American University of Beirut
Dept. of Computer Science

CMPS/Math 211
Discrete Structures

Section 3.1: **Algorithms**

Abu al-Khowarizmi
(ca. 780-850)

Algorithms

- The foundation of computer programming.
- Most generally, an *algorithm* just means a definite procedure for performing some sort of task.
- A computer *program* is simply a description of an algorithm, in a language precise enough for a computer to understand, requiring only operations that the computer already knows how to do.
- We say that a program *implements* (or "is an implementation of") its algorithm.

Algorithms You Already Know

- Grade-school arithmetic algorithms:
  - How to add any two natural numbers written in decimal on paper, using carries.
  - Similar: Subtraction using borrowing.
  - Multiplication & long division.
- Your favorite cooking recipe.
- How to register for classes at a university.

Programming Languages

- Some common programming languages:
  - **Newer:** Java, C, C++, C#, Visual Basic, JavaScript, Perl, Tcl, Pascal, many others…
  - **Older:** Fortran, Cobol, Lisp, Basic
  - Assembly languages, for low-level coding.
- In this class we will use an informal, Pascal-like "*pseudo-code*" language.
- You should know at least 1 real language!

Algorithm Example (English)

- **Task:** Given a sequence $\{a_i\}=a_1,\ldots,a_n$, $a_i \in \mathbf{N}$, say what its largest element is.
- One algorithm for doing this, in English:
  - Set the value of a *temporary variable* v (largest element seen so far) to a_1's value.
  - Look at the next element a_i in the sequence.
  - If $a_i > v$, then re-assign v to the number a_i.
  - Repeat then previous 2 steps until there are no more elements in the sequence, & return v.

Executing an Algorithm

- When you start up a piece of software, we say the program or its algorithm are being *run* or *executed* by the computer.
- Given a description of an algorithm, you can also execute it by hand, by working through all of its steps with pencil & paper.
- Before ~1940, "computer" meant a *person* whose job was to execute algorithms!

Executing the Max algorithm

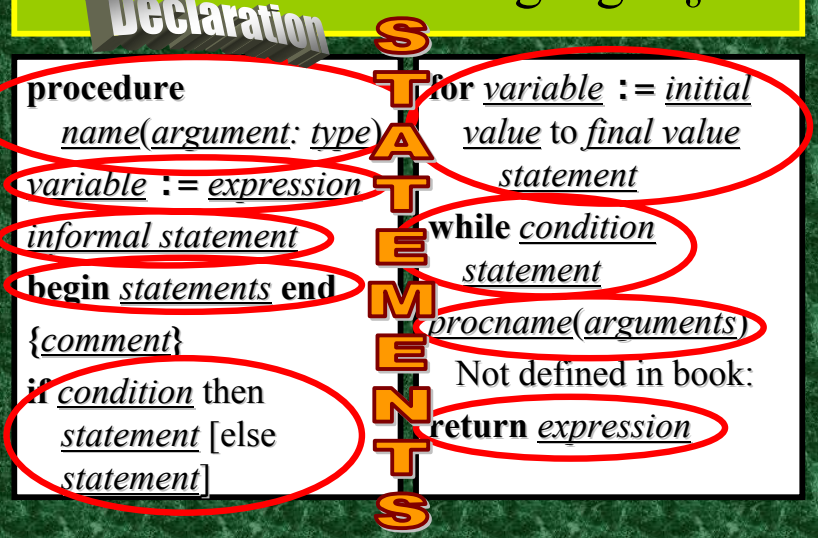
- Let $\{a_i\}=7,12,3,15,8$. Find its maximum…
- Set $v = a_1 = 7$.
- Look at next element: $a_2 = 12$.
- Is $a_2 > v$? Yes, so change v to 12.
- Look at next element: $a_2 = 3$.
- Is $3 > 12$? No, leave v alone….
- Is $15 > 12$? Yes, v=15…

Algorithm Characteristics

Some important general features of algorithms:

- *Input*. Information or data that comes in.
- *Output*. Information or data that goes out.
- *Definiteness*. Algorithm is precisely defined.
- *Correctness*. Outputs correctly relate to inputs.
- *Finiteness*. Won't take forever to describe or run.
- *Effectiveness*. Individual steps are all do-able.
- *Generality*. Works for many possible inputs.
- *Efficiency*. Takes little time & memory to run.

