Discrete Structures Spring 2024 – Week 5

Algorithms

Lecture 1

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

Abu Ja'far Mohammed Ibin Musa Al-Khowarizmi
(780-850)

- *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.

Algorithms

- 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.
3. If it is larger than the temporary maximum, set the temporary maximum equal to this integer.
4. Repeat the previous step if there are more integers. If not, stop.

- 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.
- 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?

Exercise

- Determine which characteristics of an algorithm the following procedures have and which they lack.

- **procedure** *double*(n: positive integer)
- **while** $n > 0$
- $n := 2n$

Some Example Algorithm Problems

- Classes of problems to 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.

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.

1. 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.

1. Linear Search Algorithm

procedure *linear search*(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}

2. Binary Search Algorithm

- 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.

2. Binary Search Algorithm

procedure binary search(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 Algorithm

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 8^{th} 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 12^{th} position with a value of 16. Since 19 > 16, further search is restricted to the 13^{th} position and above.

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

Binary Search Algorithm

3. The midpoint of the current list is now the 14th position with a value of 19. Since 19 $\ngtr$ 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

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 Algorithms

Lecture 2

Sorting

- 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:
- Lot of computing resources are spent 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.).

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}

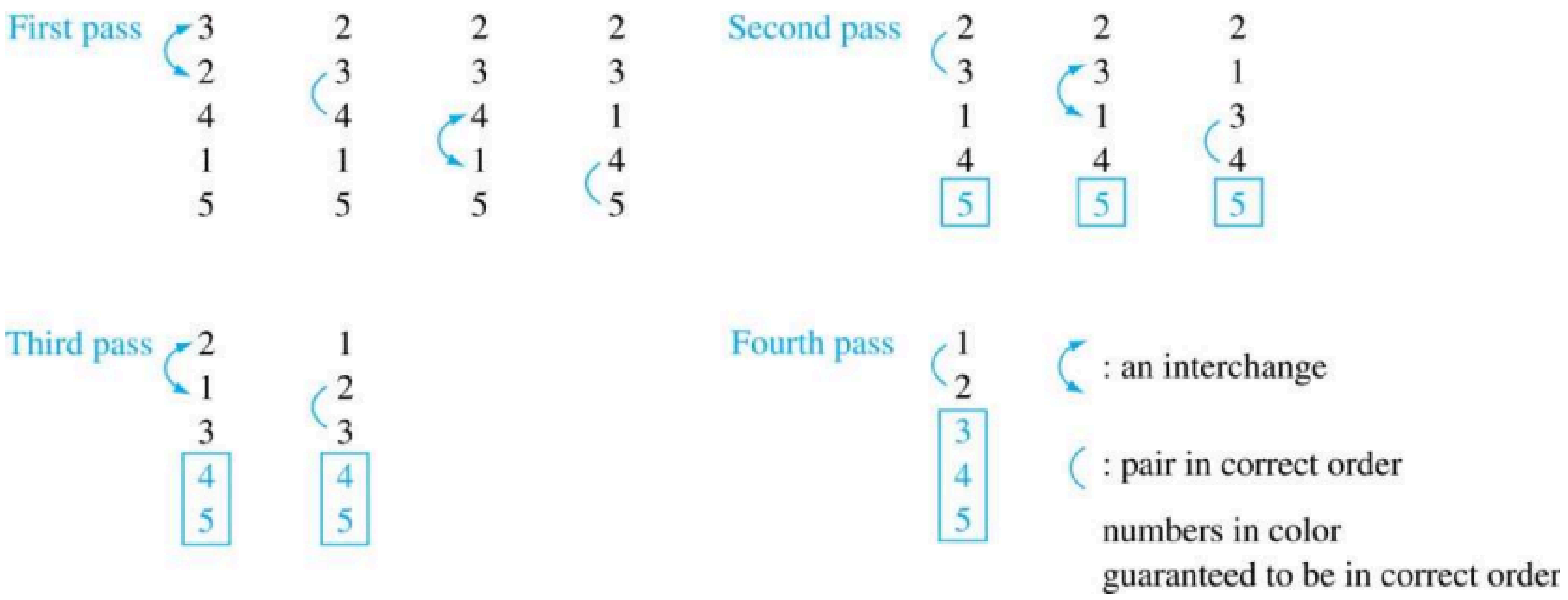
Bubble Sort

- 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.

Bubble Sort



Bubble Sort – Illustration

https://en.wikipedia.org/wiki/Bubble_sort#/media/File:Bubble-sort-example-300px.gif

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 $n+1$st element is put into its correct position among the first $n+1$ elements.
- Linear search is used to find the correct position.

