ICS141: Discrete Mathematics for Computer Science I

Dept. Information & Computer Sci., University of Hawaii

Jan Stelovsky
based on slides by Dr. Baek and Dr. Still
Originals by Dr. M. P. Frank and Dr. J.L. Gross
Provided by McGraw-Hill

Lecture 13

Chapter 2. Basic Structures

2.4 Sequences and Summations

Chapter 3. The Fundamentals

3.1 Algorithms

Nested Summations

University of Hawaii

- These have the meaning you'd expect.

$$\sum_{i=1}^{4}\sum_{j=1}^{3} ij = \sum_{i=1}^{4}\left(\sum_{j=1}^{3} ij\right)$$

- Note issues of free vs. bound variables, just like in quantified expressions, integrals, etc.

Nested Summations (cont.)

- Summations can be interchanged:

$$\sum_{S(x)} \sum_{T(y)} f(x,y) = \sum_{T(y)} \sum_{S(x)} f(x,y)$$

$$\sum_{i=k}^{l} \sum_{j=m}^{n} f(i,j) = \sum_{j=m}^{n} \sum_{i=k}^{l} f(i,j)$$

Summations: Conclusion

University of Hawaii

- You need to know:
  - How to read, write & evaluate summation expressions like:

$$\sum_{i=j}^{k} a_i \qquad \sum_{i=j}^{\infty} a_i \qquad \sum_{x \in X} f(x) \qquad \sum_{P(x)} f(x)$$

  - Summation manipulation laws we covered.
  - Shortcut closed-form formulas, & how to use them.

3.1 Algorithms

- In general, an *algorithm* just means a defined procedure for performing and completing some sort of task

- Algorithms are the foundation of computer programming.

3.1 Programs

- 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.
- Often we when we say *program* we mean as collection of algorithms

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 UH.
- How to get from your home to UH.

Executing an Algorithm

University of Hawaii

- When you start up a piece of software, we say the program or its algorithm is *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!

Algorithm Example (English)

University of Hawaii

- **Task:** Given a sequence $\{a_i\} = a_1,\ldots,a_n$, where $a_i \in \mathbf{N}$, 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 the Max algorithm

- Let $\{a_i\}$ = 7, 12, 3, 15. 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: $a_3 = 3$.
- Is 3 > 12 ? No, leave v alone….
- Is 15 > 12 ? Yes, $v = 15$…
- We are done: return 15

Algorithm Characteristics

The necessary features of an algorithm:

- *Definiteness.* The steps of an algorithm must be precisely defined.
- *Effectiveness.* Individual steps are all do-able.
- *Finiteness.* It won't take forever to produce the desired output for any input in the specified domain.
- *Output.* Information or data that goes out.

Other important features of an algorithm:

- *Input*. Information or data that comes in.
- *Correctness.* Outputs correctly relate to inputs.
- *Generality.* Works for many possible inputs.
- *Efficiency.* Takes little time & memory to run.

Parts of an Algorithm

- Keywords (reserved words)
  - special words that have specific meaning: **begin**, **if**, **for**
- Names (identifers)
  - you define so that you can later can refer to
- Constants (litterals)
  - e.g. 1, 3.14, "some text"
- Operands
  - e.g., +, -, ^, and, or, not, …
- Types
  - data are objects, each object has a specific type,
    e.g. integer, string, boolean (true, false)
    or even a type that you define yourself

Parts of an Algorithm (cont.)

University of Hawaii

- Variables
  - placeholders for data objects, whose values (and in some languages even types) can change
- Expressions
  - combine the values of variables (and constants, and results of procedure calls) using operands to produce a value
- Statements
  - steps in the algorithm
- Procedures
  - parts of an algorithm that are defined ("declared") separately, named and referenced (called) at different places in the algorithm (sub-algorithms)

Pseudocode (Appendix 3)

University of Hawaii

Declaration

STATEMENTS

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

procname(*arguments*)

Not defined in book:

return *expression*

Procedure Declaration

University of Hawaii

- procedure *<u>procedure</u>*
  (*<u>argument1</u>*: *<u>type1</u>*, *<u>argument2</u>*: *<u>type2</u>*, …)
- Declares that the subsequent text defines a procedure named *<u>procedure</u>* that takes inputs (arguments) named *<u>argument1</u>* which is data object of the type *<u>type1</u>*, *<u>argument2</u>* of the type *<u>type2</u>*, etc.
  - Example:
    procedure *maximum*(*L*: list of integers)
    [statements defining *maximum*…]

Assignment

- *variable* := *expression*
  - An assignment statement evaluates the expression *expression*, then lets the variable *variable* hold 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
- References to this *variable* in expressions will be subsituted by the value that was assigned last

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. (Except as a comment.)
- 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.

