UC Davis, ECS20, Winter 2017

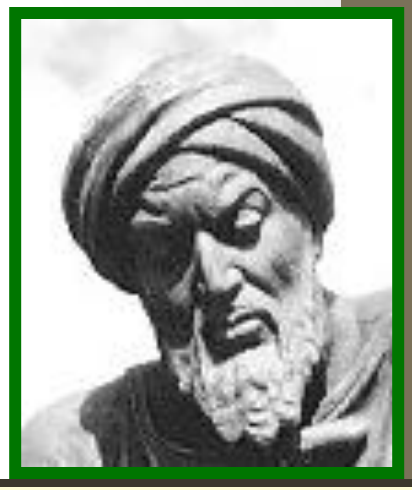
Discrete Mathematics for Computer Science

Prof. Raissa D'Souza
(slides adopted from Michael Frank and Haluk Bingöl)

Lecture 11

Algorithms 3.1-3.2

Algorithms



Abu al-Khowarizmi
(ca. 780-850)

- Member of the "House of wisdom" in Bagdad
  - Invented Algebra & procedures for arithmetic operations
  - Invented Algorithms

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 UCD.

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/ deterministic.
- *Correctness*. Outputs correctly relate to inputs. (Hard to prove; bugs abound in code.)
- *Finiteness*. Won't take forever to describe or run.
- *Effectiveness*. Individual steps are all do-able.
- *Generality*. Works for all inputs of a specified domain.
- *Efficiency*. Takes little time & memory to run.

Inventing an Algorithm

- Requires a lot of creativity and intuition
  - Like writing proofs.

- Unfortunately, we can't give you an algorithm for inventing algorithms.
  - Just look at lots of examples…
  - And practice (preferably, on a computer)
  - And look at more examples…
  - And practice some more… etc., etc.

- Typically no unique solution (but some algorithms are more efficient than others)

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!
  (e.g. the movie "Hidden Figures")

Programming Languages

- Some common programming languages:
  - **Newer:** Java, C, C++, C#, Visual Basic, JavaScript, Perl, Tcl, Pascal, Python, 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

Task: Given an arbitrary sequence

$\{a_i\}=a_1,\ldots,a_n$, $a_i \in \mathbb{N}$,

find what is the value of the largest element.

(e.g. Let $\{a_i\}=7,12,3,15,8$)

Example of 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…

How to we tell the computer when to stop searching the list?

Our Pseudocode Language:

- **procedure** *name*(*argument: type*)
- *variable* := *expression*
- *informal statement*
- **begin** *statements* **end**
- {*comment*}
- **if** *condition* then *statement* [else *statement*]
- **for** *variable* := *initial value* to *final value* *statement*
- **while** *condition* *statement*
- **return** *expression*

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

- Often just say "*procname(arg)*" without the 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 := 3*x*+7 (If *x* is 2, changes *v* to 13.)
  - Also use
  - *v* <- 3x + 7

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.

Swap two values

- V1 := 10
- V2:= 20

```
swap (V1, V2)
        V1 = V2
        V2 = V1        (What ends up being the value of V2?)

swap (V1, V2)
        temp := V1
        V1 = V2
        V2 = temp    (This one works)
```

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*.
- **Question:** What happens if *stmt* changes the value of var, or the value that *initial* or *final* evaluates to?

