Chapter 3
Algorithms

Section 3.1

Algorithms

- **Definition: An algorithm is a finite set of precise instructions for performing a computation or for solving a problem**

- **Example 1.** Describe an algorithm for finding the maximum (largest) value in a finite sequence of integers. (what is a sequence?)

- **Solution:**
  - Set the temporary max equal to the first integer in the sequence.
  - Compare the next integer in the sequence and if it is larger, set the temporary maximum equal to this new value.
  - Repeat previous step if more integers to test.
  - Stop when no more integers. At this point the temporary max becomes the max.

Pseudocode

- If a particular computer language were chosen, then the resulting algorithm would be expressed in that language only.
- So instead, a form of **pseudo-code** is used. It is a cross between basic computer language operations, mathematical notation and English.

Algorithm 1 in pseudocode

```
procedure find-max ( $a_1, a_2, \ldots, a_n$ : integers)
    $max := a_1$
    for $i := 2$ to $n$
        if $max < a_i$ then $max := a_i$
```

Linear Search Algorithm

note: algorithms usually have 3 things

- assignment statements i := 1
- looping statements while … or for …
- conditional statements if … then …else …

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

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 search interval )
j: = n ( j is the right endpoint of the search interval )
    while i < j
    begin
        $m := \lfloor (i+j)/2 \rfloor$
        if  $x > a_m$ then
                i : = m + 1
        else
                j: = m
    end
    if  $x = a_i$  then  location := i
    else location : = 0
{location is the subscript of the term that equals x or is 0 if x is not found)
```

```
2 6 7 34 76 123 234 567 677 986
i         m                  j

2 6 7 34 76 123 234 567 677 986
             i       m       j

2 6 7 34 76 123 234 567 677 986
             i   m   j

2 6 7 34 76 123 234 567 677 986
             im   j

2 6 7 34 76 123 234 567 677 986
            imj

2 6 7 34 76 123 234 567 677 986

2 6 7 34 76 123 234 567 677 986
```

Search for 123

i = first
j = last
m = middle point

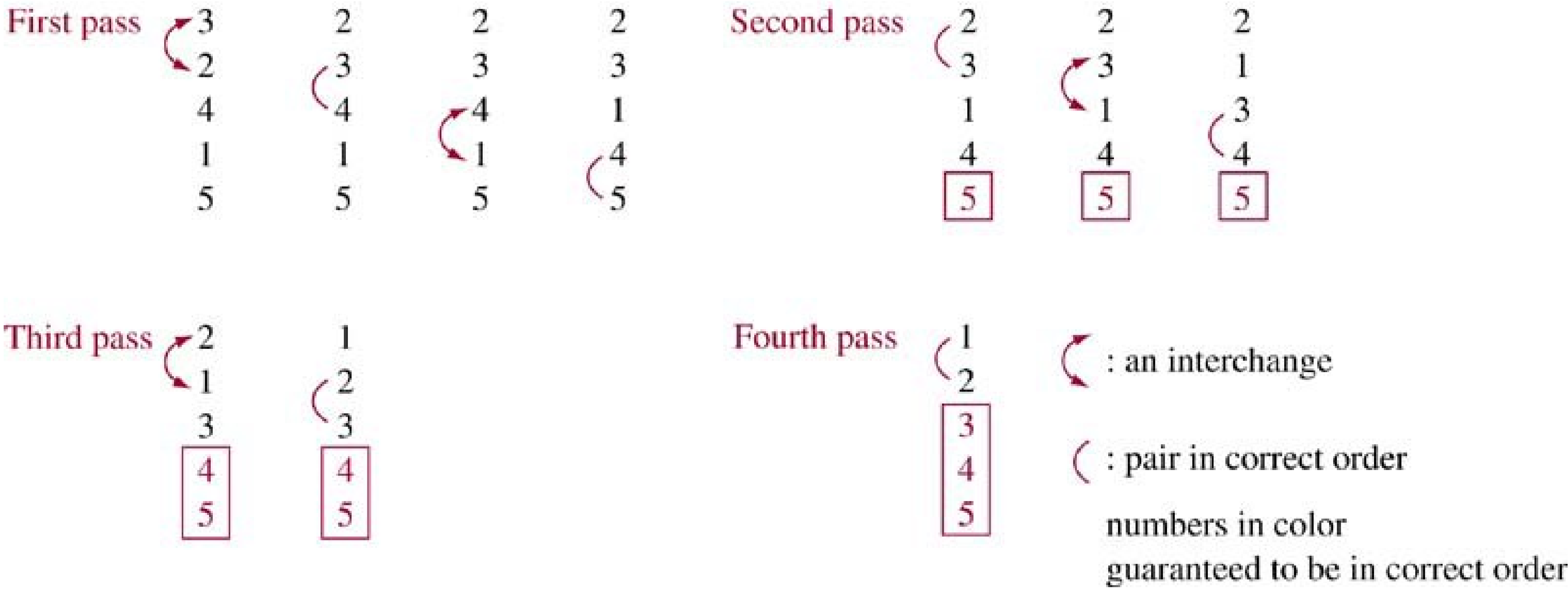
End when i > j

