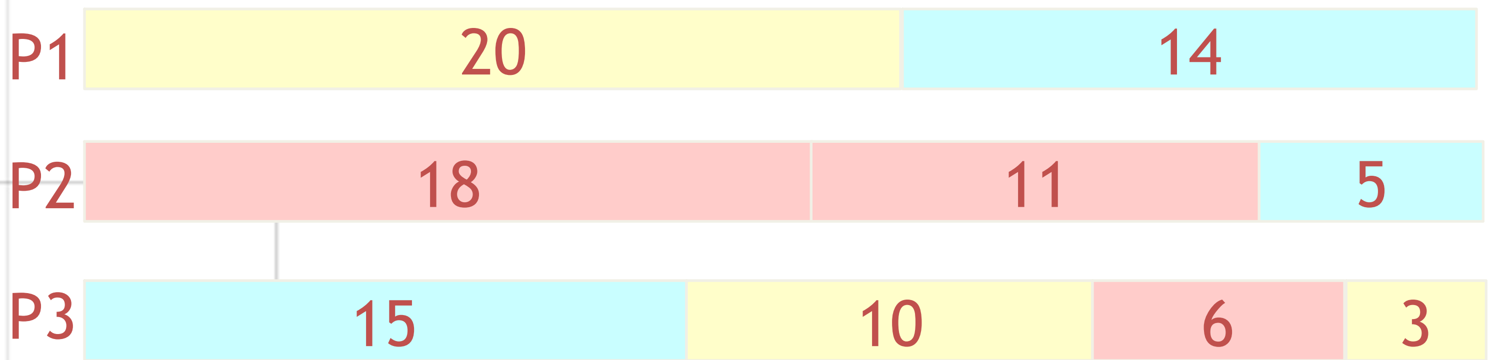
- What would be the result if you ran the *shortest* job first?
- Again, the running times are 3, 5, 6, 10, 11, 14, 15, 18, and 20 minutes



- That wasn't such a good idea; time to completion is now 6 + 14 + 20 = 40 minutes
- Note, however, that the greedy algorithm itself is fast
  - All we had to do at each stage was pick the minimum or maximum

An optimum solution

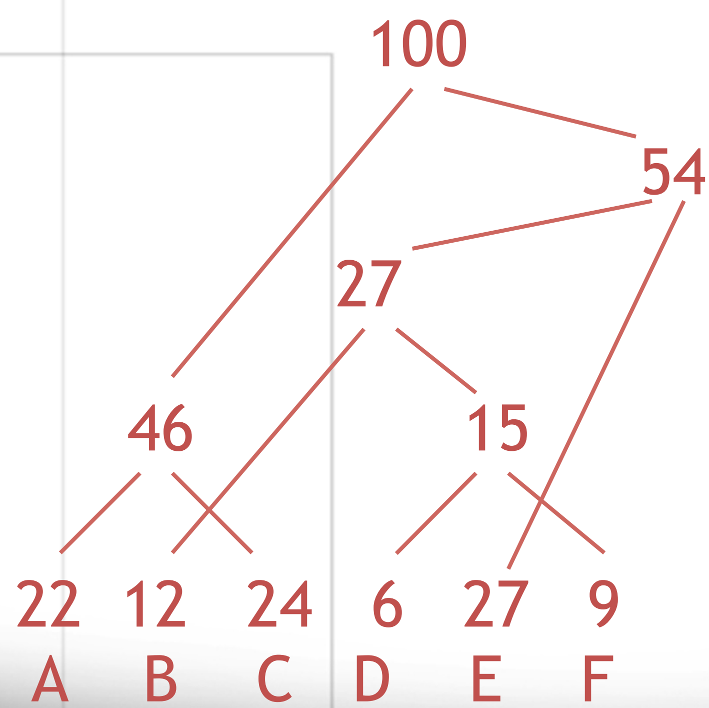
- Better solutions do exist:



- This solution is clearly optimal (why?)
- Clearly, there are other optimal solutions (why?)
- How do we find such a solution?
  - One way: Try all possible assignments of jobs to processors
  - Unfortunately, this approach can take exponential time

Huffman encoding

- The Huffman encoding algorithm is a greedy algorithm
- You always pick the two smallest numbers to combine

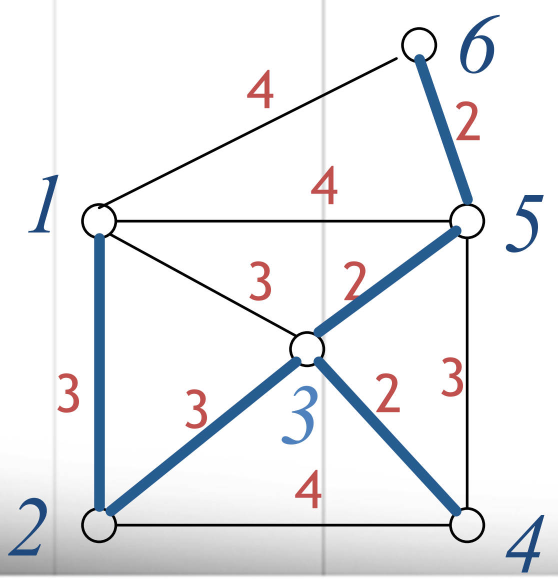


A=00
B=100
C=01
D=1010
E=11
F=1011

- Average bits/char: 0.22*2 + 0.12*3 + 0.24*2 + 0.06*4 + 0.27*2 + 0.09*4 = 2.42
- The Huffman algorithm finds an optimal solution

