CS 2305: Discrete Mathematics for Computing I

Lecture 19

- KP Bhat

Greedy Algorithms

Optimization problems minimize or maximize some parameter over all possible inputs.

- Finding a route between two cities with the smallest total mileage.
- Determining how to encode messages using the fewest possible bits.
- Finding the fiber links between network nodes using the least amount of fiber.

Optimization problems can often be 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.

After specifying what the "best choice" at each step is, we try to prove that this approach always produces an optimal solution, or find a counterexample to show that it does not.

- This is a very important step because in some cases greedy algorithms yield a very poor solution

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 = 67 cents, first choose a quarter leaving 67−25 = 42 cents. Then choose another quarter leaving 42 −25 = 17 cents
2. Then choose 1 dime, leaving 17 − 10 = 7 cents.
3. Choose 1 nickel, leaving 7 – 5 = 2 cents.
4. Choose a penny, leaving one cent. Choose another penny leaving 0 cents.

Greedy Change-Making Algorithm 1

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

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

For the example of U.S. currency, we may have quarters, dimes, nickels and pennies, with $c_1 = 25$, $c_2 = 10$, $c_3 = 5$, and $c_4 = 1$.

Proving Optimality for U.S. Coins 1

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

- If we had 3 dimes, we could replace them with a quarter and a nickel.
- If we had 2 nickels, we could replace them with 1 dime.
- If we had 5 pennies, we could replace them with a nickel.
- If we had 2 dimes and 1 nickel, we could replace them with a quarter.
- ∴ 24 cents (2 dimes and 4 pennies) is the most money we can have in dimes, nickels, and pennies when we make change using the fewest number of coins

Proving Optimality for U.S. Coins 2

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, $q' \leq q$ where 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 1, since the value of the coins other than quarters can not be greater than 24 cents.

3. Similarly, by Lemma 1, the two algorithms must have the same number of dimes, nickels, and quarters.

- The value of the coins other than quarters and dimes can not be greater than 9 cents.
- The value of the coins other than quarters, dimes and nickels can not be greater than 4 cents.

Greedy Change-Making Algorithm 2

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.

- Consider the example of 30 cents. The optimal number of coins is 3, i.e., 3 dimes. However the algorithm uses 6 coins—1 quarter and 5 pennies

