

# CPSC 510: Algorithms

## Unit 3: Greedy Algorithms

Reading: Ch. 4

# Outline

- **Introduction to Greedy Algorithms**
- Example: Interval Scheduling
- Greedy Algorithm Proofs
- Additional Examples

# Greedy Algorithm Example

Consider you want to give  $x$  cents in change to a person using US coins (\$1, 50¢, 25¢, 10¢, 5¢, 1¢). Develop an algorithm that minimizes the number of coins you give to the person. Assume you have an unlimited supply of each type of coin.

ⓐ GIVE-CHANGE( $x$ )  
1 Give as many \$1 coins as possible. Reduce  $x$  appropriately.  
2 " " " 50¢ " " " " " "  
3 " " " 25¢  
4 " " " 10¢  
5 " " " 5¢  
6 " " " 1¢

# Greedy Algorithms

- **Greedy algorithms** consist of a series of small steps. At each step, choose the choice that looks best at the moment.
  - In other words, a series of local optimal choices will result in a global optimal solution.
- Greedy algorithms are commonly applied to optimization problems – problems which seek an optimal solution.
- There are two challenges with greedy algorithms:
  - Determining what is “the choice that look best at the moment”.
  - Proving that greedy algorithm does or does not produce an optimal solution.

# Greedy Algorithm Example Revisited

Reconsider the change giving problem. This time the coin denominations are \$1, 50¢, 30¢, 15¢, 5¢, 1¢. Does the greedy algorithm of choosing as many of the largest denomination coins each step work? *No*

*Counter example. If  $x$  is 60¢, greedy would choose (50¢, 5¢, 5¢)  
Optimal: (30¢, 30¢).*

# Outline

- Introduction to Greedy Algorithms
- **Example: Interval Scheduling**
- Greedy Algorithm Proofs
- Additional Examples

# Example: Interval Scheduling

To illustrate the difficulty of selecting the best choice, we will use the example of interval scheduling.

Assume we have a single conference room and  $n$  meetings that would like to use the conference room. Each meeting  $i$  has a starting time  $s(i)$  and finishing time  $f(i)$  with  $s(i) < f(i)$ . Develop an algorithm that schedules as many of the  $n$  meetings as possible.

# Example: Interval Scheduling

A basic structure of a greedy algorithm is this. Let  $M$  be the set of  $n$  meetings and  $S$  be the resulting schedule.

SCHEDULE( $M$ ):

- 1 let  $S$  be an empty set
- 2 while  $M$  is not empty:
  - 3     remove the best meeting  $m$  from  $M$
  - 4     add  $m$  to  $S$
  - 5     remove all meetings that overlap with  $m$  from  $M$
- 6 return  $S$

# Example: Interval Scheduling

The key to the greedy algorithm is determining what the best meeting is in step 3. Here are some possibilities:

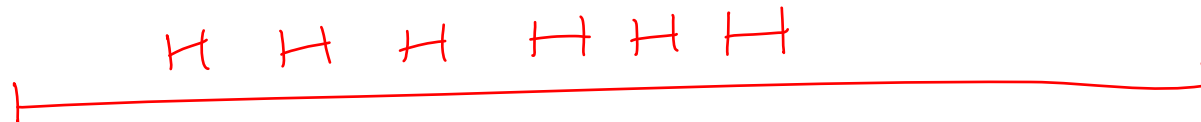
- a. The meeting with the earliest start time ( $s(i)$  is smallest).
- b. The meeting with the earliest end time ( $f(i)$  is smallest).
- c. The meeting with the shortest duration ( $f(i) - s(i)$  is smallest).
- d. The meeting that overlaps with the fewest other meetings.

Only one of these is correct. Which one?

