Algorithm Examples

Section Summary

- ✓ Properties of Algorithms
- ✓ Algorithms for Searching and Sorting
- ✓ Greedy Algorithms

Problems and Algorithms

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

steps of a procedure that takes a valid input and produces the desired output

This procedure is called an ***algorithm***.

Definition

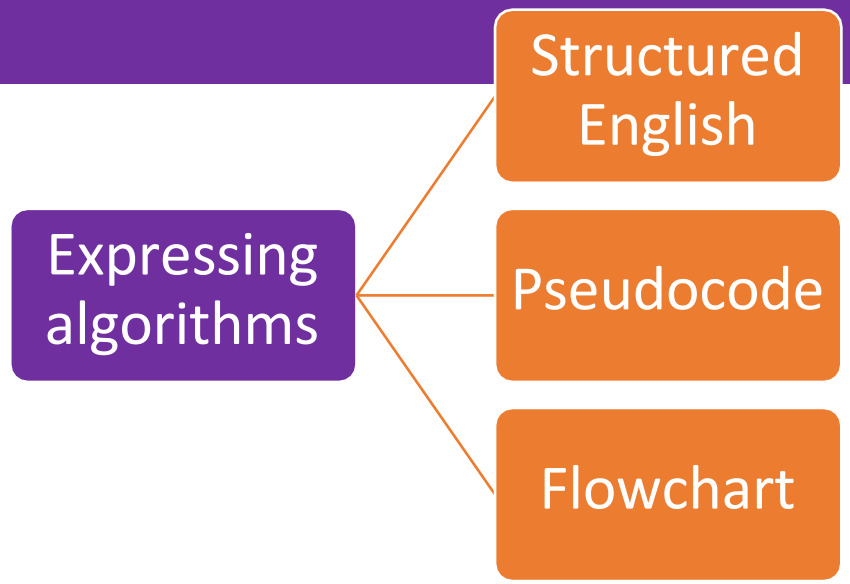
An *algorithm* is

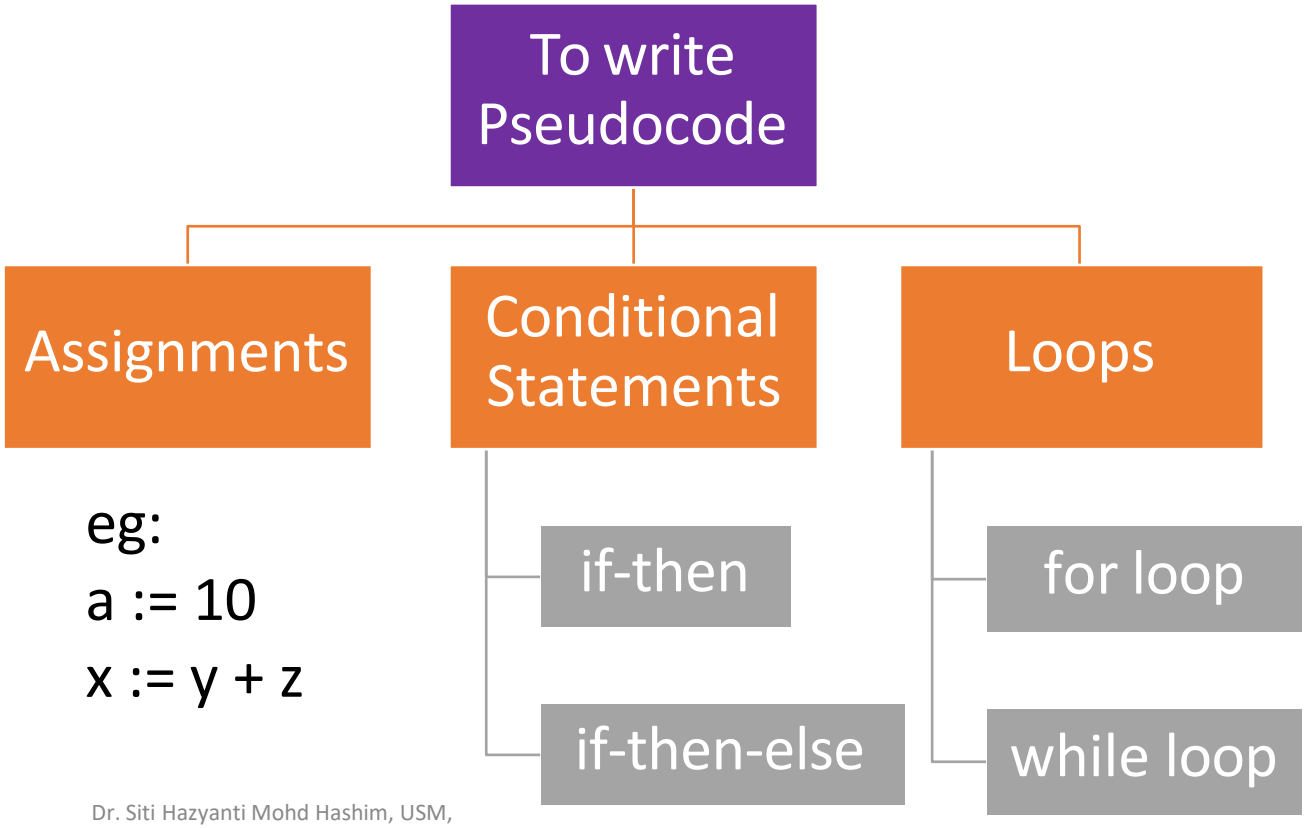
a **finite set of precise instructions**