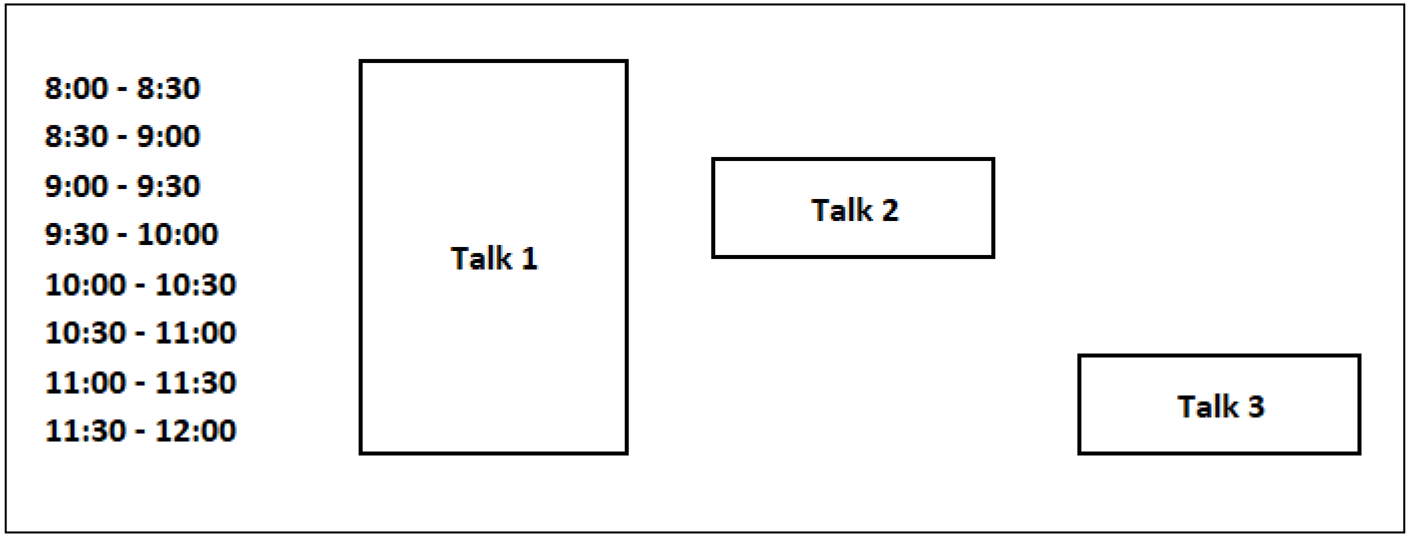
Greedy Scheduling 1

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.
- How should we make the "best choice" at each step of the algorithm? That is, which talk do we pick ?
  - 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 2

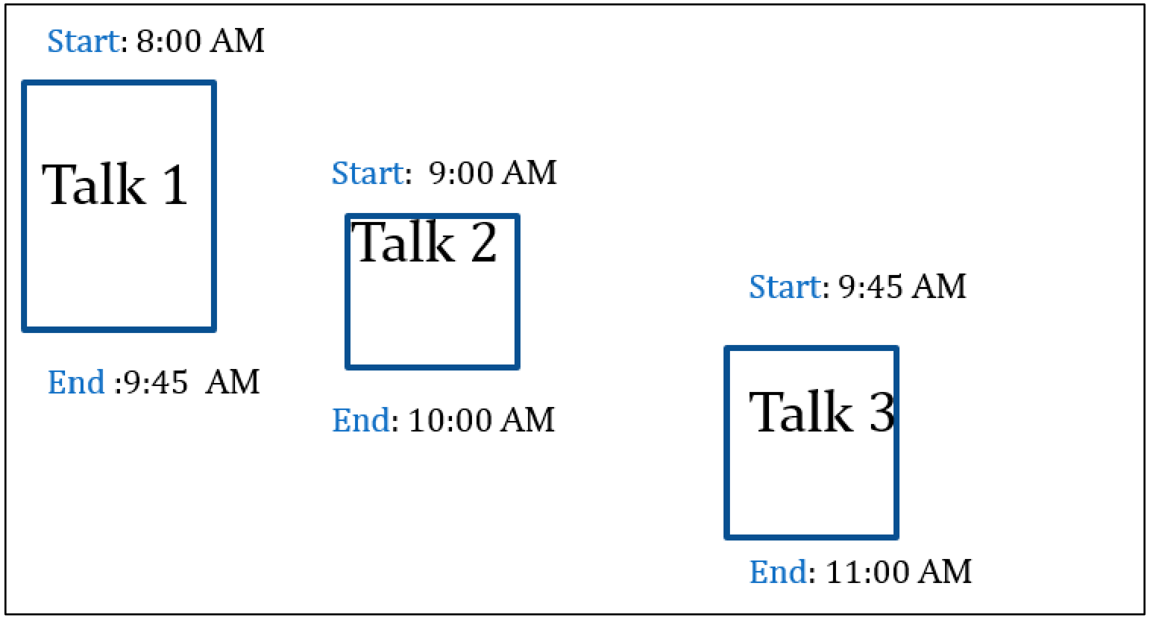
Picking the earliest talk doesn't work.



For example by picking Talk 1 first we select only 1 talk, although we can easily accommodate 2 talks if we first select Talk 2 first

Greedy Scheduling 3

Picking the shortest talk doesn't work either



For example by picking Talk 2 first we can select only 1 talk, although we can easily accommodate Talk 1 and Talk 3, since they do not overlap

However greedy scheduling works by picking the talk that ends soonest. The algorithm is specified on the next slide.