```
i := 1
j := n

while $i < j$
        begin
            $m := \lfloor (i+j)/2 \rfloor$
            if $x > a_m$ then
                $i := m + 1$
            else
                $j := m$
        end
if $x = a_i$ then $location := i$
else $location := 0$
```

Sorting

- Sorting rearranges a sequence of numbers so that the numbers are in increasing order (from smallest to largest)
- Sorting data is one of the most common operations performed on data.
- We begin with **bubblesort**, where at each pass, the largest ("heaviest") element is moved to the bottom of the sequence.


First pass
Second pass
Third pass
Fourth pass
: an interchange
: pair in correct order
numbers in color
guaranteed to be in correct order

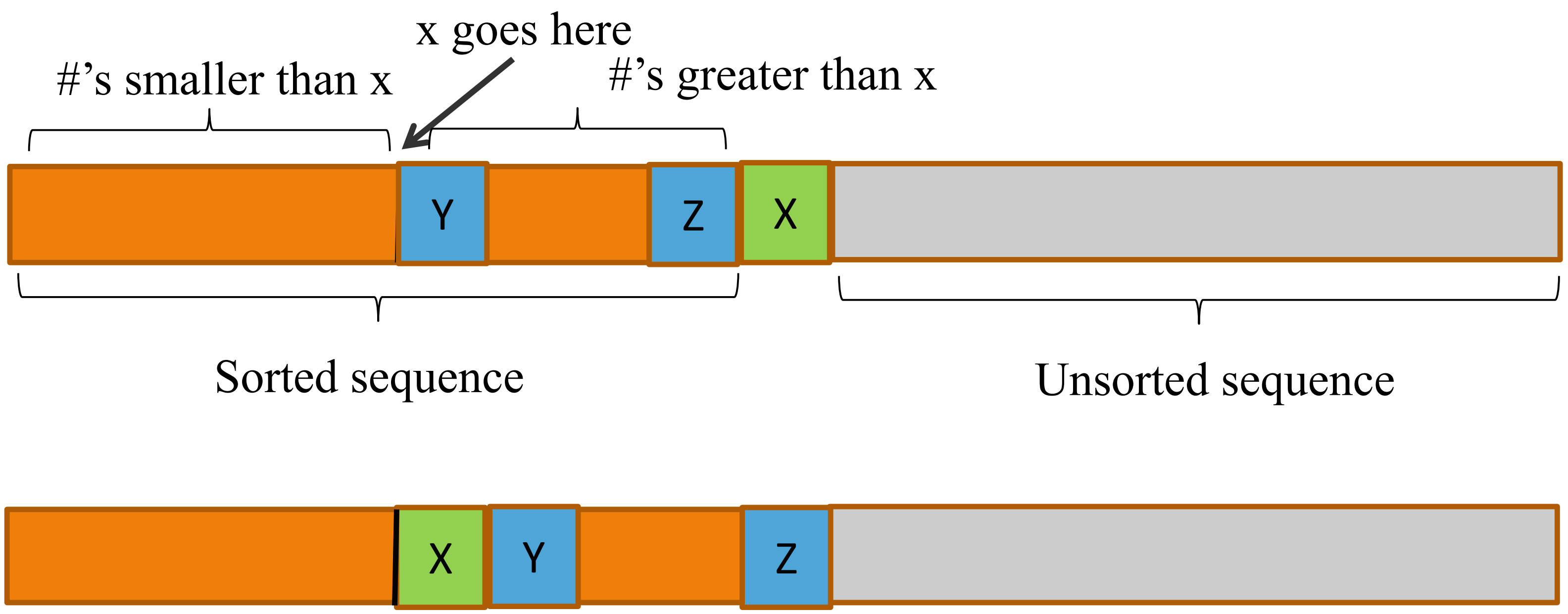
Bubblesort PseudoCode

