Key Concepts in Computer Science (COMP-1000)

Algorithms

Algorithms I

School of Computer Science
Faculty of Science
University of Windsor

Dr. Dima Alhadidi (Section 01)
Dr. Scott Goodwin (Section 30)

Fall 2021

University of Windsor

Algorithms I

Algorithms I - Outline

- Properties of Algorithms
- Algorithms for Searching
- Algorithms for Sorting

Problems and Algorithms

- In many domains there are key general problems that ask for **output** with specific properties when given **valid input**

- The first step is to precisely state the problem, using the appropriate structures to specify the **input** and the **desired output**

- We then solve the general problem by specifying the **steps of a procedure** that takes a **valid input** and produces the desired **output**
  - This procedure is called an ***algorithm***

Specifying Algorithms

- An **algorithm** is a sequence of unambiguous instructions for solving a specific problem
- Algorithms can be specified in different ways
  - Their steps can be described in English or in ***pseudocode***

- Programmers can use the description of an algorithm in pseudocode to construct a program in a particular language (e.g., Java or C++)

- Pseudocode helps us analyze the time required to solve a problem using an algorithm, **independent** of the actual programming language used to implement algorithm

Properties of Algorithms

- **Properties** of an algorithm
  - **Input**: an algorithm has input values from a specified set
  - **Output**: From each input, an algorithm produces an output from a specified set of values. The output is the solution to the problem
  - **Definiteness**: The steps of the algorithm must be defined precisely
  - **Finite**: an algorithm must eventually terminate
  - **Correctness**: each solution returned by the algorithm is correct
  - **Completeness**: the algorithm returns all of the correct solutions
  - **Effectiveness**: it must be possible to perform each step exactly and in a finite amount of time

- There might be many algorithms to solve the same problem
  - What is the difference between them?!

Finding the Value of the Maximum Element in a Finite Sequence

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

- Does this algorithm have all the properties listed on the previous slide?
  - √ **Input**: an algorithm has input values from a specified set
  - √ **Output**: From each input, an algorithm produces an output from a specified set of values. The output is the solution to the problem
  - √ **Definiteness**: The steps of the algorithm must be defined precisely
  - √ **Finite**: an algorithm must eventually terminate
  - √ **Correctness**: each solution returned by the algorithm is correct
  - √ **Completeness**: the algorithm returns all of the correct solutions
  - √ **Effectiveness**: it must be possible to perform each step exactly and in a finite amount of time

Finding the Position of the Maximum Element in a Finite Sequence

```
procedure maxPos($a_1$, $a_2$, …., $a_n$: integers)
    $max := a_1$ ;    $pos := 1$
    for $i := 2$ to $n$
        if $max < a_i$ then $max := a_i$ ;  $pos := i$
    return pos {pos is the position of the largest element}
```

- Does this algorithm have all the properties listed on the previous slide?
  - √ **Input**: an algorithm has input values from a specified set
  - √ **Output**: From each input, an algorithm produces an output from a specified set of values. The output is the solution to the problem
  - √ **Definiteness**: The steps of the algorithm must be defined precisely
  - √ **Finite**: an algorithm must eventually terminate
  - √ **Correctness**: each solution returned by the algorithm is correct
  - × **Completeness**: the algorithm returns all of the correct solutions
  - √ **Effectiveness**: it must be possible to perform each step exactly and in a finite amount of time

Some Example Algorithm Problems

- **Three** classes of problems will be studied in this section:

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

Searching Problems

Definition: The general *searching problem* is to locate an element ***x*** in the **list** of distinct elements $a_1, a_2, \ldots, a_n$, or determine that it is not in the list

- The solution to a searching problem is the location of the term in the list that equals ***x*** (that is, *i* is the solution if ***x*** $= a_i$) or 0 if ***x*** is not in the list.
- For example, a library might want to check to see if a patron is on a list of those with overdue books before allowing him/her to checkout another book
- We will study two different searching algorithms; linear search and binary search