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)
sort 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
        $S := S \cup \{\text{talk } j\}$
return S [ S is the set of talks scheduled]
```

A talk is incompatible with those already selected if it overlaps at least one of them

Will be proven correct by induction in Chapter 5.

Unsolvable Problems

- There is a class of problems for which it can be shown that no algorithm exists (i.e. cannot be solved using any procedure)
- Typically these problems require a yes/no answer, but where there cannot possibly be any algorithm that always gives the correct answer
- The most famous problem in this category is The Halting Problem
- For more examples:
  - https://en.wikipedia.org/wiki/List_of_undecidable_problems

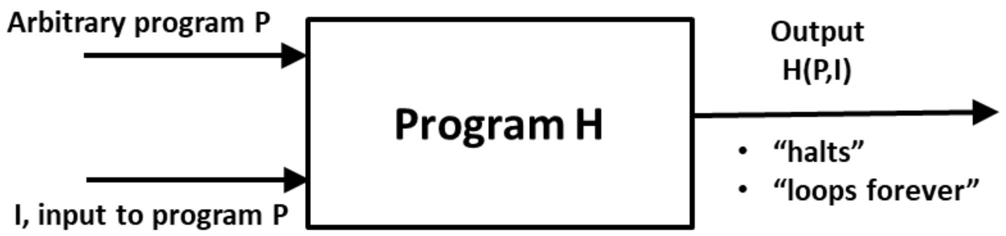
Halting Problem[1]

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.

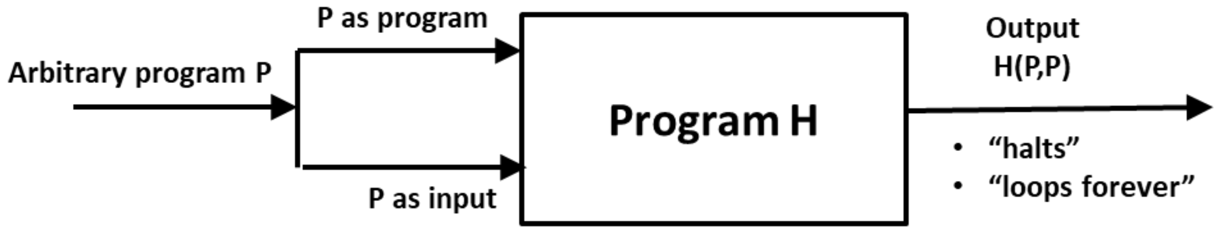
Assume that there is such a procedure and call it *H*(*P*,*I*). The procedure *H*(*P*,*I*) takes as input a program *P* and *I* the input to program *P*.

- *H* outputs "halt" if it is the case that *P* will stop when run with input *I*.
- Otherwise, *H* outputs "loops forever."



Halting Problem 2

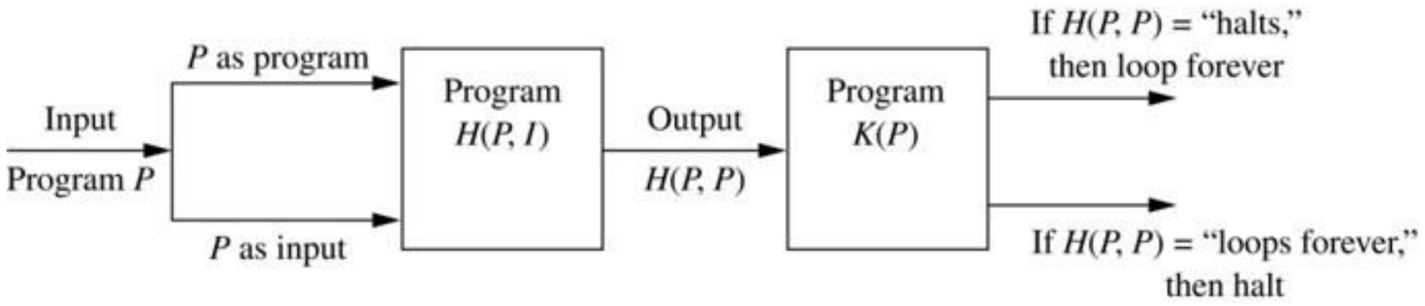
Since a program is a string of characters, we can invoke *H* using *P* both as the program as well as its input i.e. *H*(*P*,*P*). In this case *H* tells us whether or not **<u>any</u>** program *P* will halt when it is given a copy of itself as input



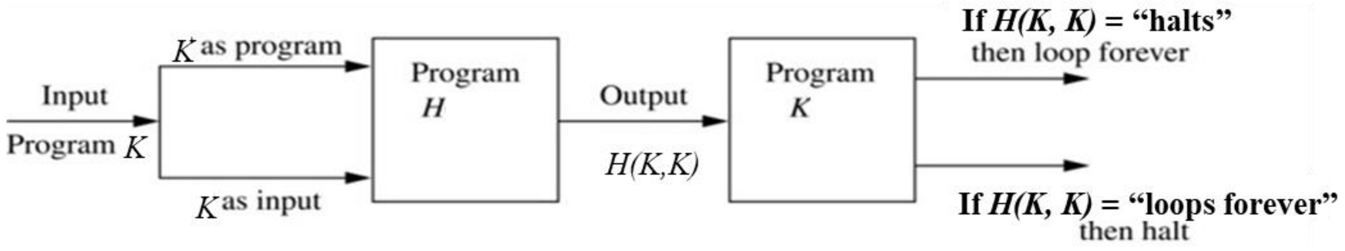
Halting Problem

Construct a procedure *K*(*P*), whose input is the output of Procedure H i.e. H(P, P). The procedure *K*(*P*) 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.


Input
Program P
P as program
P as input
Program H(P, I)
Output H(P, P)
Program K(P)
If H(P, P) = "halts," then loop forever
If H(P, P) = "loops forever," then halt

Halting Problem 4


Input
Program K
K as program
K as input
Program H
Output
$H(K,K)$
Program K
If $H(K, K)$ = "halts" then loop forever
If $H(K, K)$ = "loops forever" then halt