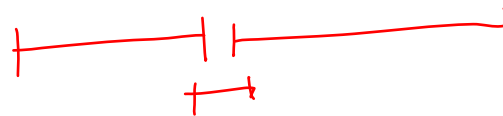
# Example: Interval Scheduling

Counterexamples for the incorrect cases:

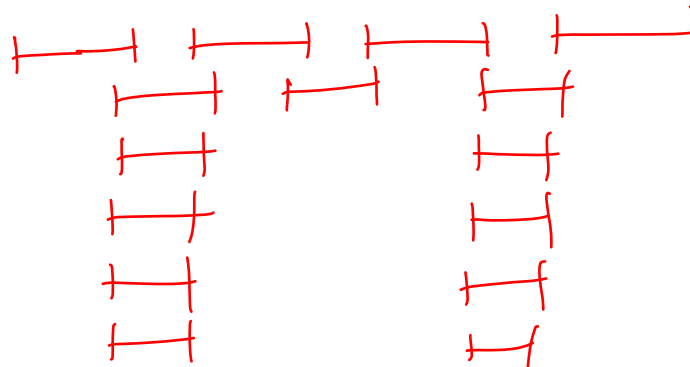
a. Earliest start time



c. Smallest duration



d. Lowest overlap



# Example: Interval Scheduling

What is the run time of this algorithm in the worst-case?

$O(n^2)$

SCHEDULE(M):

1 let S be an empty set

2 while M is not empty:  $O(n)$

3     remove the best meeting  $m$  from M  $O(n)$

4     add  $m$  to S

5     remove all meetings that overlap with  $m$  from M  $O(n)$

6 return S

# Example: Interval Scheduling

- Is this algorithm optimal? Does a faster algorithm exist?

*No, a faster one exists*

*Sort by finishing time*

# Example: Interval Scheduling

SCHEDULE(M):

1 let S be an empty set

2 sort M by their finishing time  $O(n \lg n)$

3 let  $time = 0$  // assuming that all times are greater than 0

4 for  $i = 1$  to  $M.length$

$O(n)$  5 if  $s(M[i]) \geq time$

6 add M[i] to S

7  $time = f(M[i])$

8 return S

$O(n \lg n)$

# Greedy Algorithms and Recursion

- Another way of implementing the scheduling problem is to use recursion:

SCHEDULE(M):

1 if M is empty

2 return  $\emptyset$

3 pick the best meeting  $m$  from M

4 let M' be the set of meetings from M that do not overlap with  $m$

5 return  $\{m\} \cup \text{SCHEDULE}(M')$

- It can be helpful to think of greedy algorithms in this fashion.
  - Each step (except the last) results in a single smaller instance of the problem.
  - The last step is typically an empty instance of the problem.

# Outline

- Introduction to Greedy Algorithms
- Example: Interval Scheduling
- **Greedy Algorithm Proofs**
- Additional Examples

# Greedy Choice Property

- Greedy algorithms exhibit the **greedy choice property**: a globally optimal solution by making locally optimal (greedy) choices.
  - Need to prove that this is case.
- Once proven, greedy algorithms are often faster than algorithms, such as dynamic programming, that consider a wider range of choices.

# Greedy Algorithm Proofs: Proof by Contradiction

- One proof strategy is to use proof by contradiction in this fashion:
  - Let  $G$  be a solution returned by the greedy algorithm.
  - Let  $O$  be an optimal solution.
  - Assume that  $G$  is not optimal.
  - Show a contradiction such as:
    - $O$  cannot be optimal.
    - The algorithm cannot produce  $G$ .
    - $O$  and  $G$  must be the same size.
- A common tactic is to show that the greedy algorithm “stays ahead” of any other algorithm.

# Example: Interval Scheduling Proof

- In interval scheduling, prove that the best choice leads to an optimal solution.

# Greedy Algorithm Proofs: Exchange Argument

- Another proof strategy is to use an exchange argument like this:
  - Let  $G$  be a solution returned by the greedy algorithm.
  - Let  $O$  be an optimal solution not returned by the greedy algorithm.
  - Gradually modify  $O$ , using a series of transformations that preserve optimality, until you arrive at the same answer as  $G$ .
- This proof is typically applied to algorithms that produce an optimal ordering.
  - A common transformation is to “exchange” two items in the ordering.