Our Pseudocode Language: §A2



procedure *procname*(*arg*: *type*)

- Declares that the following text defines a procedure named *procname* that takes inputs (*arguments*) named *arg* which are data objects of the type *type*.
  - Example:
    procedure *maximum*(*L*: list of integers)
    [statements defining *maximum*…]

variable := *expression*

- An *assignment* statement evaluates the expression *expression*, then reassigns the variable *variable* to the value that results.
  - Example assignment statement:
    $v := 3x+7$ (If x is 2, changes v to 13.)
- In pseudocode (but not real code), the *expression* might be informally stated:
  - $x :=$ the largest integer in the list L

Informal statement

- Sometimes we may write a statement as an informal English imperative, if the meaning is still clear and precise: *e.g.*,"swap *x* and *y*"
- Keep in mind that real programming languages never allow this.
- When we ask for an algorithm to do so-and-so, writing "Do so-and-so" isn't enough!
  - Break down algorithm into detailed steps.

begin *statements* **end**

- Groups a sequence of statements together:

```
begin
   statement 1
   statement 2
   ...
   statement n
end
```

Curly braces {} are used instead in many languages.

- Allows the sequence to be used just like a single statement.
- Might be used:
  - After a **procedure** declaration.
  - In an **if** statement after **then** or **else**.
  - In the body of a **for** or **while** loop.

{*comment*}

- Not executed (does nothing).
- Natural-language text explaining some aspect of the procedure to human readers.
- Also called a *remark* in some real programming languages, *e.g.* BASIC.
- Example, might appear in a *max* program:
  - {Note that *v* is the largest integer seen so far.}

if *condition* **then** *statement*

- Evaluate the propositional expression *condition*.
  - If the resulting truth value is **True**, then execute the statement *statement*;
  - otherwise, just skip on ahead to the next statement after the **if** statement.
- Variant: **if** *cond* **then** *stmt1* **else** *stmt2*
  - Like before, but iff truth value is **False**, executes *stmt2*.

while *condition* *statement*

- *Evaluate* the propositional (Boolean) expression *condition*.
- If the resulting value is **True**, then execute *statement*.
- Continue repeating the above two actions over and over until finally the *condition* evaluates to **False**; then proceed to the next statement.

while *condition* *statement*

- Also equivalent to infinite nested **if**s, like so:

```
if condition
    begin
        statement
        if condition
            begin
                statement
                ...(continue infinite nested if's)
            end
    end
```


for *var* := *initial* to *final* *stmt*

- *Initial* is an integer expression.
- *Final* is another integer expression.
- **Semantics:** Repeatedly execute *stmt*, first with variable *var* := *initial*, then with *var* := *initial*+1, then with *var* := *initial*+2, *etc*., then finally with *var* := *final*.

for *var* := *initial* to *final* *stmt*

- **For** can be exactly defined in terms of **while,** like so:

```
begin
    var := initial
    while var ≤ final
        begin
            stmt
            var := var + 1
        end
end
```

procedure(*argument*)

- A *procedure call* statement invokes the named *procedure*, giving it as its input the value of the *argument* expression.
- Various real programming languages refer to procedures as *functions* (since the procedure call notation works similarly to function application *f*(*x*)), or as *subroutines*, *subprograms*, or *methods*.

Max procedure in pseudocode

procedure *max*($a_1, a_2, \ldots, a_n$: integers)
  $v := a_1$ {largest element so far}
  for $i := 2$ **to** n {go thru rest of elems}
    if $a_i > v$ **then** $v := a_i$ {found bigger?}
  {at this point *v*'s value is the same as the largest integer in the list}
  return *v*

Another example task

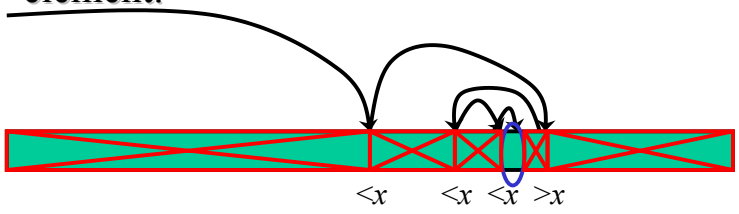
- Problem of *searching an ordered list*.
  - Given a list *L* of *n* elements that are sorted into a definite order (*e.g.*, numeric, alphabetical),
  - And given a particular element *x*,
  - Determine whether *x* appears in the list,
  - and if so, return its index (position) in the list.
- Problem occurs often in many contexts.
- Let's find an *efficient* algorithm!