Minimum spanning tree

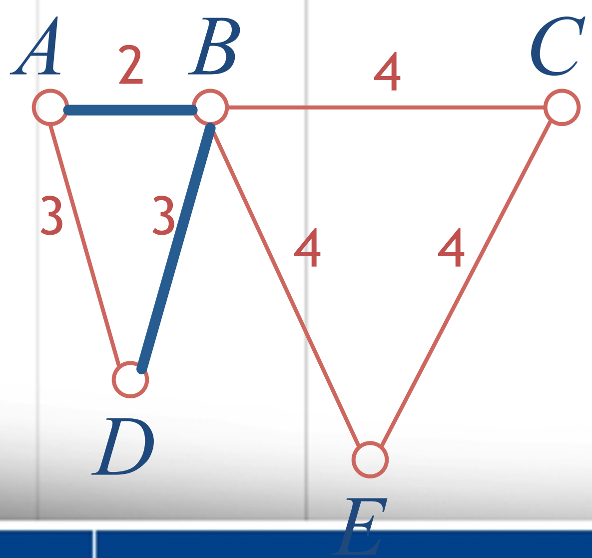
- A minimum spanning tree is a least-cost subset of the edges of a graph that connects all the nodes
  - Start by picking any node and adding it to the tree
  - Repeatedly: Pick any *least-cost* edge from a node in the tree to a node not in the tree, and add the edge and new node to the tree
  - Stop when all nodes have been added to the tree



- The result is a least-cost (3+3+2+2+2=12) spanning tree
- If you think some other edge should be in the spanning tree:
  - Try adding that edge
  - Note that the edge is part of a cycle
  - To break the cycle, you must remove the edge with the greatest cost
    - This will be the edge you just added

Traveling salesman

- A salesman must visit every city (starting from city ***A***), and wants to cover the least possible distance
  - He can revisit a city (and reuse a road) if necessary
- He does this by using a greedy algorithm: He goes to the next nearest city from wherever he is



- From ***A*** he goes to ***B***
- From ***B*** he goes to ***D***
- This is *not* going to result in a shortest path!
- The best result he can get now will be ***ABDBCE***, at a cost of 16
- An actual least-cost path from ***A*** is ***ADBCE***, at a cost of 14

Analysis

- A greedy algorithm typically makes (approximately) n choices for a problem of size n
  - (The first or last choice may be forced)
- Hence the expected running time is: $O(n * O(choice(n)))$, where $choice(n)$ is making a choice among n objects
  - Counting: Must find largest useable coin from among k sizes of coin (k is a constant), an $O(k)=O(1)$ operation;
    - Therefore, coin counting is (n)
  - Huffman: Must sort n values before making n choices
    - Therefore, Huffman is $O(n \log n) + O(n) = O(n \log n)$
  - Minimum spanning tree: At each new node, must include new edges and keep them sorted, which is $O(n \log n)$ overall
    - Therefore, MST is $O(n \log n) + O(n) = O(n \log n)$

Other greedy algorithms

- Dijkstra's algorithm for finding the shortest path in a graph
  - Always takes the *shortest* edge connecting a known node to an unknown node
- Kruskal's algorithm for finding a minimum-cost spanning tree
  - Always tries the *lowest-cost* remaining edge
- Prim's algorithm for finding a minimum-cost spanning tree
  - Always takes the *lowest-cost* edge between nodes in the spanning tree and nodes not yet in the spanning tree

Dijkstra's shortest-path algorithm

- Dijkstra's algorithm finds the shortest paths from a given node to all other nodes in a graph
  - Initially,
    - Mark the given node as *known* (path length is zero)
    - For each out-edge, set the distance in each neighboring node equal to the *cost* (length) of the out-edge, and set its *predecessor* to the initially given node
  - Repeatedly (until all nodes are known),
    - Find an unknown node containing the smallest distance
    - Mark the new node as known
    - For each node adjacent to the new node, examine its neighbors to see whether their estimated distance can be reduced (distance to known node plus cost of out-edge)
      - If so, also reset the predecessor of the new node