Linear Search Algorithm

- The linear search algorithm locates an item in a list by examining elements in the sequence one at a time, starting at the beginning
- First compare ***x*** with a_1. If they are equal, return the position 1
- If not, try a_2. If $x = a_2$, return the position 2
- Keep going, and if no match is found when the entire list is scanned, return 0

```
**procedure** *linearSearch*($x$:integer, $a_1, a_2, \ldots, a_n$: distinct integers)
$i := 1$
**while** ($i \leq n$ and $x \neq a_i$)
    $i := i + 1$
**if** $i \leq n$ **then** *location* := $i$
**else** *location* := 0
**return** *location*
{*location* is the subscript of the term that equals $x$, or is 0 if $x$ is not found}
```

Binary Search

- Assume the input is a list of items in **increasing order**
- The algorithm begins by comparing the element to be found with the middle element
  - If the middle element is lower, the search proceeds with the upper half of the list
  - If it is not lower, the search proceeds with the lower half of the list (through the middle position)
- Repeat this process until we have a list of size 1
  - If the element we are looking for is equal to the element in the list, the position is returned
  - Otherwise, 0 is returned to indicate that the element was not found
- Later, we show that the binary search algorithm is much **more efficient** than linear search (assuming the given input list is sorted)

Binary Search

```
procedure binarySearch($x$: integer, $a_1, a_2, \ldots, a_n$: increasing integers)
 $i := 1$ {$i$ is the left endpoint of interval}
 $j := n$ {$j$ is right endpoint of interval}
 while $i < j$
        $m := \lfloor (i + j)/2 \rfloor$
        if $x > a_m$ then $i := m + 1$
        else $j := m$
  if $x = a_i$ then $location := i$
  else $location := 0$
  return $location$
  {location is the subscript $i$ of the term $a_i$ equal to $x$, or 0 if $x$ is not found}
```

- Note the calculation of the midpoint index m using the floor function.
- If the search value x is greater than the midpoint value a_m then the left index is changed to m=1, otherwise the right index is changed to m.

Binary Search

Example: The steps taken by a binary search for **19** in the list:

1 2 3 5 6 7 8 <u>10</u> 12 13 15 16 18 19 20 22

1. The list has 16 elements, so the midpoint is 8. The value in the 8th position is 10. Since 19 > 10, further search is restricted to positions 9 through 16

   1 2 3 5 6 7 8 10 12 13 15 <u>16</u> 18 19 20 22

2. The midpoint of the list (positions 9 through 16) is now the 12th position with a value of 16. Since 19 > 16, further search is restricted to the 13th position and above

   1 2 3 5 6 7 8 10 12 13 15 16 18 <u>19</u> 20 22

3. The midpoint of the current list is now the 14th position with a value of 19. Since 19 ≯ 19, further search is restricted to the portion from the 13th through the 14th positions

   1 2 3 5 6 7 8 10 12 13 15 16 <u>18</u> 19 20 22

4. The midpoint of the current list is now the 13th position with a value of 18 Since 19 > 18, search is restricted to the portion from the 14th position through the 14th

   1 2 3 5 6 7 8 10 12 13 15 16 18 <u>19</u> 20 22

5. Now the list has a single element and the loop ends. Since 19=19, the location 14 is returned

Sorting

- To ***sort*** the elements of a list is to put them in increasing or decreasing order (numerical order, alphabetic, and so on)
- Sorting is an important problem because:
  - A nontrivial percentage of all computing resources are devoted to sorting different kinds of lists
  - An amazing number of fundamentally different algorithms have been invented for sorting
    - Their relative advantages and disadvantages have been studied extensively
- A variety of sorting algorithms are proposed
  - binary, insertion, bubble, selection, merge, quick, and tournament
- We'll study some of them and later, we'll study the amount of time required to sort a list using the sorting algorithms covered in this section

Bubble Sort

- **Bubble sort** makes multiple passes through a list
- Every pair of elements that are found to be out of order are interchanged