for performing a **computation** or

for **solving** a **problem**.

Algorithms





eg:

a := 10

x := y + z

Pseudocode

Assignment of variables

Example:

```
a := 10
i := i+ 1
```

if...then statement

Example:

```
if  a = 1 then  count := count + 1
```

if...then...else statement

Example:

```
if  a > 0 then  pc := pc + 1
else nc := nc + 1
```

While loop

Example:

```
i := 1
while i ≤ n
      i := i + 1
```

for loop

Example:

```
for  i := 1 to n
      sum := sum + i
```

procedure

Example:

procedure *max* (*a1, a2,…, an: integers*)
Body the procedure
return *max*

Example

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.

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

Does this algorithm have all the properties listed on the previous slide?

Steps

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

List all the steps used to find the maximum value in the following list:
{ 10, 6, 8, 20, 2}

(a_1:=10, a_2:=6, a_3:=8, a_4:=20, a_5:=2)				
max := 10				
i:= 2 *true* if 10 < 6 *false* *(no change)* i := 2 + 1 := 3	i:= 3 *true* if 10 < 8 *false* *(no change)* i := 3 + 1 := 4	i:= 4 *true* if 10 < 20 *true* max := 20 i := 4 + 1 := 5	i:= 5 *true* if 20 < 2 *false* *(no change)* i := 5 + 1 := 6	i:= 6 *false*
return 20				

Algorithms – Q & A

Student Work

1. Find the maximum element in a finite sequence.
    a. for loop
    b. Using while loop

a. for loop

```
procedure  max (a1, a2,..., an: integers )
max := a1
for i := 2 to n
    if max < ai then max := ai
return max{max is the largest element}
```

b. Using while loop

```
procedure  max (a1, a2,..., an: integers )
max := a1
i := 2
while  i ≤ n
    if max < ai then max := ai
     i := i  + 1
return max{max is the largest element}
```

Algorithms – Q & A

Student Work

2. Find the minimum element in a finite sequence.
    a. for loop
    b. Using while loop

a. for loop

```
procedure  min ( a1, a2,..., an: integers )
min := a1
for i := 2 to n
    if min > ai then min := ai
return min{min is the smallest element}
```

b. Using while loop

```
procedure  min ( a1, a2,..., an: integers )
min := a1
i := 2
while  i ≤ n
    if min > ai then min := ai
     i := i  + 1
return min{min is the smallest element}
```

Algorithms – Q & A

Student Work

3. Devise an algorithm that finds the sum of all the integers in a list.

procedure $AddUp(a_1, \ldots, a_n$: integers)
$sum := a_1$
for $i := 2$ **to** n
  $sum := sum + a_i$
return sum

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.

Show that Algorithm for finding the maximum element in a finite sequence of integers has all the properties listed.

Student Work

- *Input:*
- *Output:*
- *Definiteness:*
- *Correctness:*
- *Finiteness:*
- *Effectiveness:*
- *Generality:*

Show that Algorithm for finding the maximum element in a finite sequence of integers has all the properties listed.