```
procedure bubblesort($a_1, . . . , a_n$ : real numbers
    with $n \geq 2$)
for $i$ := 1 to $n$-1   {$i$ is the pass number}
    for $j$ := 1 to $n$-$i$ {$j$ is the position in the column}
        if $a_j > a_{j+1}$ then interchange $a_j$ and $a_{j+1}$

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

Insertion Sort

- Simple, but not efficient



Insertion Sort example

- 3 2 4 1 5 {colored list is sorted}
- 2 3 4 1 5 {compare 2 and 3, put 2 in front}
- 2 3 4 1 5 {compare 4 against 2 and 3, already in order}
- 1 2 3 4 5 {compare 1 against 2 3 4 (actually, just against 2), and put in front.
- 1 2 3 4 5 {compare 5 against 1 2 3 and 4, so place it at the end}

Insertion Sort

```
procedure insertion sort ($a_1, a_2, \ldots, a_n$: real numbers with $n \geq 2$)
for $j$ := 2 to $n$   {$a_j$ is the number being inserted}
begin
    $i$ := 1
    while $a_j > a_i$
        $i$ := $i$ + 1  {compare the next number on the list with all those at the
                        beginning of the list until it finds one greater than the number to
                        be inserted. Note those first numbers are in increasing order}
    {$a_j$ should be inserted into the position where $a_i$ is}
    $m$ := $a_j$                  {save the value to be inserted}
                                  {it should be inserted in position $i$}
    for $k$ := $j$ down to $i$+1   {make space to insert the item}
        $a_k := a_{k-1}$

    $a_i := m$

end
```

Greedy algorithms

Frequently a problem is presented which desires an optimization of some procedure.

Surprisingly, one of the simplest approaches often leads to a solution.

This approach selects the best choice at each step (or locally) instead of considering all sequences of steps possible.

Algorithms which make use of the "BEST" choice at each step are called **greedy** algorithms.

Greedy Change-making algorithm

- Use least number of coins to obtain *n* cents
- Greedy: use the largest denomination at each step.

```
Procedure change ( $c_1, c_2, \ldots, c_r$ : values of denominations of coins, where
                   $c_1 > c_2 > .. > c_r$ ;
                   n: integer)
for  i := 1 to r
    while $n \geq c_i$
    begin
        add a coin with value $c_i$ to the change;
        $n := n - c_i$
    end
```

Lemma 1

If *n* is a positive integer, then *n* cents change using quarters, dimes, nickels and pennies using the *fewest coins possible* has:

a) at most two dimes,

b) at most one nickel,

c) at most four pennies,

d) cannot have two dimes and a nickel,

e) the amount of change in dimes, nickels and pennies cannot exceed 24 cents.

Proof by *contrapositive*.

The idea of the proof is to assume e.g., that if we have *more* than two dimes, we can reduce the number of coins, and thus we don't have the fewest coins possible. We do this for every statement a) through d)

continued …

a) If we have three or more dimes, then three dimes can be replaced by a quarter and a nickel (hence, less coins)

b) If we have two or more nickels, we can replace two nickels for a dime (hence less coins)

c) If we have five or more pennies, we can replace them by a nickel (hence less coins)

d) If we have two dimes and a nickel we can replace them by a quarter.

e) From a) through d) the total change in dimes, nickels and pennies cannot be more than 24 cents.

Theorem: The greedy algorithm makes change with the fewest coins possible.