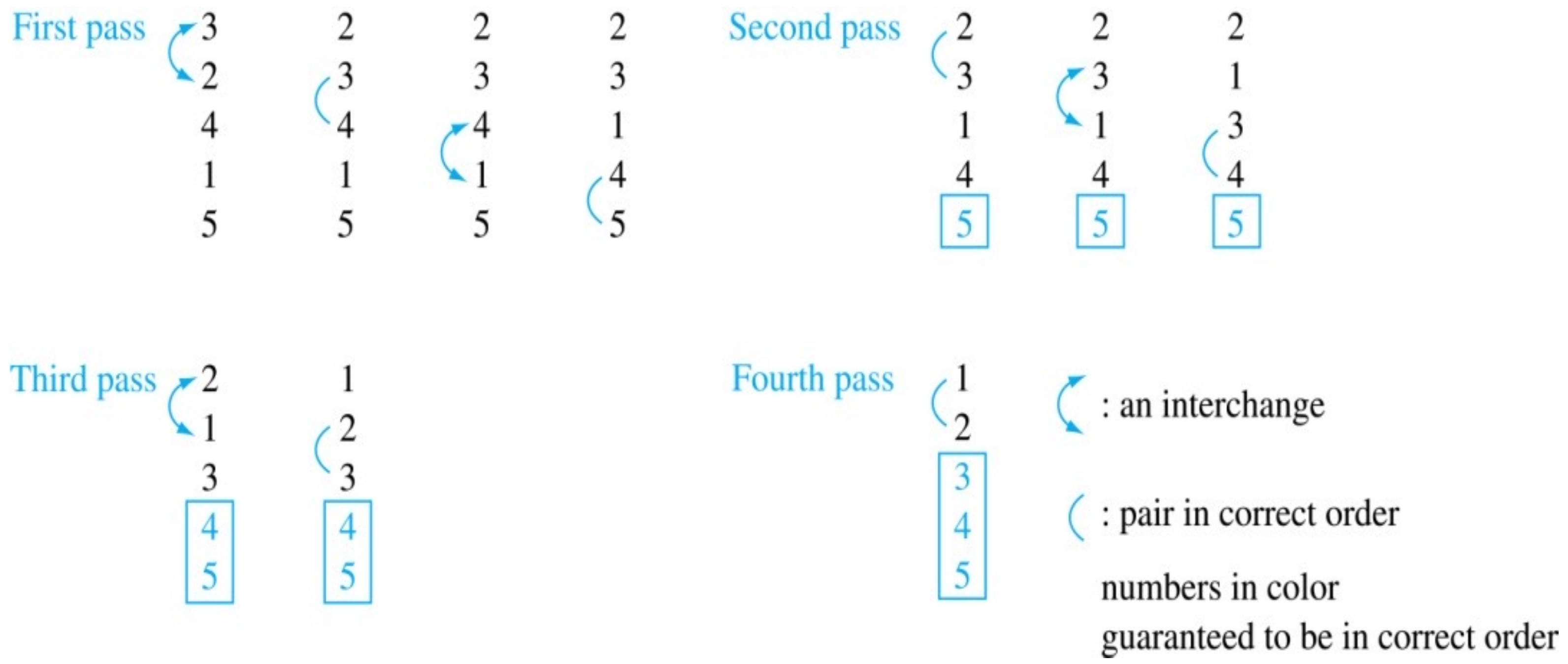
```
procedure bubbleSort($a_1,\ldots,a_n$: real numbers with $n \geq 2$)

  for $i$ := 1 to $n - 1$
    for $j$ := 1 to $n - i$
      if $a_j > a_{j+1}$ then interchange $a_j$ and $a_{j+1}$

{$a_1,\ldots, a_n$ is now in increasing order}
```

Bubble Sort

Example: Show the steps of bubble sort with 3 2 4 1 5



- At the first pass the largest element has been put into the correct position
- At the end of the second pass, the 2nd largest element has been put into the correct position
- In each subsequent pass, an additional element is put in the correct position

Insertion Sort

- Insertion sort begins with the 2nd element
- It compares the 2nd element with the 1st and puts it before the first if it is not larger
- Next the 3rd element is put into the correct position among the first 3 elements
- In each subsequent pass, the n+1st element is put into its correct position among the first n+1 elements
- Linear search is used to find the correct position

```
procedure insertion sort
  ($a_0,\ldots,a_{n-1}$: real numbers with $n \geq 2$)
    i := 1
    while i < n
        j := i
        while j > 0 and $a_{j-1} > a_j$
            swap $a_{j-1}$ and $a_j$
            j := j – 1
        i := i +1

{Now $a_0,\ldots,a_{n-1}$ is in increasing order}
{Note: This can easily be adapted for use
    with a 0-based array instead of a list of
    elements.}
```