Student Work

- ***Input:*** a sequence of integers
- ***Output:*** the largest integer in the sequence
- ***Definiteness:*** Each step of the algorithm is precisely defined, because only assignments, a finite loop, and conditional statements occur.
- ***Correctness:*** The algorithm is correct, because when the algorithm terminates, the value of the variable max equals the maximum of the terms of the sequence.
- ***Finiteness:*** The algorithm uses a finite number of steps, because it terminates after all the integers in the sequence have been examined.
- ***Effectiveness:*** The algorithm can be carried out in a finite amount of time because each step is either a comparison or an assignment, there are a finite number of these steps, and each of these two operations takes a finite amount of time.
- ***Generality:*** Algorithm 1 is general, because it can be used to find the maximum of any finite sequence of integers.

Properties of Algorithms – 0 &

Student Work

Determine which characteristics of an algorithm described in the text the following procedures have and which they lack.

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

b)
procedure *divide*(*n*: positive integer)
while $n \geq 0$
  $m := 1/n$
  $n := n - 1$

c)
procedure *sum*(*n*: positive integer)
sum := 0
while $i < 10$
  sum := *sum* + *i*

d)
procedure *choose*(*a*, *b*: integers)
  x := either *a* or *b*

a) This procedure is not finite, since execution of the while loop continues forever.

b) This procedure is not effective, because the step m := 1/n cannot be performed when n = 0, which will eventually be the case.

c) This procedure lacks definiteness, since the value of i is never set.

d) This procedure lacks definiteness, since the statement does not tell whether x is to be set equal to a or to b.

Example of Algorithm Problem

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.

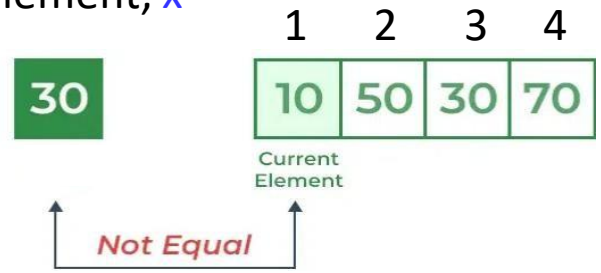
Searching Algorithms

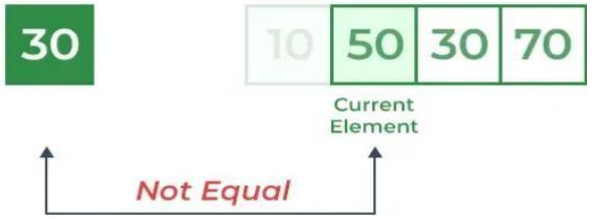
The general searching problem can be described as follows:

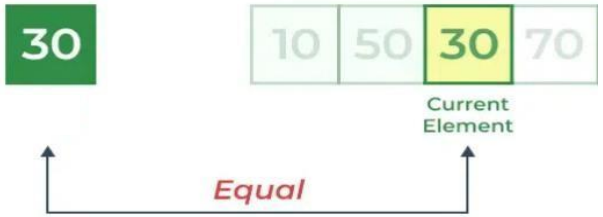
- Locate an element x in a list of distinct elements $a_1, a_2, \ldots, a_n$, or determine that it is not in the list.
- The solution to this search problem
  - is the location of the term in the list that equals x (that is, i is the solution if $x = a_i$) and
  - is 0 if x is not in the list.

Linear Search Algorithm - Example

Searching element, x

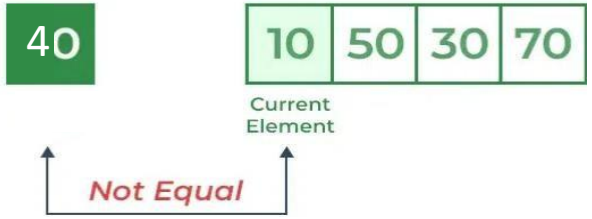






Location = 3

Searching element,
x = 40



Location = 0

Searching Problem

Definition: The general *searching problem* is to locate an element x in the list of distinct elements $a_1, a_2, \ldots, an$, **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 $x = ai$) **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.