Proof:

- Let *C* be the change breakup for *n* cents using greedy algo.
- Let *C ′* be an optimal change-making breakup for *n* cents.
- Let *q* be the ***number of quarters*** in *C* (similarly *q′* and *C′)*
- Let *r* be the ***number of cents*** using non-quarters in *C* (similarly *r′* and *C′)*
  - $q' \leq q$, because greedy algorithm must use the most quarters compared to *any other algorithm* (it chooses the largest # of quarters possible)
  - $r' \leq 24$ from lemma (add (a) (b) (c) in lemma).
  - Thus, *q′* is the largest possible (not enough cents remain for another quarter), i.e., it is the same as first grabbing as many quarters as possible
  - Thus, $q' = q$, and also therefore $r' = r$

continued …

a) Both methods have the same cents made from “non-quarters” r = r’.

b) Let d and d’ be the number of dimes.
   - From the Lemma, C’ has at most one nickel and at most four pennies
   - Hence, d’ has the largest number of dimes possible from r’
   - We know our algorithm also uses as many dimes as possible from r.
   - Hence, d = d’

c) Repeat argument a) and b) to show they have the same # of nickels

d) Thus, what remains after the nickels is the same in both, which can be added up with pennies in only one way.

Exceptions to the change algorithm

If we do not include nickels in our allowable change, this algorithm will not give the fewest coins possible. Notice the use of the lemma was important in proving the theorem.

In fact if we are not allowed nickels, and look at 30 cents, using the algorithm gives us one quarter and 5 pennies for a total of 6 coins. This is not the fewest number of coins because 3 dimes also make 30 cents.

I.e., ***greedy is not always best!***

Characteristics of algorithms

- *Input.* An algorithm has input values from a specified set
- *Output* From each set of input values an algorithm produces output values from a specified set. The output values are the solution to the problem
- *Definiteness,* The steps of the algorithm are defined precisely.
- *Correctness.* An algorithm should produce the correct value for each set of input values.
- *Finiteness* An algorithm should produce the desired output after a finite number of steps for any input allowed.
- *Effectiveness*   It must be possible to perform each step of an algorithm exactly and in a finite amount of time.
- *Generality,* The procedure should be applicable for all problems of the desired form, not just for a particular set of input values

The Halting Problem

The **Halting Problem** asks whether it is possible to devise a procedure H that takes as input the code of another (arbitrary) procedure P and the input I to that procedure and decides (returns yes or no) whether P(I) will eventually halt with the given input.

The Halting Problem is not solvable. The proof will be by contradiction and therefore such a procedure cannot exist

We will assume such a procedure exists and lead to a contradiction.

Proof

Assume there exists a procedure H(P,I) which returns “halt” if P halts on input I, or returns “no halt” if P loops forever on input I.

As any procedure can be encoded to be input, it becomes just a sequence of bits.

This means *H (P,P) makes sense.* Also, *H* (*H,H*) makes sense.

Now construct a procedure K(P) that uses the output of H(P,P) as follows;

Proof continued

Define *K*(*P*) to be

$$K(P) = \begin{cases} halts & if\ \ H(P,P)\ returns\ "no\ halt" \\ loops\ \ forever & if\ \ H(P,P)\ returns\ "halt" \end{cases}$$

K (*P*) does the opposite of *H* (*P, P*).

Consider now what happens when the procedure *K* is input to *K*.

$$K(K) = \begin{cases} halts & if\ \ H(K,K)\ returns\ "no\ halt" \\ loops\ \ forever & if\ \ H(K,K)\ returns\ "halt" \end{cases}$$

Proof continued

From this we see that if *H* (*K, K*) returns “halt” then *K*(*K*) *does not halt*.

Consider now what *H* (*K, K*) means. This is a procedure that determines if procedure *K* halts upon input *K*. However, this result is not possible because if *H* (*K, K*) decides that K halts on input K, then *K* (*K*) does not halt. This is not possible.

Similarly if *H* (*K, K*) returns “no halt”, then *K* (*K*) halts. This is also not possible.

Such a procedure as *H* cannot exist.

conclusion…

Note: This is not the same as asserting a program exists and we don't know how to write it or that it is very difficult to write such a program!

We are simply saying that H cannot exist.
There is no such thing as H.