CS1382 Discrete Computational Structures

Lecture 06: Algorithms

Spring 2019

Richard Matovu

TEXAS TECH UNIVERSITY.

References

- The materials of this presentation is mostly from the following:
    - Discrete Mathematics and Its Applications (Text book and Slides)

      By Kenneth Rosen, 7th edition

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***.

Algorithms

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 different ways. Their steps can be described in English or in ***pseudocode***.

- Pseudocode is an intermediate step between an English language description of the steps and a coding of these steps using a programming language.
- It uses some of the structures found in popular languages such as C++ and Java.
- Programmers can use the description of an algorithm in pseudocode to construct a program in a particular language.
- 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

- **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:

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?

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

The general ***searching problem*** is to locate an element x in the list of distinct elements $a_1, a_2, ..., a_n$ 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 on, we show that the binary search algorithm is much more efficient than linear search.

Binary Search

Here is a description of the binary search algorithm in pseudocode.

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}

Binary Search - Example

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

1 2 3 5 6 7 8 10 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 16 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 19 20 22

3. The midpoint of the current list is now the 14th position with a value of 19.
   Since 19 $\not>$ 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 18 19 20 22

Binary Search

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 19 20 22

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

Sorting Problems

- To ***sort*** the elements of a list is to put them in increasing 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, especially applications involving large databases of information that need to be presented in a particular order (e.g., by customer, part number etc.).
  - An amazing number of fundamentally different algorithms have been invented for sorting. Their relative advantages and disadvantages have been studied extensively.
  - Sorting algorithms are useful to illustrate the basic notions of computer science.
- A variety of sorting algorithms; binary, insertion, bubble, selection, merge, and quick.
- Later on, we'll study the amount of time required to sort a list using the sorting algorithms

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,…,a_n: real numbers with $n \geq 2$)