Our focus → linear search & 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 *linear search*(x: $integer, a_1, a_2, \ldots, an$: distinct integers)

$i := 1$

while ($i \leq n$ and $x \neq ai$)

$i := i + 1$

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 Algorithm

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

procedure binary search(x: integer, $a_1, a_2, \ldots, a_n$: increasing integers)

$i := 1$ $\{i \text{ is the left endpoint of interval}\}$

$j := n$ $\{j \text{ is right endpoint of interval}\}$

while $i < j$

$m := \lfloor (i + j)/2 \rfloor$

if $x > am$ then $i := m + 1$

else $j := m$

if $x = ai$ **then** $location := i$

else $location := 0$

return $location\{location \text{ is the subscript } i \text{ of the term } ai \text{ equal to } x, \text{or } 0 \text{ if } x \text{ is not found}\}$

Binary Search Algorithm

Assume the input is a list of items in increasing order. **Example**:

To search for **19** in the list: 1 2 3 5 6 7 8 10 12 13 15 16 18 19 20 22

1. Split into two smaller lists

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

2. Compare 19 and the largest term in the first list

10 < 19

Move searching to the 'second list'

3. Split again into two smaller lists (NEW)

12 13 15 16 18 19 20 22

4. Compare 19 and the largest term in the first list

16 < 19

Move searching to the 'second list'

5. Split again into two smaller lists (NEW)

18 19 20 22

6. Compare 19 and the largest term in the first list

19 = 19

Search restricted to **this** list

7. Split again into two smaller lists (NEW)

18 19

8. Compare: 18 < 19, then search restricted to the second list (i.e. the 14th term in the original list)

UNIVERSITI SAINS MALAYSIA

www

List all the steps used to search for 19 in the list

procedure *binary search* (x: integer, $a_1, a_2, \ldots, a_n$: increasing integers)
$i := 1$ {i is left endpoint of search interval}
$j := n$ {j is right endpoint of search 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}

Student Work

(x := 19, a_1:=1, a_2:=2, a_3:=3, a_4:=5, a_5:=6 a_6:=7, a_7:=8, a_8:=10, a_9:=12, a_{10}:=13, a_{11}:=15, a_{12}:=16,a_{13}:=18, a_{14}:=19, a_{15}:=20, a_{16}:=22)

i := 1
j := 16

while 1 < 16 *True* *(split the list into two smaller lists)* m := (1+16)/2 := 8 if 19 > 10 *True* *(select second half and change left endpoint)* i := 8+1 := 9	while 9 < 16 *True* *(split the list into two smaller lists)* m := (9+16)/2 := 12 if 19 > 16 *True* *(select second half and change left endpoint)* i := 12+1 :=13	while 13 < 16 *True* *split the list into two smaller lists* m := (13+16)/2 := 14 if 19 > 19 *false* *(select first half and change right endpoint)* j :=14	while 13 < 14 *True* *split the list into two smaller lists* m := (13+14)/2 := 13 if 19 > 18 *true* *(select second half and change left endpoint)* i := 13 + 1 := 14	while 14 < 14 *false*

if 19 := 19 *True*
  location := 14

Sorting Algorithm

To ***sort*** the elements of a list is to put them in **increasing order** (numerical order, alphabetic, and so on).

A variety of sorting algorithms; binary, **insertion**, **bubble**, selection, merge, quick, and tournament.

Sorting Algorithm: 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 $bubblesort(a_1, \ldots, an: \ real\ numbers\ with\ n \geq 2)$