# Outline

- Introduction to Greedy Algorithms
- Example: Interval Scheduling
- Greedy Algorithm Proofs
- **Additional Examples**

# Example 1: Interval Scheduling with Deadlines

Assume we have a single resource and  $n$  requests that would like to use the resource. Each request  $i$  has a contiguous time requirement of  $t(i)$  and a deadline  $d(i)$ . The request can be scheduling at any time, ideally at a time such that it can finish at or before the deadline. Only one request can be handled by the resource at a time. Develop an algorithm that produces a schedule where the maximum lateness is minimized.

$$lateness(i) = \begin{cases} 0 & f(i) \leq d(i) \\ f(i) - d(i) & f(i) > d(i) \end{cases} \quad f(i) \text{ is the finishing time}$$

# Example 1 Algorithm

SCHEDULE\_WITH\_DEADLINES(R):

- 1 sort R by their deadlines (earliest first)
- 2 return R

Greedy choice is to select the job with the earliest deadline.

# Example 1 Proof

# Example 2: Cell Tower Placement

Imagine a long straight east-west road in the countryside that has houses scattered sparsely along it. Unfortunately the residents of these houses do not have cell phone coverage for far too long. Finally, a cell-phone company is interested in installing cell phone base stations along the road. A house has coverage if it is within four miles of a base station. Develop an algorithm that makes sure every house along the road is within four miles of a base station while minimizing the number of base stations.

# Example 2 Algorithm

Let  $H$  be the set of houses represented by how far they are away from the west end of the road and  $L$  be the length of the road in miles.

PLACE-BASE-STATIONS( $H, L$ ):

- 1 let  $B$  be an empty set
- 2 sort  $H$  by their distance from west to east
- 3 let  $distance = 0$  // distance from west end of road
- 4 for  $i = 1$  to  $H.length$
- 5     if  $H[i] \geq distance$
- 6         add  $\min(H[i] + 4, L)$  to  $B$
- 7          $distance = H[i] + 8$
- 8 return  $B$

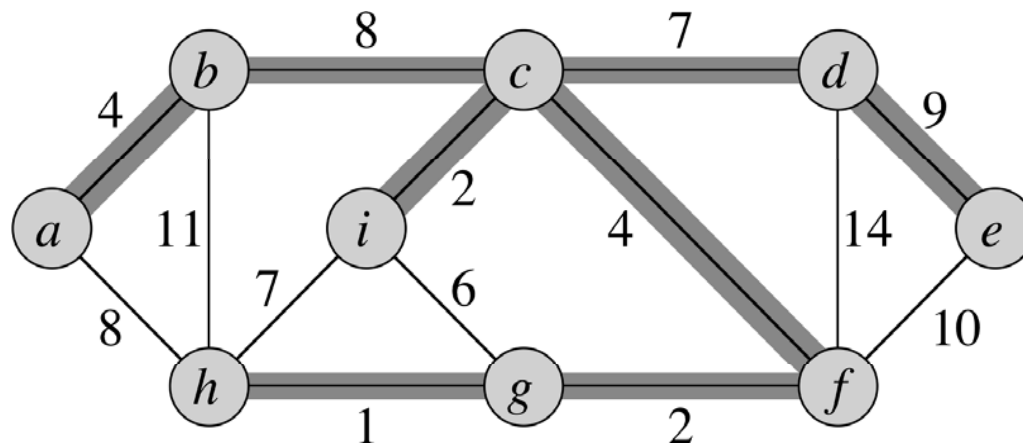
This is a greedy algorithm where each step we place the westernmost base station as far east as possible without missing any houses.

# Example 2 Proof

# Example 3: Minimum Spanning Trees

- A **minimum spanning tree** of an undirected weighted connected graph is a tree such that...
  - all edges in the tree are edges in the graph.
  - all vertices in the graph are in the tree.
  - the edges are chosen such that the total weight of the tree is minimized.

