CompSci 230
Discrete Math for Computer Science

October 8, 2013
Prof. Rodger

Announcements

• Read for next time Chap. 3.1-3.3
• No recitation Oct 11 or 14

Interview Time ....
We will solve a Microsoft interview question
Chap. 3.1 Algorithms

**Definition:** An *algorithm* is a finite set of precise instructions for performing a computation or for solving a problem.

**Example:** Describe an algorithm for finding the maximum value in a finite sequence of integers.

**Solution:** Perform the following steps:
1. Set the temporary maximum equal to the first integer in the sequence.
2. Compare the next integer in the sequence to the temporary maximum.
   - If it is larger than the temporary maximum, set the temporary maximum equal to this integer.
3. Repeat the previous step if there are more integers. If not, stop.
4. When the algorithm terminates, the temporary maximum is the largest integer in the sequence.

**Specifying Algorithms**
- Algorithms can be specified in English or in *pseudocode*.
- Pseudocode is an intermediate step between an English and coding using a programming language.
- Appendix 3 specifies pseudocode for this book (similar to Java).
- Pseudocode helps analyze the time required to solve a problem using an algorithm, independent of the actual programming language used to implement algorithm.

**Properties of Algorithms**
- *Input:* An algorithm has input values from a specified set.
- *Output:* From the input values, the algorithm produces the output values from a specified set. The output values are the solution.
- *Correctness:* An algorithm should produce the correct output values for each set of input values.
- *Finiteness:* An algorithm should produce the output after a finite number of steps for any input.
- *Effectiveness:* It must be possible to perform each step of the algorithm correctly and in a finite amount of time.
- *Generality:* The algorithm should work for all problems of the desired form.

**Finding the Maximum Element in a Finite Sequence**
- The algorithm in pseudocode:

```plaintext
procedure max(a_1, a_2, ..., a_n: integers)
    max := a_1
    for i := 2 to n
        if max < a_i then max := a_i
    return max {max is the largest element}
```

- Does this algorithm have all the properties listed on the previous slide?
Problem

• Describe an algorithm that determines whether a function from a finite set of integers to another finite set of integers is onto.

Solution: Algorithm

• Assume A has n elements, B has m elements.

Some Example Algorithm Problems

• Three classes of problems will look at in this chapter
  1. *Searching Problems*: finding the position of a particular element in a list.
  2. *Sorting problems*: putting the elements of a list into increasing order.
  3. *Optimization Problems*: determining the optimal value (maximum or minimum) of a particular quantity over all possible inputs.

Greedy Algorithms

• *Optimization problems* minimize or maximize some parameter over all possible inputs.
• Examples:
  – Finding a route between two cities with the smallest total mileage.
  – Determining how to encode messages using the fewest possible bits.
• Solved using a *greedy algorithm*, which makes the “best” choice at each step. Making the “best choice” at each step does not necessarily produce an optimal solution to the overall problem, but in many instances, it does.
• Try to prove that this approach always produces an optimal solution, or find a counterexample to show that it does.
Greedy Algorithms: Making Change

**Example:** Design a greedy algorithm for making change (in U.S. money) of \( n \) cents with the following coins: quarters (25 cents), dimes (10 cents), nickels (5 cents), and pennies (1 cent), using the least total number of coins.

**Idea:** At each step choose the coin with the largest possible value that does not exceed the amount of change left.

1. If \( n = 0 \) cents, first choose \( \frac{1}{4} \) quarter leaving \( 0 \) cent.

**Greedy Change-Making Algorithm**

**Solution:** Greedy change-making algorithm for \( n \) cents. The algorithm works with any coin denominations \( c_1, c_2, \ldots, c_r \).

```plaintext
procedure change(c_1, c_2, \ldots, c_r; \text{values of coins, where } c_1 > c_2 > \ldots > c_r; \text{ } n: \text{a positive integer})
for i := 1 to r
    d_i := 0 \{d_i \text{ counts the coins of denomination } c_i\}
while n \geq c_i
    d_i := d_i + 1 \{add a coin of denomination } c_i\}
    n := n - c_i
[d_i \text{ counts the coins } c_i]
```

- For the example of U.S. currency, we may have quarters, dimes, nickels and pennies, with \( c_1 = 25, c_2 = 10, c_3 = 5, \text{and } c_4 = 1 \).

