Lecture 11

Algorithms 3.1-3.2
Algorithms

• 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!
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...]
  ```
\textbf{variable} := \textbf{expression}

- An \textit{assignment} statement evaluates the expression \textit{expression}, then reassigns the variable \textit{variable} to the value that results.
  
  - Example assignment statement: 
    \[ v := 3x + 7 \] (If \( x \) is 2, changes \( v \) to 13.)

- Also use
  
  \[ v \gets 3x + 7 \]
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

\[
\begin{align*}
\text{swap (V1, V2)} & \\
V1 &= V2 \\
V2 &= V1 & (\text{What ends up being the value of V2?})
\end{align*}
\]

\[
\begin{align*}
\text{swap (V1, V2)} & \\
temp &= V1 \\
V1 &= V2 \\
V2 &= temp & (\text{This one works})
\end{align*}
\]
begin statements end

• Groups a sequence of statements together:
  ```plaintext
  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.
• 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 ν is the largest integer seen so far.\}
if \textbf{condition then statement}

- Evaluate the propositional expression \textbf{condition}.
  - If the resulting truth value is \textbf{True}, then execute the statement \textbf{statement};
  - otherwise, just skip on ahead to the next statement after the \textbf{if} statement.

- Variant: \textbf{if cond then stmt1 else stmt2}
  - Like before, but iff truth value is \textbf{False}, executes \textbf{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 \textit{condition} statement

• Also equivalent to infinite nested if statements, like so:

\begin{verbatim}
if \textit{condition} \\
begin \\
  \textit{statement} \\
if \textit{condition} \\
begin \\
  \textit{statement} \\
... (continue infinite nested if’s) \\
end \\
end
\end{verbatim}
for \( \text{var} := \text{initial} \) to \( \text{final} \) \( \text{stmt} \)

- **Initial** is an integer expression.
- **Final** is another integer expression.
- **Semantics:** Repeatedly execute \( \text{stmt} \), first with variable \( \text{var} := \text{initial} \), then with \( \text{var} := \text{initial} + 1 \), then with \( \text{var} := \text{initial} + 2 \), etc., then finally with \( \text{var} := \text{final} \).
- **Question:** What happens if \( \text{stmt} \) changes the value of \( \text{var} \), or the value that \( \text{initial} \) or \( \text{final} \) 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
  
  \( (x: \text{integer}, a_1, a_2, \ldots, a_n: \text{distinct integers}) \)
  
  \( i := 1 \)  \{start at beginning of list\}
  
  \( \textbf{while} \ (i \leq n \land x \neq a_i) \ \{\text{not done, not found}\} \)
  
  \( i := i + 1 \)  \{go to the next position\}

  \( \textbf{if} \ i \leq n \ \textbf{then} \ location := i \ \{\text{it was found}\} \)

  \( \textbf{else} \ location := 0 \ \{\text{it wasn’t found}\} \)

  \( \textbf{return} \ location \ \{\text{index or 0 if not found}\} \)
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`

\[(x:integer, a_1, a_2, \ldots, a_n: \text{distinct integers})\]

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\( (a_1, a_2, ..., a_n): \text{integers} \)

\[
\begin{align*}
s & := 0 \quad \text{\{sum of elems so far\}} \\
\text{for } i & := 1 \text{ to } n \quad \text{\{go thru all elems\}} \\
& \quad s := s + a_i \quad \text{\{add current item\}} \\
\text{at this point } s & \text{ is the sum of all items} \\
\text{return } s
\end{align*}
\]
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.
3.1 Algorithms

The steps of this algorithm are illustrated in Figure 1. Begin by comparing the first two elements, 3 and 2. Because 3 > 2, interchange 3 and 2, producing the list 2, 3, 4, 1, 5. Because 3 < 4, continue by comparing 4 and 1. Because 4 > 1, interchange 1 and 4, producing the list 2, 3, 1, 4, 5. Because 4 < 5, the first pass is complete. The first pass guarantees that the largest element, 5, is in the correct position.

The second pass begins by comparing 2 and 3. Because these are in the correct order, 3 and 1 are compared. Because 3 > 1, these numbers are interchanged, producing 2, 1, 3, 4, 5. Because 3 < 4, these numbers are in the correct order. It is not necessary to do any more comparisons for this pass because 5 is already in the correct position. The second pass guarantees that the two largest elements, 4 and 5, are in their correct positions.

The third pass begins by comparing 2 and 1. These are interchanged because 2 > 1, producing 1, 2, 3, 4, 5. Because 2 < 3, these two elements are in the correct order. It is not necessary to do any more comparisons for this pass because 4 and 5 are already in the correct positions. The third pass guarantees that the three largest elements, 3, 4, and 5, are in their correct positions.

The fourth pass consists of one comparison, namely, the comparison of 1 and 2. Because 1 < 2, these elements are in the correct order. This completes the bubble sort.

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.