Insertion Sort

Example: Show all the steps of insertion sort with the input: **3 2 4 1 5**

i. 2 3 4 1 5 (*first two positions are interchanged*)

ii. 2 3 4 1 5 (*third element remains in its position*)

iii. 1 2 3 4 5 (*fourth is placed at beginning*)

iv. 1 2 3 4 5 (*fifth element remains in its position*)

Problems

1. Describe an algorithm that takes as input a list of ***n*** integers and finds the number of negative integers in the list

A. Linear scan similar to MaxValue but add to sum for each negative encountered.

B. Sort then linear scan and count until encountering first positive.

C. Binary split?

Problems

2. Specify the steps of an algorithm that locates an element in a list of increasing integers by successively splitting the list into **four** sublists of equal (or as close to equal as possible) size, and restricting the search to the appropriate piece.

Modify binary search m = floor(i+j/4)

Problems

3. Use the bubble sort to sort 3, 1, 5, 7, 4, showing the lists obtained at each step.

n = 5 so four passes needed.

First Pass	Second Pass	Third Pass	Fourth Pass
$\underline{3\ 1}$ 5 7 4	$\underline{1\ 3}$ 5 4 7	$\underline{1\ 3}$ 4 5 7	$\underline{1\ 3}$ 4 5 7
1 $\underline{3\ 5}$ 7 4	1 $\underline{3\ 5}$ 4 7	1 $\underline{3\ 4}$ 5 7	1 3 4 5 7
1 3 $\underline{5\ 7}$ 4	1 3 $\underline{5\ 4}$ 7	1 3 4 5 7	
1 3 5 $\underline{7\ 4}$	1 3 4 5 7		
1 3 5 4 7			

Key Concepts in Computer Science (60-100)

Algorithms

Algorithms II (Optional)

School of Computer Science

Faculty of Science

University of Windsor

Dr. Scott Goodwin (Section 01)

Mr. Nabil Abdullah (Section 30)

Fall 2018

Algorithms II (Optional)