Example:



# Example 3: Minimum Spanning Trees

- To find a minimum spanning tree for a graph, we are going to use a greedy algorithm that grows the tree one edge at a time. Generically the algorithm looks like this:

GENERIC-MST(G):

- 1 let  $A$  be an empty set
- 2 while  $A$  does not form a spanning tree
- 3     find an edge  $(u, v)$  that is safe for  $A$
- 4      $A = A \cup \{(u, v)\}$
- 5 return  $A$

- What is the greedy choice for step 3?

# Example 3: Kruskal's Algorithm

In Kruskal's algorithm, the greedy choice is to pick the edge (out of the remaining edges) that has the least weight that does not introduce a cycle. It's faster to sort the edges by their weights first and then check to see if they part of the spanning tree.

MST-KRUSKAL(G):

- 1 let A be an empty set
- 2 let  $E_{sorted} = \text{sort } G.E$  into ascending order by their weight  $O(E \lg E)$
- 3 for each edge  $(u, v)$  in  $E_{sorted}$
- 4 if  $A \cup \{(u, v)\}$  is acyclic  $O(V)$
- 5  $A = A \cup \{(u, v)\}$
- 6 return A  $O(VE)$

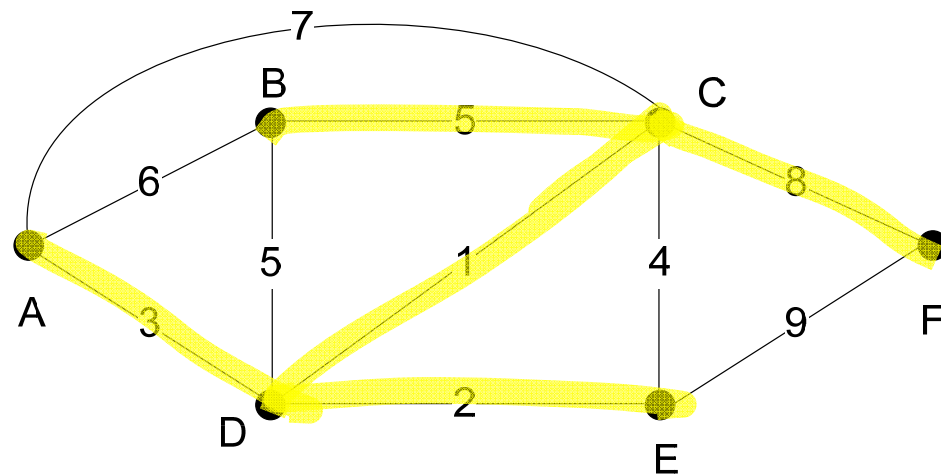
$O(E)$   
iteration

$$V \leq E < V^2$$

$$\lg E < V \text{ when } V \geq 4$$

$$O(E \lg E) + O(VE) = O(VE)$$

# Example 3: Kruskal's Algorithm



# Example 3: Run-Time Analysis

- What is the running time of this algorithm?

# Example 3: Additional Comments

- A faster algorithm with  $O(E \lg E)$  worst-case running time can be implemented using the Union-Find data structure described in section 4.6.
- An alternative algorithm is Prim's algorithm.
  - Unlike Kruskal's algorithm, the set  $A$  at each step of the algorithm is a tree (rather than a forest) that will eventually grow into the minimum spanning tree.
  - The tree is started by arbitrarily selecting a vertex to be the root.
  - The greedy choice is the edge with the smallest weight that connects a vertex in the tree to a vertex outside the tree.
- Proofs that the greedy choices made by both Kruskal's and Prim's algorithm lead to optimal solutions are presented in the text.