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
```

Search alg. #1: Linear Search

Find if an specific item is a member of a specified sequence

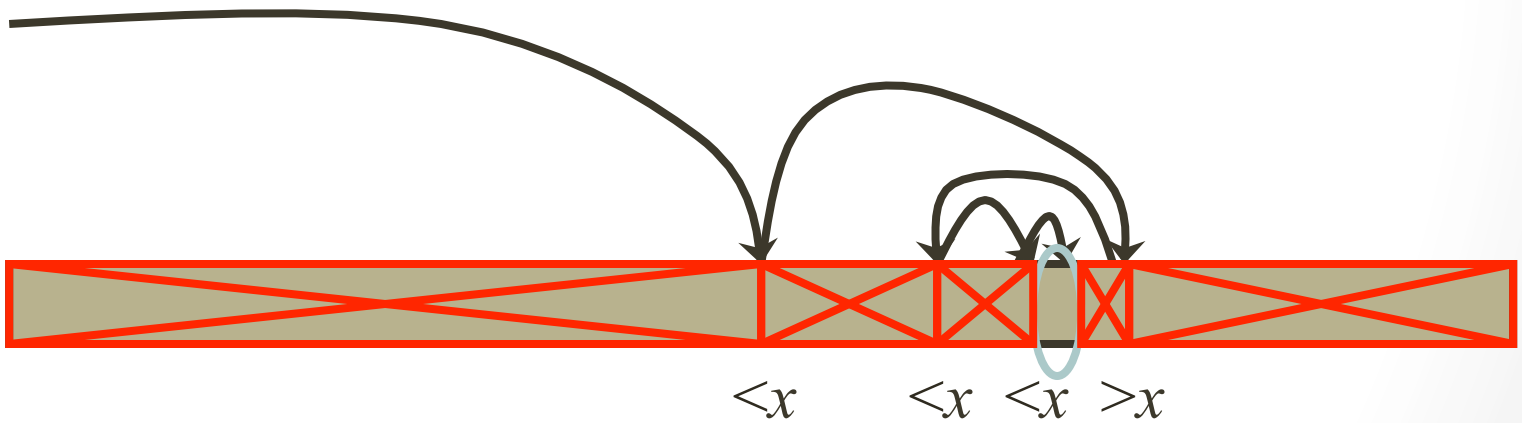
- **procedure** *linear search*
  (x: integer, a_1, a_2, …, a_n: distinct integers)
  i := 1 {start at beginning of list}
  while ($i \leq n \wedge x \neq a_i$) {not done, not found}
  i := i + 1 {go to the next position}
  if $i \leq n$ **then** *location* := i {it was found}
  else *location* := 0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

Can we do it more efficiently?

- What if the sequence is sorted in some way?
  - From smallest to largest
  - Alphabetically, etc.

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, …, 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*

Another example

- 2.1.3: Devise an algorithm that finds the sum of all the integers in a list of integers

```
procedure sum(a_1, a_2, …, 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

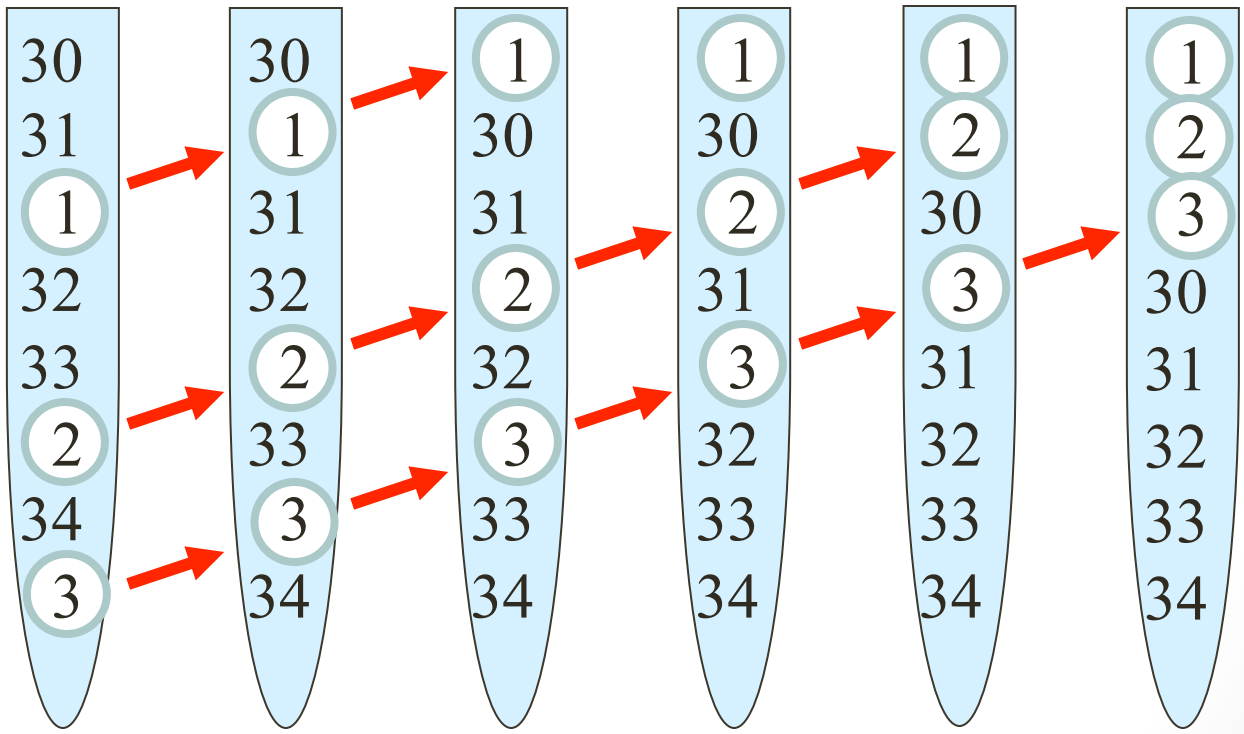
- Sorting is a common operation in many applications.
  - *E.g.* spreadsheets and databases
- It is also widely used as a subroutine in other data-processing algorithms.
- Two sorting algorithms shown in textbook:
  - Bubble sort
  - Insertion sort