for $i := 1$ to $n - 1$

    for $j := 1$ to $n - i$

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

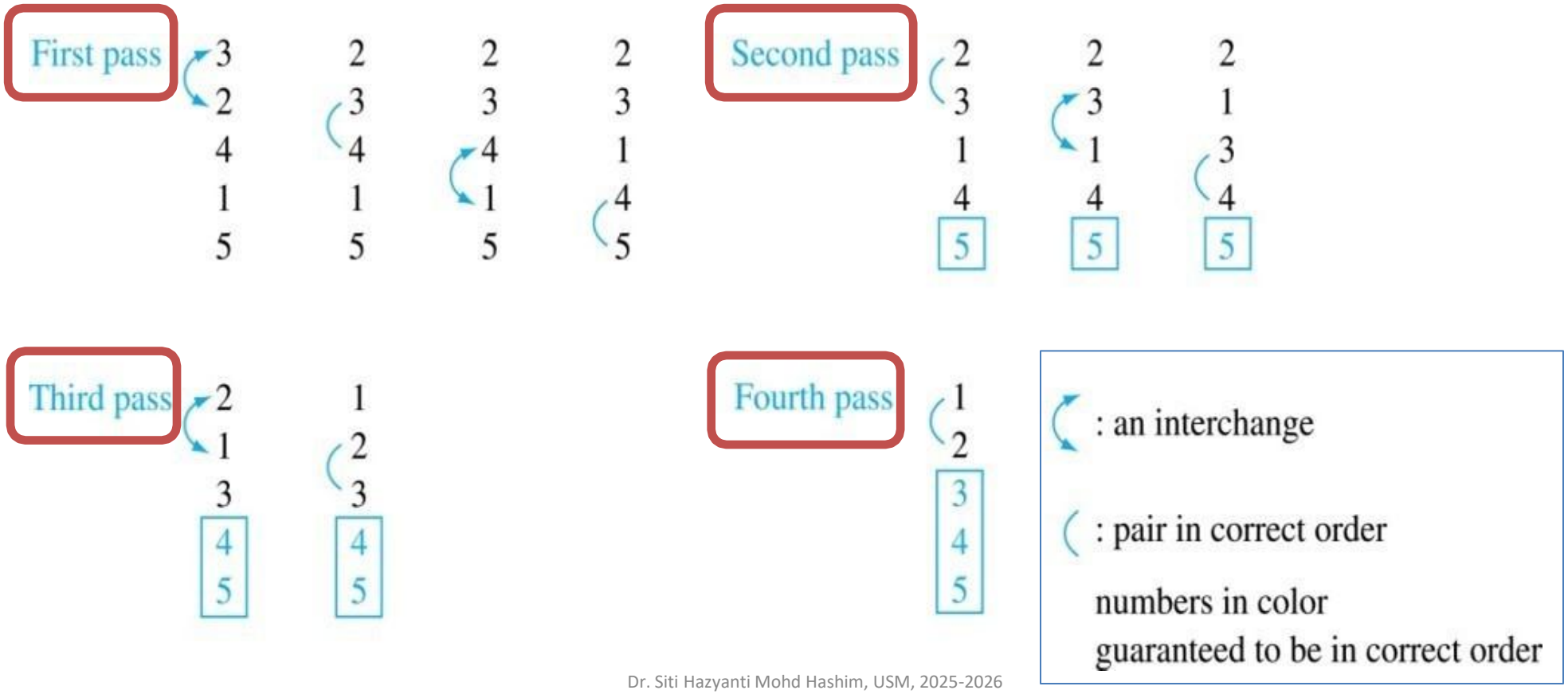
$\{a_1, \ldots, an\ is\ now\ in\ increasing\ order\}$

Sorting Algorithm: Bubble Sort

Example: Use the bubble sort to put 3 2 4 1 5 into increasing order.

Sorting Algorithm: Bubble Sort

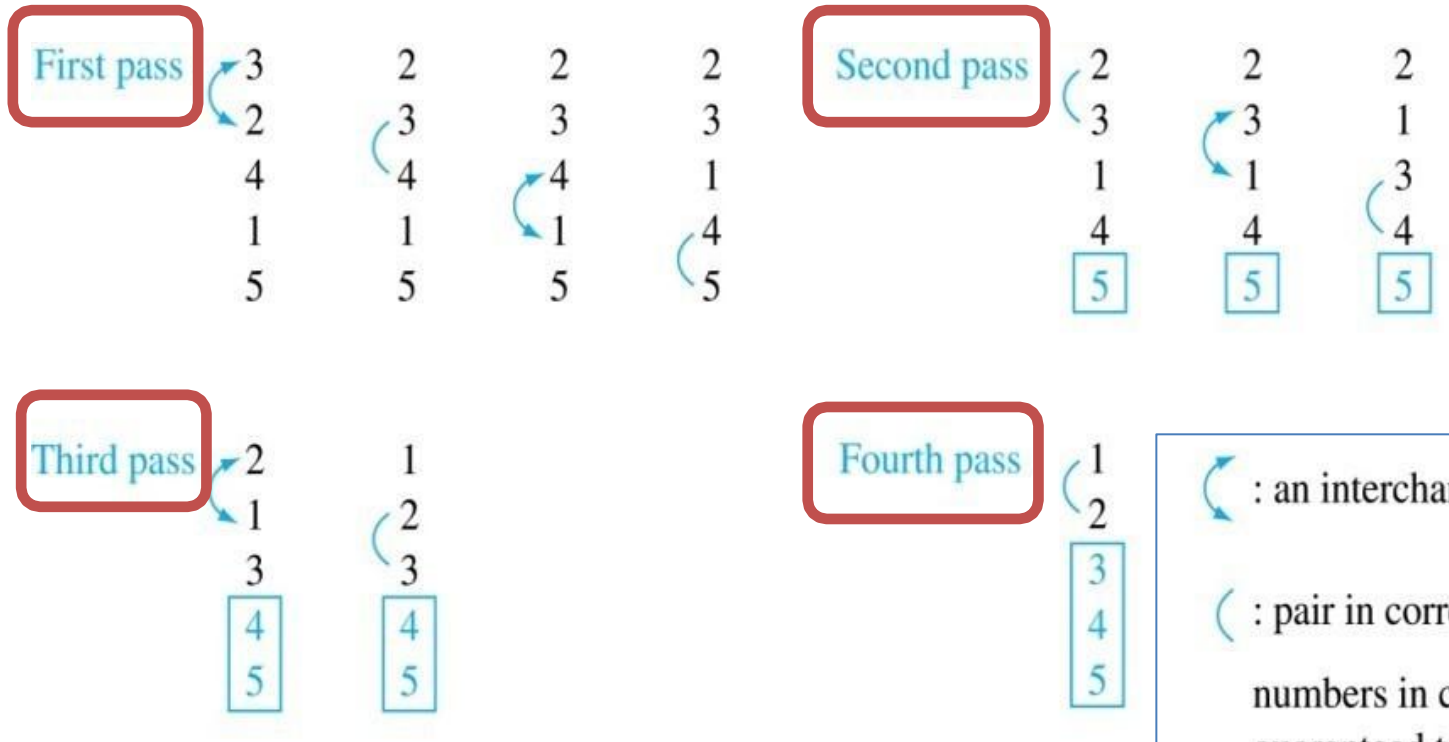
Example: Use the bubble sort to put 3 2 4 1 5 into increasing order.



