Introduction to Computer Science

Algorithms

Dr. Lonnie Cheney

Outline

- The Concept of an Algorithm
- Algorithm Representation
- Algorithm Discovery
- Iterative Structures
- Recursive Structures
- Efficiency

Definition of an Algorithm

An algorithm is an ordered set of unambiguous, executable steps that defines a terminating process.
Example 1

Task: find largest positive integer

Step 1: make a list of all the positive integers
Step 2: arrange this list in order
Step 3: copy the first integer in this list
Step 4: stop

Not an algorithm because

- step 2 is ambiguous: what order?
- step 1 is not executable
  - although it has finite steps, it does not terminated!

Example 2

Task: take exam

Step 1: read unanswered question
Step 2: if you know the answer, write it down
Step 3: go back to step 1 if there are still unanswered questions
Step 4: stop

Is this an algorithm?

Abstract Nature of Algorithms

Distinction between “algorithm” and its “representation”

- algorithm: “underlying concept”
- representation: way of communicating the algorithm

Further distinction:

- program: formal representation of an algorithm
- process: activity of executing the program

Outline

- The Concept of an Algorithm
- Algorithm Representation
- Algorithm Discovery
- Iterative Structures
- Recursive Structures
- Efficiency
Representation

- algorithm is an abstraction of a process
- need to be able to tell other people (or a computer) how to do it
- Write it down!
- syntax – rules for how to write it down
- semantics – rules for determining meaning

Example: Folding a bird

Example: Euclidean Algorithm

Algorithm to find the greatest common divisor (gcd) of two positive integers:

1. Assign M & N the value of the larger and the smaller of the two input values, respectively.
2. Divide M by N, and call the remainder R.
3. If R is not zero, then assign M the value of N, assign N the value of R, and return to step 2; otherwise, the gcd is the value assigned to N.
Primitives

- well-defined set of building blocks
- must be unambiguous
- appropriate level of detail
- can be combined with other primitives to form “higher-level” primitives
- machine language primitives will work, however, level of detail is inappropriate for most problems

Origami Primitives

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>Shade one side of paper</td>
<td>Distinguishes between different sides of paper</td>
</tr>
<tr>
<td>as in</td>
<td>as in</td>
</tr>
<tr>
<td>Represents a valley fold</td>
<td>so that represents</td>
</tr>
</tbody>
</table>

Representations

- programming languages, e.g., C++ (implement)
- pseudo-code, i.e., English-like statements with a special simple syntax (design)

If it is cold, wear a jacket, otherwise a T-shirt is fine. Depending on the temperature, wear a jacket or not.

if (temp less than 70) then (wear jacket)
else (wear T-shirt)

Like writing an outline or taking notes - use it to develop understanding of problem

Pseudocode

- sequential statements (execute statements from top to bottom)
- assignment
  name ← expression
- conditional statements (selection)
  if (condition) then (activity 1)
  else (activity 2)
- iteration statements (loops)
  while (condition) do (activity)
Modular Design

- ease of comprehension and modification
- organize / structure the algorithm into major sections or modules
- similar in concept to using an outline to write a paper
- each module can be considered an algorithm as well
- modules provide different levels of detail

Procedures

- how to represent a module
- define the module - give a module a name
- refer to the module - instead of writing out all steps of that module, just give the name of the module
- pseudocode: procedure nameOfModule

Example

- refer to module brushTeeth from 2 other modules

```
procedure getReadyForSchool
    wakeUp
    eatBreakfast
    brushTeeth

procedure getReadyForBed
    eatDinner
    brushTeeth
    turn off lights
```

Definition of Procedure

```
procedure brushTeeth
    go to bathroom
    grab toothbrush
    open toothpaste tube
    put brush bristles under tube opening
    while (not enough paste on brush) do
        (squeeze tube)
    put brush in mouth
    replace tube cap
    while (2 minutes not past) do
        (move brush up and down, side to side)
    rinseMouth
```
Considerations

- Two things you can do to a module:
  - use it
  - define it
- You can use a module before it’s defined, but the algorithm is NOT COMPLETE until the module is defined
- Modules can be used several times without having to define them over and over.

How to write pseudocode?

- what are our primitives in pseudocode?
  - anything we need. we just make it up.
- so I can write “brushHair” and that’s my algorithm?
  - only if brushHair is executable on your target machine.
  - if it’s not, you need to treat brushHair like a procedure, and define it in more specific terms.

Outline

- The Concept of an Algorithm
- Algorithm Representation
- Algorithm Discovery
- Iterative Structures
- Recursive Structures
- Efficiency

The Art of Problem Solving

- an algorithm is a precise specification of how to solve a problem, but how do I solve problems?
- problem solving is a skill that is developed and learned
- there is NO step by step recipe for solving problems
- the key is PRACTICE !!!
Polya’s Phases

• Understand the problem!!!
• Get an idea about what kind of algorithm is needed
• Create the algorithm and implement it (program)
• Evaluate the program for accuracy and general use

Phases, NOT Steps

• not carried out in sequence typically
• depend on our experience, creativity, and imagination
• to understand a problem, sometimes we need to implement an algorithm (prototype)
• problem solving is part of software engineering