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.

Bubble Sort

- Smallest elements "float" up to the top of the list, like bubbles in a container of liquid.



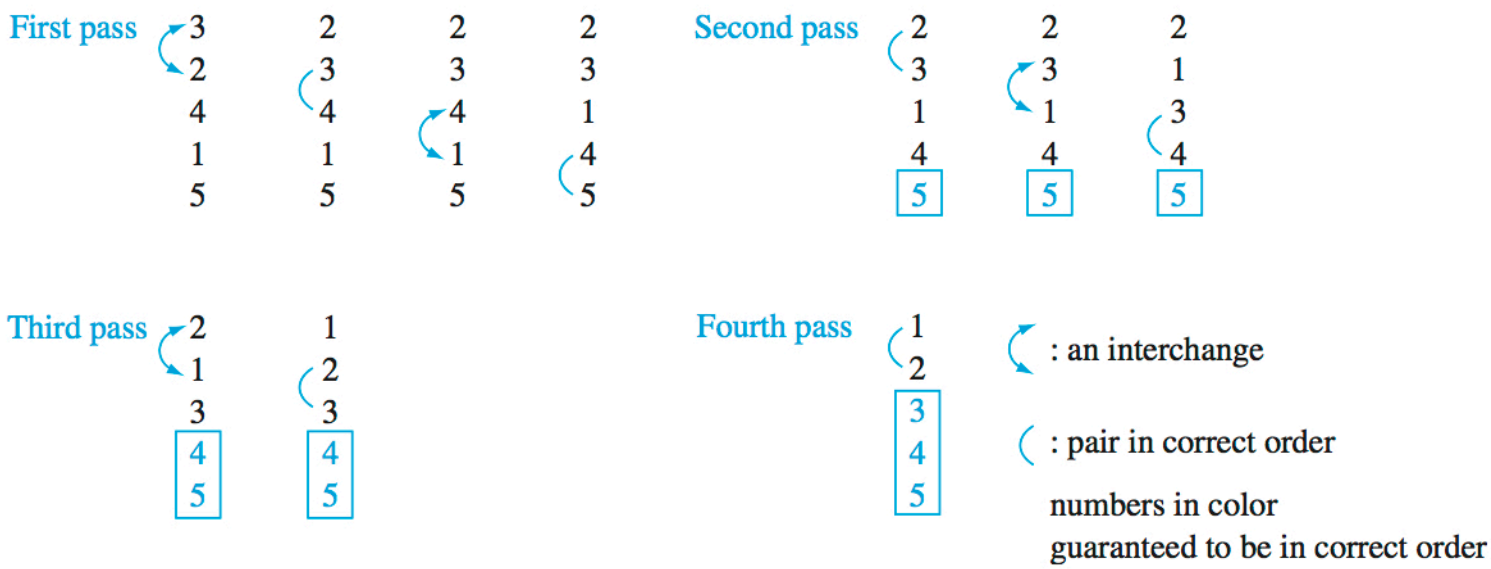


FIGURE 1 **The Steps of a Bubble Sort.**

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.

Review §3.1: Algorithms

- Characteristics of algorithms.
- Pseudocode.
- Examples: Max algorithm, linear search & binary search algorithms, sorting.
- Intuitively we see that binary search is much faster than linear search, but how do we analyze the efficiency of algorithms formally?
- Use methods of *algorithmic complexity*, which utilize the order-of-growth concepts from §1.8.