  for i := 1 to $n - 1$

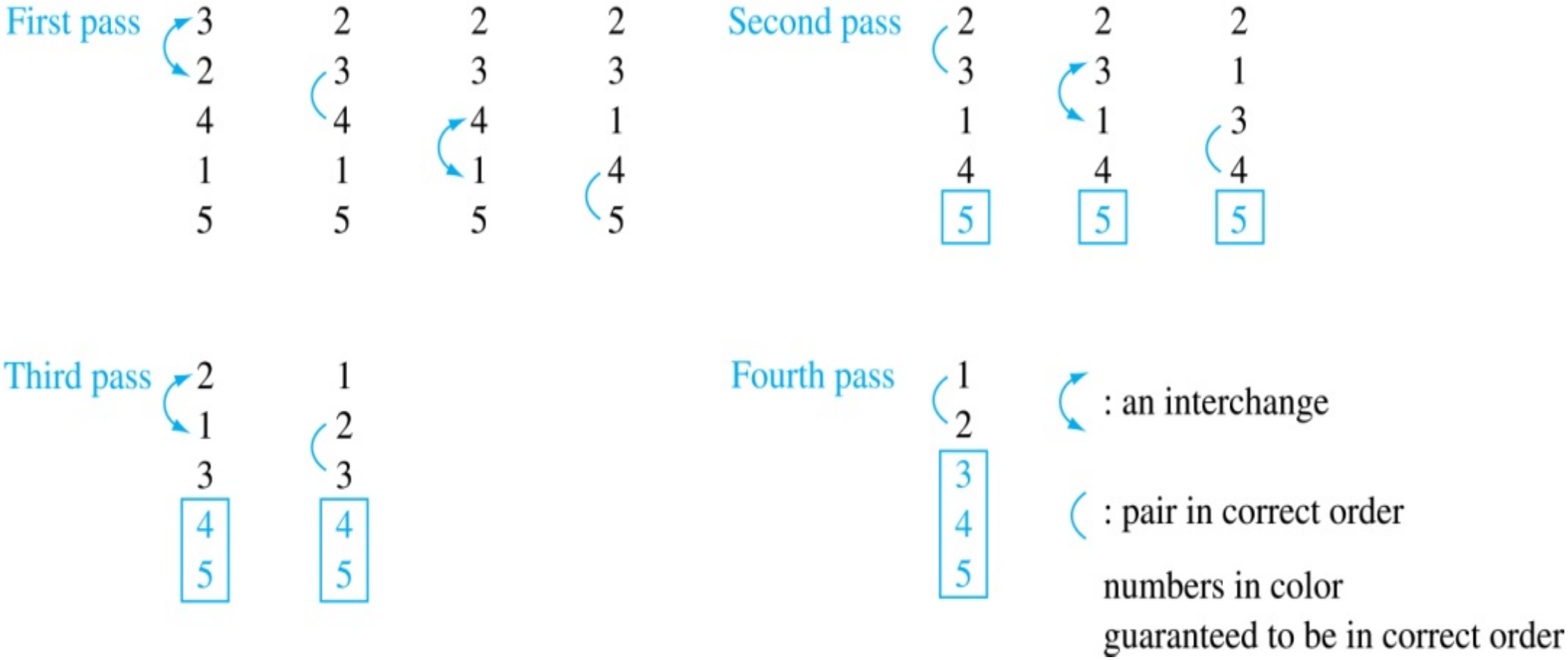
    for j := 1 to $n - i$

      if $a_j > a_{j+1}$ **then** interchange a_j and a_{j+1}

{a_1,…, a_n is now in increasing order}

Bubble Sort - Example

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



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 *insertionsort* ($a_1,\ldots,a_n$: real numbers with $n \geq 2$)

```
for j := 2 to n
    i := 1
    while a_j > a_i
        i := i + 1
    m := a_j
    for k := 0 to j − i
        a_{j-k} := a_{j-k-1}
    a_i := m
```

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

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*)

Optimization Problems

- ***Optimization problems*** minimize or maximize some parameter over all possible inputs.
- Examples of optimization problems:
  - 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.
- The greedy approach to solving problems is an example of an algorithmic paradigm, which is a general approach for designing an algorithm.

Greedy Algorithms: Making Change

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

Greedy Change-Making Algorithm

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, …, 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 $c_4 = 1$.

Greedy Change-Making Algorithm

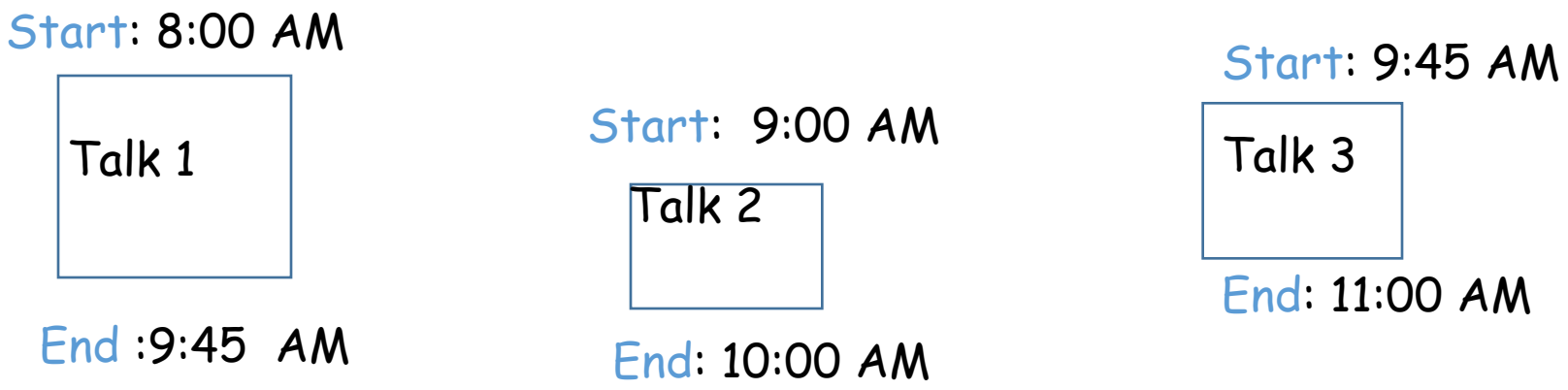
- Optimality depends on the denominations available.
- For U.S. coins, optimality still holds if we add half dollar coins (50 cents) and dollar coins (100 cents).
- But if we allow only quarters (25 cents), dimes (10 cents), and pennies (1 cent), the algorithm no longer produces the minimum number of coins.
    - Consider the example of 31 cents.
        - What does the algorithm output?
        - The optimal number of coins is 4, i.e., 3 dimes and 1 penny.

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.
  - 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 page.

Greedy Scheduling algorithm

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

procedure *schedule*($s_1 \le s_2 \le \ldots \le s_n$: start times, $e_1 \le e_2 \le \ldots \le e_n$: end times)

sort talks by finish time and reorder so that $e_1 \le e_2 \le \ldots \le e_n$

$S := \emptyset$

for $j := 1$ to n

  if talk j is compatible with S then

    S := S∪{talk j}

return S [S is the set of talks scheduled]

- Can be proven correct by induction in Chapter 5.

Questions?

Thank You!