“Getting a Foot in the Door”

• the desk drawer approach
• solve only part of a problem first
• work backwards from a solution
• study solutions to similar problems
• solve specific cases first, then generalize
• stepwise refinement (modular approach)

Outline

• The Concept of an Algorithm
• Algorithm Representation
• Algorithm Discovery
• Iterative Structures
• Recursive Structures
• Efficiency
Iteration

- iteration involves repeating some action
- but how often or for how long should action be repeated?
  - as long as a certain condition is given
    - "while some condition is given, do this"
  - until some event/condition occurs
    - "repeat doing this until condition occurs"

Iteration and Search

- searching is a process of examining objects from a list of objects
  - until you find the one that you are looking for, keep on searching
    - iteration is applicable to the search process

Sequential Search

- problem:
  - given a list of objects, and an object to search for, determine if the named object is in the list of objects (this is the WHAT)
  - one method is to examine each object in the list sequentially, until either we find the object we’re looking for, or we look through everything and see that the object is not there (this is the HOW)

Pseudocode of Sequential Search

Given: a list and an object
select the first element in the list
while (selected element is not the object) do {
  select the next element in the list
}

- problem: does this terminate?
- what happens if we run out of elements?
- how do we handle the case where the object is NOT in the list?
**Refinement 1**

```plaintext
procedure sequentialSearch(object, list)
    select first object in list
    while (there are unselected objects in list) AND
      (selected element is not the object) do {
        select the next element in the list
      }
    if (we found object) then print("found it")
      else print("not there")
```

**Refinement 2**

```plaintext
procedure sequentialSearch(object, list)
    item = firstObjectIn(list)
    while ((not isLast(item) and (item != object)) do {
        item = nextObjectIn(list)
    }
    if (item == object) then print("found it")
      else print("not there")
```

Does it work for all possible cases?

**Components of a Loop**

- **initialize**: set up the conditions, which the loop tests
- **test**: compare current loop condition to the stopping condition and terminate if they match
- **modify**: change the conditions so that the loop will eventually terminate

**while Loop**

```plaintext
while(condition) do {
    activity
}
```
Repeat Loop

```
repeat (activity)
  until (condition)
```

Outline

- The Concept of an Algorithm
- Algorithm Representation
- Algorithm Discovery
- Iterative Structures
  - Recursive Structures
- Efficiency

Binary Search

A Binary Search algorithm searches a sorted list by examining the middle item. If the item is not found, the appropriate half of the list is searched again (using the Binary Search algorithm). Each time only half of the remaining list is searched, until the item is found.

Binary Search Algorithm

```
procedure Search (List, TargetValue)
if List is empty
  then
    Report that the search failed.
else
  Select the “middle” entry in List to be the TestEntry.
  Execute the block of instructions below that is associated with the appropriate case.
  case 1: TargetValue = TestEntry
    then
      Report that the search succeeded.
  case 2: TargetValue < TestEntry
    (Apply the procedure Search to see if TargetValue is in the portion of the List preceding TestEntry, and report the result of that search.)
  case 3: TargetValue > TestEntry
    (Apply the procedure Search to see if TargetValue is in the portion of List following TestEntry, and report the result of that search.)
end if
```
**Binary Search Example**

Searching for "John"...

<table>
<thead>
<tr>
<th>Original list</th>
<th>First sublist</th>
<th>Second sublist</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alice, Bob, Carol, David, Elma, Fred, George, Harry, Irene, John, Larry, Mary, Nancy, Oliver</td>
<td>Irene, John, Larry, Mary, Nancy, Oliver</td>
<td>Irene, John, Kirby</td>
</tr>
</tbody>
</table>

**Outline**

- The Concept of an Algorithm
- Algorithm Representation
- Algorithm Discovery
- Iterative Structures
- Recursive Structures
- Efficiency

**Algorithm Efficiency**

If an algorithm does not process information efficiently:

- CPU resources are wasted
- Poor response time may result
- Possibly the problem cannot be solved in your lifetime

**Sequential Search**

For the sequential search (previously discussed), the time to find an item in the list is proportional to the size of the list divided by 2. This makes sense because some items (close to the front of the list) are found quickly but other items near the end of the list take more time to find. So on average:

\[ \text{time} \sim \frac{N}{2} \]

where \( N \) is the number of items in the list.
Binary Search

A Binary Search algorithm searches a sorted list by examining the middle item. If the item is not found, the appropriate half of the list is searched again (using the Binary Search algorithm). Each time only half of the remaining list is searched, until the item is found.

The time to perform a binary search is proportional to the logarithm of the size of the list:

time = \log_2(N)

where N is the number of items in the list.

Growth Rates

Importance of Efficiency

Towers of Hanoi

64 Golden Rings on 3 diamond pegs. Priests in India must move the rings from one peg to another, but they can move only one ring at a time and they cannot put a larger ring on top of a smaller one. Legend has it, when all the disks have been moved, the world will end.

This problem has a solution that grows exponentially:

time = 2^n

When will the world end?

Assuming the priest can move one ring per second. It will take:

2^{64} = 18,446,744,073,709,551,616 seconds

or approximately,

585 Billion years.

For perspective, the universe is only 15 Billion years old.