# Example 4: Data Compression with Huffman Codes

- A **Huffman code** is used for data compression for a file of information.
- Assume a file has  $n$  unique characters and each character  $c$  occurs  $c.freq$  times in the file. Each character is replaced by a unique codeword. The goal is to minimize the number of bits in the file.
- A **fixed-length codeword** requires codeword in the file takes the same number of bits ( $\lceil \lg n \rceil$ ).
  - For example, if the file only contained the lowercase characters a-f, three bits are needed to represent each character:  
a 000   b 001   c 010   d 011   e 100   f 101
  - If there are  $m$  characters in a file, it would take  $3m$  bits to represent the file.

# Example 4: Data Compression with Huffman Codes

- A better way is to use a *variable-length codeword* in which more frequently occurring letters are represented using fewer bits and less frequently occurring letters are represented using more bits.
- When designing the variable-length codeword, the code must not be a prefix of another bit representation.
  - For example, assume a is represented by 00 and b is represented by 001.
  - This violates the requirement since 00 is a prefix of 001. The decoding algorithm would not be able to tell if the bit string 001 is the character a (followed by the start of another character) or the character b.

# Example 4: Data Compression with Huffman Codes

Assume a file has the following frequencies:

a 60    b 5    c 15    d 17    e 11    f 2  
1    5    3    2    4    6 = 220 bits

How do we make a prefix code that minimizes the number of bits in the file? First attempt: order the character by frequency.

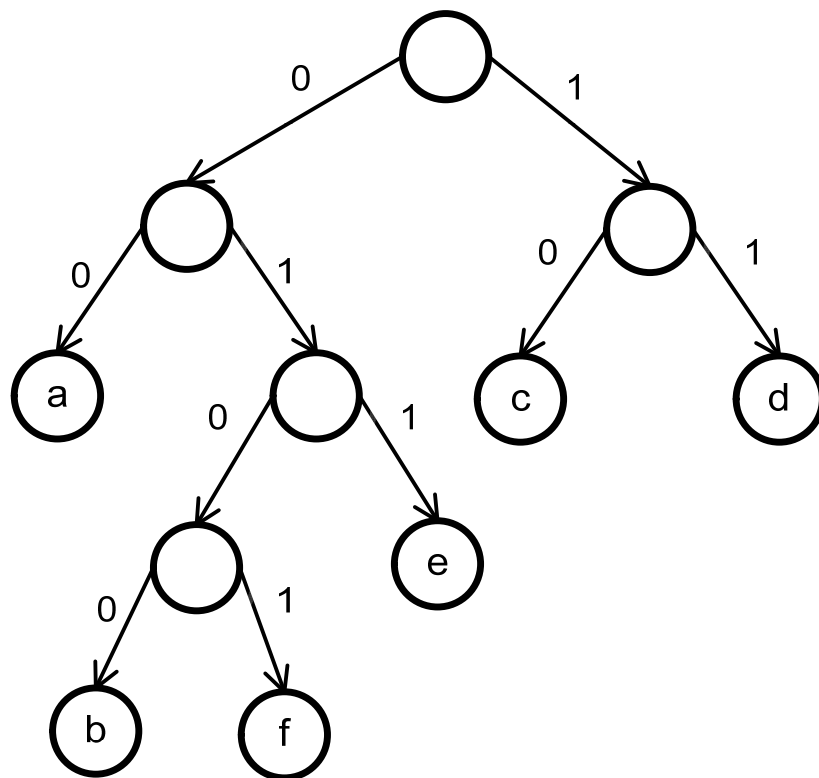
a 0  
d 10  
c 110  
e 1110  
b 11110  
f 111110

Fixed encoding  
= 3 × 110 = 330 bits

How many bits are needed for the file?