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)*

Insertion Sort – Illustration

https://en.wikipedia.org/wiki/Insertion_sort#/media/File:Insertion-sort-example-300px.gif

Insertion Sort

procedure *insertion sort*
($a_1,\ldots,a_n$:
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 $j - i - 1$
$a_{j-k} := a_{j-k-1}$
$a_i := m$
{Now $a_1,\ldots,a_n$ is in increasing order}

Complexity of Algorithms

Lecture 3

The Complexity of Algorithms

- Given an algorithm, how efficient is this algorithm for solving a problem given input of a particular size? To answer this question, we ask:

1. How much time does this algorithm use to solve a problem?
2. How much computer memory does this algorithm use to solve a

problem?

Time and Space Complexity

- When we analyze the time the algorithm uses to solve the problem given input of a particular size, we are studying the *time complexity* of the algorithm.
- When we analyze the computer memory the algorithm uses to solve the

problem given input of a particular size, we are studying the *space complexity* of the algorithm.

Time Complexity

- The focus would be on time complexity only
- Time complexity is measured in terms of the number of operations an algorithm uses and big-O and big-Theta notation is used to estimate the time complexity.

- This analysis can be used to see whether it is practical to use this algorithm to solve problems with input of a particular size and to compare the efficiency of different algorithms for solving the same problem

Time Complexity

- To analyze the time complexity of algorithms, we determine the number of operations, such as comparisons and arithmetic operations (addition, multiplication, etc.)

- We can estimate the time a computer may actually use to solve a problem using the amount of time required to do basic operations.
- We will focus on the *worst-case time* complexity of an algorithm. This provides an upper bound on the number of operations an algorithm uses to solve a problem with input of a particular size.

Time Complexity – Finding Maximum Element in a Finite Sequence

- **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}

Time Complexity – Finding Maximum Element in a Finite Sequence

- **Solution:** Count the number of comparisons.
- The max < a_i comparison is made $n - 1$ times.
- Each time i is incremented, a test is made to see if $i \leq n$.
- One last comparison determines that $i > n$.
- Exactly $2(n - 1) + 1 = 2n - 1$ comparisons are made.
- Hence, the time complexity of the algorithm is $\Theta(n)$.

Time Complexity –

Linear Search

- **procedure** *linear search*(x:integer, a_1, a_2, …,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}

Time Complexity – Linear Search

- **Solution:** Count the number of comparisons.
  - At each step two comparisons are made; $i \leq n$ and $x \neq ai$.
  - To end the loop, one comparison $i \leq n$ is made.
  - After the loop, one more $i \leq n$ comparison is made.
- If $x = a_i$, $2i + 1$ comparisons are used. If x is not on the list, $2n + 1$ comparisons are made and then an additional comparison is used to exit the loop. So, in the worst case $2n + 2$ comparisons are made. Hence, the complexity is $\Theta(n)$.

Time Complexity – Binary Search

- **procedure** binary search(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}

Time Complexity – Binary Search

• **Solution:** Assume (for simplicity) $n = 2^k$ elements. Note that k = log n. • Two comparisons are made at each stage; i < j, and x > a_m • At the first iteration the size of the list is 2k and after the first iteration it is 2k-1 then 2k-2 and so on until the size of the list is $2^1 = 2$

- At the last step, a comparison tells us that the size of the list is the size is

$2^0 = 1$ and the element is compared with the single remaining element.

• Hence, at most 2k + 2 = 2 log n + 2 comparisons are made. •

Therefore, the time complexity is Θ (log n), better than linear search.

Time Complexity – Bubble Sort

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$\}$

Time Complexity – Bubble Sort

- **Solution**: A sequence of $n-1$ passes is made through the list. On each pass $n - i$ comparisons are made.

- The worst-case complexity of bubble sort is

$$(n-1)+(n-2)+\ldots+2+1 \;=\; \tfrac{n(n-1)}{2}\Theta(n^2) \text{ since}$$

$$\frac{n(n-1)}{2} \;=\; \frac{1}{2}n^2 - \frac{1}{2}n$$

Time Complexity – Insertion Sort

- **Solution**: The total number of comparisons are:

$$2+3+\cdots+n \;=\; \frac{n(n-1)}{2} - 1$$

• Therefore the complexity is $\Theta(n^2)$

procedure *insertion sort*($a_1,\ldots,a_n$: 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 $j - i - 1$

$a_{j-k} := a_{j-k-1}$

$a_i := m$

Comparison of Time Complexity