Algorithms II - Outline

- Greedy Algorithms

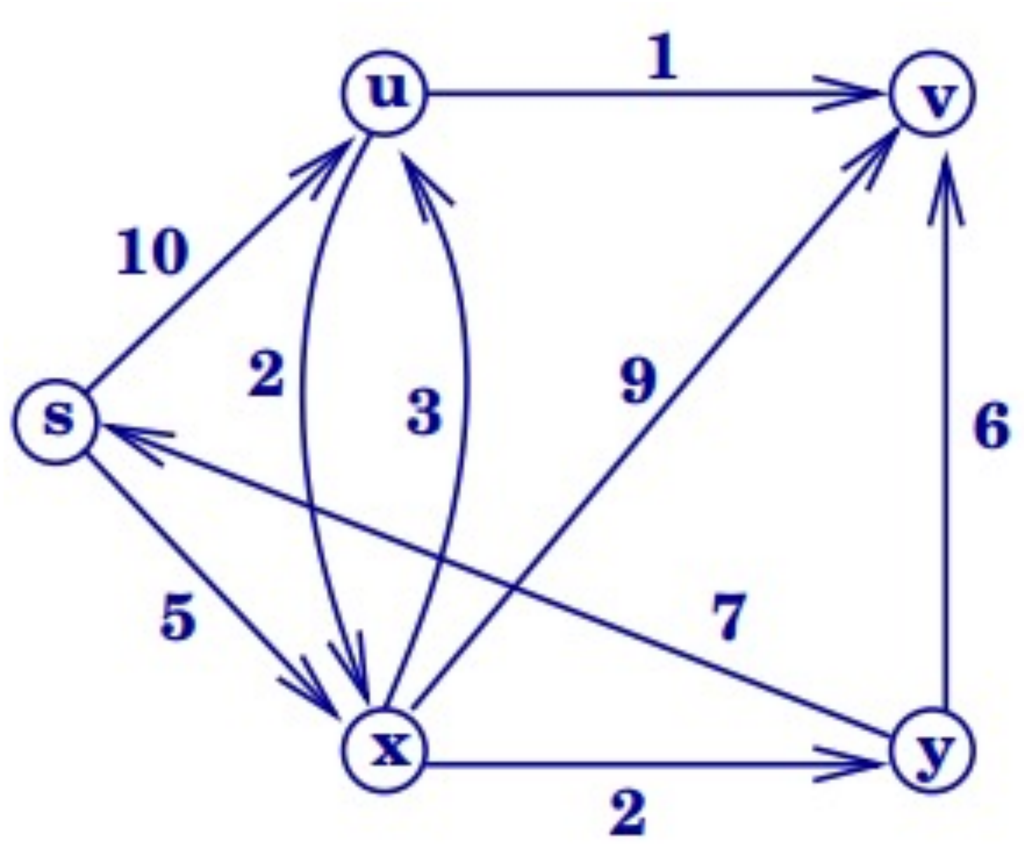
Greedy Algorithms

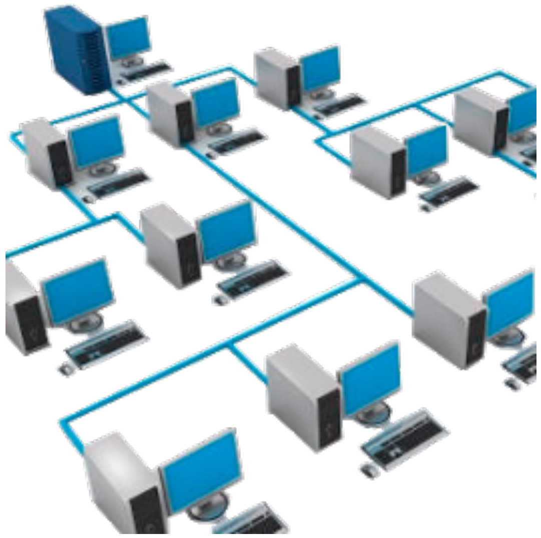
- ***Optimization problems*** minimize or maximize some parameter over all possible inputs
- **Example**:
  - Finding a route between two cities with the smallest total mileage
  - Determining how to encode messages using the fewest possible bits
  - Finding the fiber links between network nodes using the least amount of fiber

Greedy Algorithms

- 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
- The greedy approach to solving problems is an example of an algorithmic paradigm
- An algorithmic paradigm is an approach or method that underlies the design of a class of algorithms
- Examples of algorithmic paradigms are: greedy optimization, dynamic programming, branch and bound search, divide and conquer

Shortest Path





Find the shortest path from s to v.

Greedy Algorithms: Making Change

Example: Design a **greedy algorithm** for making change 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.
5. Choose another penny leaving 0 cents

Making Change

Example:

- Apply the previous idea to make change using quarters (25 cents), dimes (10 cents), nickels (5 cents), and pennies (1 cent) for:
  - 53 cents
  - 168 cents

Greedy Change-Making Algorithm

Solution:

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

```
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 Canadian currency, we may have quarters, dimes, nickels and pennies, with $c_1 = 25$, $c_2 = 10$, $c_3 = 5$, and $c_4 = 1^*$

* Note that pennies are no longer manufactured but are still accepted currency.

Proving Optimality for Canadian Coins

- Show that the change making algorithm for *Canadian* 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
- The allowable combinations, have a maximum value of 24 cents; 2 dimes and 4 pennies

Proving Optimality for Canadian Coins

Theorem: The greedy change-making algorithm for Canadian 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

Greedy Change-Making Algorithm

- Optimality depends on the **denominations** available
- For Canadian 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 31 cents
  - The optimal number of coins is 4, i.e., 3 dimes and 1 penny
  - What does the algorithm output?!

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:

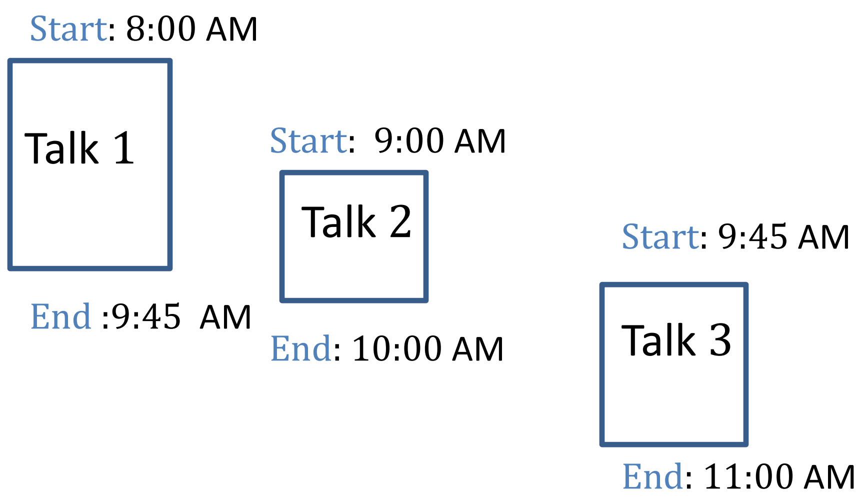
- When a talk starts, it continues till the end
- No two talks can occur at the same time
- A talk can begin at the same time that another ends
- Once we have selected some of the talks, we cannot add a talk which is incompatible with those already selected because it overlaps at least one of these previously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

Greedy Scheduling

- 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

- Picking the **shortest** talk doesn't work



- Can you find a counterexample here?
- But picking the one that **ends soonest** does work
- 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
        $S := S \cup \{$ talk $j \}$
return S [ S is the set of talks scheduled]
```

- Can be proved by induction

The Knapsack Problem (*optional*)

The famous **knapsack problem**:

- A thief breaks into a museum. Fabulous paintings, sculptures, and jewels are everywhere. The thief has a good eye for the value of these objects, and knows that each will fetch hundreds or thousands of dollars on the clandestine art collector's market. But, the thief has only brought a single knapsack to the scene of the robbery, and can take away only what he can carry. What items should the thief take to maximize the haul?

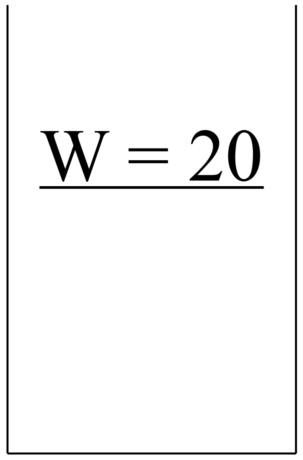
0-1 Knapsack Problem (*optional*)

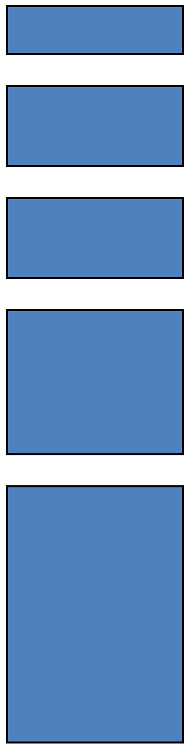
- Given a knapsack with maximum capacity $\boldsymbol{W}$, and a set $\boldsymbol{S}$ consisting of $\boldsymbol{n}$ items
- Each item $\boldsymbol{i}$ has some weight $\boldsymbol{w_i}$ and benefit value $\boldsymbol{b_i}$ (all $\boldsymbol{w_i}$, $\boldsymbol{b_i}$ and $\boldsymbol{W}$ are integer values)
- "0-1" because each item must be taken or left in entirety

- **Problem**: How to pack the knapsack to achieve maximum total value of packed items?

0-1 Knapsack Problem (*optional*)

This is a knapsack
Max weight: W = 20





Items	Weight w_i	Benefit Value b_i
1	2	3
2	3	4
3	4	5
4	5	8
5	9	10

Fractional Knapsack Problem (*optional*)

- Thief can take fractions of items
- Think of items in 0-1 problem as gold ingots, in fractional problem as buckets of gold dust

- **Problem**: How to pack the knapsack to achieve maximum total value of packed items?