# Example 4: Data Compression with Huffman Codes

A Huffman code can be represented using a binary tree.



a: 00  
b: 0100  
c: 10  
d: 11  
e: 011  
f: 0101

# Example 4 Algorithm

Here is a greedy algorithm for constructing a binary tree resulting in an optimal Huffman code:

HUFFMAN(C):

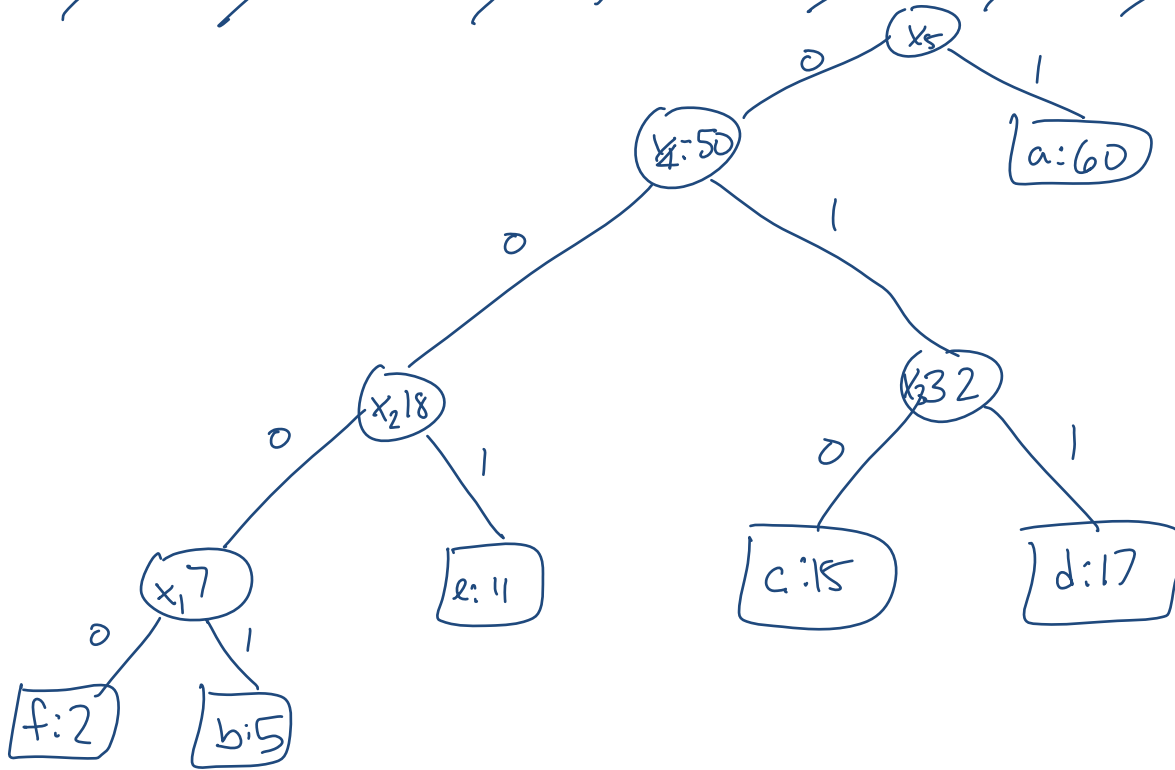
```
1   $n = |C|$ 
2  let Q be a min-priority queue (using the freq attribute) = C
3  for  $i = 1$  to  $n - 1$  // If there  $n$  leaves, there will be  $n - 1$  internal nodes in a full binary tree
4      allocate a new node  $z$ 
5       $z.left = x = \text{EXTRACT-MIN}(Q)$ 
6       $z.right = y = \text{EXTRACT-MIN}(Q)$ 
7       $z.freq = x.freq + y.freq$ 
8      INSERT(Q,  $z$ )
9  return EXTRACT-MIN(Q) // At the end, there is only one node left in Q – the root of the tree
```

The greedy choice is to create a node and merge the two least frequent characters.

# Example 4: Huffman Code Tree

Construct the tree and subsequent code for this set of characters and frequencies.

~~a 60~~ ~~b 5~~ ~~c 15~~ ~~d 17~~ ~~e 11~~ ~~f 2~~  ~~$x_1$~~   $x_2$  18  $x_3$  32  
 ~~$x_4$  50~~  
 $x_5$  110



a: 1  
b: 0001  
c: 010  
d: 011  
e: 001  
f: 0000