Now let us invoke *H* using *K* both as the program as well as its input, and feed the output to K.

There are two possibilities for the output of Procedure *H*

Possibility 1: *H(K, K)* = "loops forever"

- In this case Procedure *K* halts. This contradicts the claim of *H(K, K)* that Program *K* will loop forever on input *K*

Possibility 2: H(K, K) = "halts"

- In this case Procedure *K* loops for ever. This contradicts the claim of *H(K, K)* that Program *K* halts on input *K*

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

The Growth of Functions

Section 3.2

The Growth of Functions

In both computer science and in mathematics, there are many times when we care about how fast a function grows.

In computer science, we want to understand how quickly an algorithm can solve a problem as the size of the input grows.

- We can compare the efficiency of two different algorithms for solving the same problem.
- We can also determine whether it is practical to use a particular algorithm as the input grows.

Factors Affecting the Running Time of a Program

1. The size of the input to the program
2. The time complexity of the algorithm underlying the program
3. The quality of code generated by the compiler used to create the object program
4. The nature and speed of the instructions on the machine used to execute the program

- Factors #1 and #2 are the dominant factors
- Factors #3 and #4 are, to a very large extent, independent of factors #1 and #2 and they can be approximated by some constant

Big-*O* Notation [1]

Definition: Let *f* and *g* be functions from the set of integers or the set of real numbers to the set of real numbers. We say that *f*(*x*) is *O*(*g*(*x*)) if there are constants *C* and *k* such that

$$|f(x)| \leq C|g(x)|$$

whenever $x > k$. (illustration on next slide)

This is read as "*f*(*x*) is big-*O* of *g*(*x*)" or "*g* asymptotically dominates *f*."

The constants C and k are called *witnesses* to the relationship *f*(*x*) is *O*(*g*(*x*)). Only one pair of witnesses is needed.

Note: *f* (*x*) is *O*(*g*(*x*)) says that *f* (*x*) <u>grows slower</u> than some fixed multiple of *g*(*x*) as *x* grows without bound.