Search alg. #1: Linear Search

procedure *linear search*
  (x: integer, $a_1, a_2, \ldots, a_n$: distinct integers)
  $i := 1$ {start at beginning of list}
  while $(i \leq n \wedge x \neq a_i)$ {not done, not found}
    $i := i + 1$ {go to the next position}
  if $i \leq n$ **then** *location* $:= i$ {it was found}
  else *location* $:= 0$ {it wasn't found}
  return *location* {index or 0 if not found}

Search alg. #2: Binary Search

- Basic idea: On each step, look at the *middle* element of the remaining list to eliminate half of it, and quickly zero in on the desired element.



Search alg. #2: Binary Search

procedure *binary search*
(x:integer, $a_1, a_2, \ldots, a_n$: distinct integers)
i := 1 {left endpoint of search interval}
j := n {right endpoint of search interval}
while $i<j$ **begin** {while interval has >1 item}
m := $\lfloor (i+j)/2 \rfloor$ {midpoint}
if $x>a_m$ **then** i := $m+1$ **else** j := m
end
if $x = a_i$ **then** *location* := i **else** *location* := 0
return *location*

Practice exercises

- Devise an algorithm that finds the sum of all the integers in a list. [2 min]
- **procedure** *sum*($a_1, a_2, \ldots, a_n$: integers)
  s := 0 {sum of elems so far}
  for i := 1 **to** n {go thru all elems}
  s := $s + a_i$ {add current item}
  {at this point s is the sum of all items}
  return s

Sorting Algorithms

- Sorting is a common operation in many applications.
  - *E.g.* spreadsheets and databases
- It is also widely used as a subroutine in other data-processing algorithms.
- Two sorting algorithms shown in textbook:
  - Bubble sort
  - Insertion sort

} However, these are *not* very efficient, and you should not use them on large data sets!

We'll see some more efficient algorithms later in the course.

Insertion Sort Algorithm

- English description of algorithm:
  - For each item in the input list,
    - "Insert" it into the correct place in the sorted output list generated so far. Like so:
      - Use linear or binary search to find the location where the new item should be inserted.
      - Then, shift the items from that position onwards down by one position.
      - Put the new item in the hole remaining.
  - Use this algorithm to put the elements of the list 3, 2, 4, 1, 5 in increasing order

Insertion Sort

Alg.: INSERTION-SORT*(A)*

```
for j ← 2 to n
    do key ← A[ j ]
        Insert A[ j ] into the sorted sequence A[1 . . j -1]
        i ← j - 1
        while i > 0 and A[i] > key
            do A[i + 1] ← A[i]
                i ← i - 1
        A[i + 1] ← key
```


Correctness

- **Loop Invariant (a condition that never changes)**: at the start of the **for** loop the elements in A[1 . . j-1] are in sorted order

Proving Loop Invariants

- **Initialization (base case):** It is true prior to the first iteration of the loop
- **Maintenance (inductive step):** If it is true before an iteration of the loop, it remains true before the next iteration
- **Termination:** When the loop terminates, the invariant gives us a useful property that helps show that the algorithm is correct. Stop the induction when the loop terminates

Loop Invariant for Insertion Sort

- **Initialization:**
  - Just before the first iteration, $j = 2$:

    the subarray $A[1 . . j-1] = A[1]$, (the element originally in $A[1]$) – is sorted

Loop Invariant for Insertion Sort

- **Maintenance:**
  - Assume the list $A[1,\ldots,j-1]$ is sorted. Show that at the end of one loop iteration, $A[1,\ldots,j]$ is also sorted.
  - the **while** inner loop moves $A[j-1]$, $A[j-2]$, $A[j-3]$, and so on, by one position to the right until the proper position for key (which has the value that started out in $A[j]$) is found.

Loop Invariant for Insertion Sort

- **Maintenance:**
  - At that point, the value of key is placed into this position.
  - Since all elements having been moved to the right side of the key are already sorted, and the key is in its right position in that all elements to its left are less than itself, the entire list $A[1,\ldots,j]$ is also so.

Loop Invariant for Insertion Sort

- **Termination:**
  - The outer **for** loop ends when $j > n$ (i.e, $j = n + 1$)

    $\Rightarrow j-1 = n$
  - Replace n with $j-1$ in the loop invariant:
    - the subarray $A[1 . . n]$ consists of the elements originally in $A[1 . . n]$, but in sorted order
- The entire array is sorted!