Analysis of Dijkstra's algorithm I

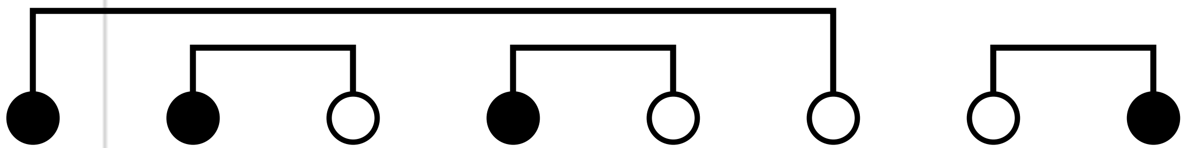
- Assume that the *average* out-degree of a node is some constant k
  - Initially,
    - Mark the given node as *known* (path length is zero)
      - This takes O(1) (constant) time
    - For each out-edge, set the distance in each neighboring node equal to the *cost* (length) of the out-edge, and set its *predecessor* to the initially given node
      - If each node refers to a list of k adjacent node/edge pairs, this takes O(k) = O(1) time, that is, constant time
      - Notice that this operation takes *longer* if we have to extract a list of names from a hash table

Analysis of Dijkstra's algorithm II

- Repeatedly (until all nodes are known), (n times)
  - Find an unknown node containing the smallest distance
    - Probably the best way to do this is to put the unknown nodes into a priority queue; this takes k * O(log n) time *each* time a new node is marked "known" (and this happens n times)
  - Mark the new node as known -- O(1) time
  - For each node adjacent to the new node, examine its neighbors to see whether their estimated distance can be reduced (distance to known node plus cost of out-edge)
    - If so, also reset the predecessor of the new node
    - There are k adjacent nodes (on average), operation requires constant time at each, therefore O(k) (constant) time
  - Combining all the parts, we get:
    O(1) + n*(k*O(log n)+O(k)), that is, O(nk log n) time

Connecting wires

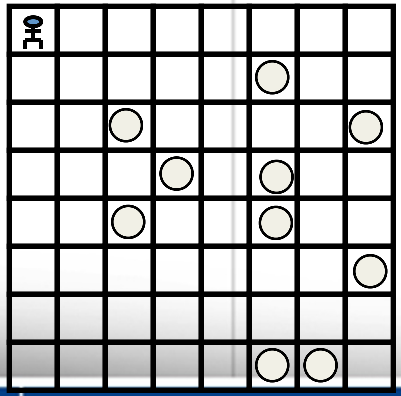
- There are n white dots and n black dots, equally spaced, in a line
- You want to connect each white dot with some one black dot, with a minimum total length of "wire"
- Example:



- Total wire length above is 1 + 1 + 1 + 5 = 8
- Do you see a greedy algorithm for doing this?
- Does the algorithm guarantee an optimal solution?
  - Can you prove it?
  - Can you find a counterexample?

Collecting coins

- A checkerboard has a certain number of coins on it
- A robot starts in the upper-left corner, and walks to the bottom left-hand corner
  - The robot can only move in two directions: right and down
  - The robot collects coins as it goes
- You want to collect *all* the coins using the *minimum* number of robots
- Example:



- Do you see a greedy algorithm for doing this?
- Does the algorithm guarantee an optimal solution?
  - Can you prove it?
  - Can you find a counterexample?

Dynamic Programming

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 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
  - It is *infeasible* to solve for 63¢

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.

The algorithm in Java

```
• public static void makeChange(int[] coins, int differentCoins,
                                int maxChange, int[] coinsUsed,
                                int[] lastCoin) {
    coinsUsed[0] = 0; lastCoin[0] = 1;
    for (int cents = 1; cents < maxChange; cents++) {
        int minCoins = cents;
        int newCoin = 1;
        for (int j = 0; j < differentCoins; j++) {
            if (coins[j] > cents) continue; // cannot use coin
            if (coinsUsed[cents – coins[j]] + 1 < minCoins) {
                minCoins = coinsUsed[cents – coins[j]] + 1;
                newCoin = coins[j];
            }
        }
        coinsUsed[cents] = minCoins;
        lastCoin[cents] = newCoin;
    }
}
```

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, then combining them to find the solution to larger problems
- Dynamic programming can be thought of as bottom-up

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

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

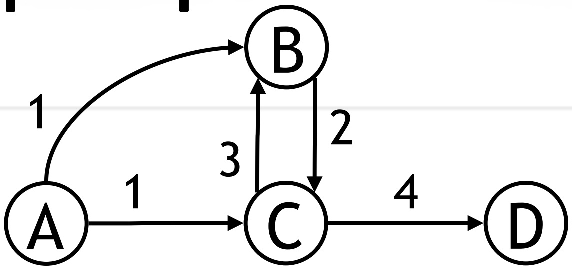
- Dynamic programming is a technique for finding an *optimal* solution
- The principle of optimality applies if the optimal solution to a problem always contains optimal solutions to all subproblems
- Example: Consider the problem of making N¢ with the fewest number of coins
  - Either there is an N¢ 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

Longest simple path

- Consider the following graph:



- The longest simple path (path not containing a cycle) from A to D is A B C D
- However, the subpath A B is not the longest simple path from A to B (A C B is longer)
- The principle of optimality is not satisfied for this problem
- Hence, the longest simple path problem cannot be solved by a dynamic programming approach

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