Compound Statement

- **begin** *statements* **end**
  - Groups a sequence of statements together:

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

Curly braces {} or just indentation are used 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.

Comments

- {*comment*}
  - Not executed (does nothing).
  - Natural-language text explaining some aspect of the procedure to human readers.
  - Also called a *remark* in some programming languages, *e.g.* BASIC.
  - Example, as it may appear in a *max* program:
    - {Note that *v* is the largest integer seen so far.}
- Comments should be well formulated, but terse!
  - Better: {*v* is largest so far}
  - More comments, less words, but more precise words
- Different languages use different syntax:
  - # …, // …, /*several lines*/, …

if Statement

- **if** *condition* **then** *statement*
  - Evaluate the propositional expression *condition*.
  - If the resulting truth value is **True**, then execute the statement *statement*;
  - Otherwise, just skip and go ahead to the next statement after the **if** statement.
- **if** *condition* **then** *statement* **else** *statement2*
  - Like before, but iff truth value is **False**, executes *statement2*.

while Statement

University of Hawaii

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

while 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 Statement

University of Hawaii

- for *variable* := *initial* to *final* *statement*
  - *Initial* is an integer expression.
  - *Final* is another integer expression.
  - **Semantics** (i.e., meaning):
    Repeatedly execute *statement*,
    first with variable *variable* := *initial*,
    then with *variable* := *initial* + 1,
    then with *variable* := *initial* + 2, *etc*.,
    then finally with *variable* := *final*.
  - **Question:** What happens if *statement* changes the value of *variable*, or the *initial* or *final* values?

for Statement

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

begin
 $\underline{variable}$:= $\underline{initial}$
 while $\underline{variable} \leq \underline{final}$ **begin**
  $\underline{statement}$
  $\underline{variable}$:= $\underline{variable} + 1$
 end
end

Procedure Call

- *procedure* (*argument1*, *argument2*, …)
  - A procedure call statement invokes the procedure named *procedure*, giving it the values of the *argument1*, *argument2*, …, *argument-n* expressions as its inputs.
  - Various real programming languages refer to procedures as ***functions***
    (since the procedure call notation works similarly to function application $f(x)$),
    or as ***methods***, ***subprograms*** or ***subroutines***.

Max Procedure in Pseudocode

University of Hawaii

procedure *max*($a_1, a_2, \ldots, a_n$: integers) **begin**
  $v := a_1$ {largest element so far}
  for $i := 2$ **to** n **begin** {go through elements $a_2, \ldots, a_n$}
   {check element a_i}
    if $a_i > v$ **then begin** {is a_i bigger than max up to now}
     $v := a_i$
   end {now value in v is the largest in $a_1, \ldots, a_i$}
  end {now value in v is the largest integer in $a_1, \ldots, a_n$}
  return v
end

- The {now ..} comments are "postconditions"

Inventing an Algorithm

- Requires a lot of creativity and intuition
  - Like writing proofs.
- Unfortunately, we ther is no 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.

Practice Exercises

University of Hawaii

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

- **procedure** *sum*($a_1, a_2, \ldots, a_n$: integers)

  $s := 0$ {sum of elements so far}

  for $i := 1$ **to** n {go thru all elements}

  $s := s + a_i$ {add current item}

  {now s is the sum of all items}

  return s

Searching Algorithms

University of Hawaii

- 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!

Alg. #1: Naïve Linear Search

{Given a list of integers and an integer x to look up, returns the index of x within the list or 0 if x is not in the list}

procedure *linear_search*

(x: integer, $a_1, a_2, \ldots, a_n$: distinct integers)

$i := 1$ {start at beginning of list}

while $(i \leq n \land x \neq a_i)$ {not done and not found}

$i := i + 1$ {go to the next position}

if $i \leq n$ **then** $index := i$ {it was found}

else $index := 0$ {it wasn't found}

return *index* {index where found or 0 if not found}