Sorting Algorithm: Bubble Sort

Example: Use the bubble sort to put 3 2 4 1 5 into increasing order.

3 2 4 1 5
[1] [2] [3] [4] [5]



procedure
$bubblesort(a_1, \ldots, an$: $real\ numbers\ with\ n \geq 2)$
for $i := 1$ to $n - 1$
  for $j := 1$ to $n - i$
    if $a_j > aj_{+1}$ **then**
    interchange a_j and a_{j+1}

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

Sorting Algorithm: 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*

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

  for $j := 2$ **to** n

    $i := 1$

    while $a_j > ai$

      $i := i + 1$

    $m := aj$

    for $k := 0$ **to** $j - i - 1$

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

      $a_i := m$

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

Sorting Algorithm: Insertion Sort

Example: Use the insertion sort to put 3 2 4 1 5 into increasing order.

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*)

Sorting Algorithm: Insertion Sort

Example: Use the insertion sort to put 3 2 4 1 5 into increasing order.

3 [1] 2 [2] 4 [3] 1 [4] 5 [5]

procedure *insertion sort*

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

  for $j := 2$ **to** n

    $i := 1$

    while $a_j > ai$

      $i := i + 1$

    $m := aj$

    for $k := 0$ **to** $j - i - 1$

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

      $a_i := m$

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

The steps of insertion sort

The insertion sort first compares 2 and 3.

Because 3 > 2, it places 2 in the first position, producing the list 2, 3, 4, 1, 5 (the sorted part of the list is shown in color). At this point, 2 and 3 are in the correct order.

Next, it inserts the third element, 4, into the already sorted part of the list by making the comparisons 4 > 2 and 4 > 3. Because 4 > 3, 4 remains in the third position. At this point, the list is 2, 3, 4, 1, 5 and we know that the ordering of the first three elements is correct.

Next, we find the correct place for the fourth element, 1, among the already sorted elements, 2, 3, 4. Because 1 < 2, we obtain the list 1, 2, 3, 4, 5.

Finally, we insert 5 into the correct position by successively comparing it to 1, 2, 3, and 4. Because 5 > 4, it stays at the end of the list, producing the correct order for the entire list.

Week 3: Class Activity

List all the steps used to search for **9** in the sequence

1, 3, 4, 5, 6, 8, 9, 11

Using: a) a **linear** search

b) a **binary** search

Optimization Problem

- *Optimization problems* **minimize** or **maximize** some parameter over all possible inputs.
- Among the many optimization problems that we will study are:
  - 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

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 Algorithms: Making Change

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 $c_4 = 1$.

Greedy Algorithms: Making Change

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]

U.S. currency; quarters, dimes, nickels pennies
$c_1 = 25$, $c_2 = 10$, $c_3 = 5$, and $c_4 = 1$.

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.