**What is the brute-force way to show Optimality for U.S. Coins?**

**Proving Optimality for U.S. Coins**

- Show that the change making algorithm for U.S. coins is optimal.

**Lemma 1:** If \( n \) is a positive integer, then \( n \) cents in change using quarters, dimes, nickels, and pennies, using the fewest coins possible has at most 2 dimes, 1 nickel, 4 pennies, and cannot have 2 dimes and a nickel. The total amount of change in dimes, nickels, and pennies must not exceed 24 cents.

**Proof:** By contradiction
Proving Optimality for U.S. Coins

**Theorem:** The greedy change-making algorithm for U.S. coins produces change using the fewest coins possible.

**Proof:** By contradiction.

1. Assume there is a positive integer \( n \) such that change can be made for \( n \) cents using quarters, dimes, nickels, and pennies, with a fewer total number of coins than given by the algorithm.

2. Then, \( q \) is the number of quarters used in this optimal way and \( q' \) is the number of quarters in the greedy algorithm's solution. But this is not possible by Lemma.

Greedy Change-Making Algorithm

- Optimality depends on the denominations available.
- For U.S. coins, optimality still holds if we add half dollar coins (50 cents) and dollar coins (100 cents).
- But if we allow only quarters (25 cents), dimes (10 cents), and pennies (1 cent), the algorithm no longer produces the minimum number of coins.
  - Give an example amount that it doesn’t work for.

Scheduling problems are huge!

- How did Duke schedule the rooms assigned to courses for this semester?
- How do they schedule students to take courses?

Greedy Scheduling

**Example:** We have a group of proposed talks with start and end times. Construct a greedy algorithm to schedule as many as possible in a lecture hall, under the following assumptions:

- When a talk starts, it continues till the end.
- No two talks can occur at the same time.
- A talk can begin at the same time that another ends.
- Once we have selected some of the talks, we cannot add a talk which is incompatible with those already selected because it overlaps at least one of these previously selected talks.
What algorithm should we use to pick talks?

• The talk that starts earliest among those compatible with already chosen talks?
• The talk that is shortest among those already compatible?
• The talk that ends earliest among those compatible with already chosen talks?

Greedy Scheduling algorithm

Solution: At each step, choose the talks with the earliest ending time among the talks compatible with those selected.

```
procedure schedule(s_1 \leq s_2 \leq \ldots \leq s_n: start times, e_1 \leq e_2 \leq \ldots \leq e_n: end times)
osort talks by finish time and reorder so that e_1 \leq e_2 \leq \ldots \leq e_n
S := \emptyset
for j := 1 to n
    if talk j is compatible with S then
        S := S \cup \{talk j\}
return S [S is the set of talks scheduled]
```

Halting Problem

• One of the most famous computer science problems
• Can you decide if a program you wrote will ever halt on an input?

Halting Problem

Example: Can we develop a procedure that takes as input a computer program along with its input and determines whether the program will eventually halt with that input.

• Solution: Proof by contradiction.
• Assume that there is such a procedure and call it \(H(P,I)\). The procedure \(H(P,I)\) takes as input a program \(P\) and the input \(I\) to \(P\).
  - \(H\) outputs “halt” if it is the case that \(P\) will stop when run with input \(I\).
  - Otherwise, \(H\) outputs “loops forever.”
Halting Problem

• Since a program is a string of characters, we can call \( H(P,P) \). Construct a procedure \( K(P) \), which works as follows.
  – If \( H(P,P) \) outputs “loops forever” then \( K(P) \) halts.
  – If \( H(P,P) \) outputs “halt” then \( K(P) \) goes into an infinite loop printing “ha” on each iteration.

Halting Problem

• Now we call \( K \) with \( K \) as input, i.e. \( K(K) \).
  – If the output of \( H(K,K) \) is “loops forever” then \( K(K) \) halts. A Contradiction.
  – If the output of \( H(K,K) \) is “halts” then \( K(K) \) loops forever. A Contradiction.

• Therefore, there cannot be a procedure that can decide whether or not an arbitrary program halts. The halting problem is unsolvable.