Lecture 18: Introduction to Computer Programming Course - CS1010

DEPARTMENT OF COMPUTER SCIENCE | 11/12/2019

Rensselaer

Goals for Today

- **Problem Solving and Design**
    - Learn about problem solving skills
    - Explore the algorithmic approach for problem solving
    - Learn about algorithm development
    - Become aware of problem solving process

How to solve a problem?

- Programming is a problem-solving activity.
- If you are a good problem solver, you could become a good programmer.
- Problem-solving methods are covered in many subject areas:

  - Business students learn to solve problems with a **systems approach**

  - Engineering and Science students use the **engineering and science methods**

  - Programmers use the **Software Development Method**

Software Development Method

To solve a problem using a computer, you need to:

- Think carefully about the problem & its solution
  - Analyze the problem
  - Specify the problem requirements

- Plan it out beforehand (write an **algorithm**)
- Check it over to see that it addresses the problem
- Modify the solution if necessary
- Use the outlines to write a **program** that you can run on a computer
- Finally, the program is tested to verify that it behaves as intended.

Introduction to Algorithms

- The **algorithm** is the abstract idea of solving a problem.
- An **algorithm** is a step-to-step problem solving process in which a solution is arrived at a finite amount of time.
- The **algorithm** is written in an algorithmic language (or a pseudo code).

Algorithms vs. programs

- When an algorithm is coded using any programming language (e.g. Python), then it is called a **program**.

- The **program** is a set of instructions that can run by the computer.

Characteristics of Algorithms

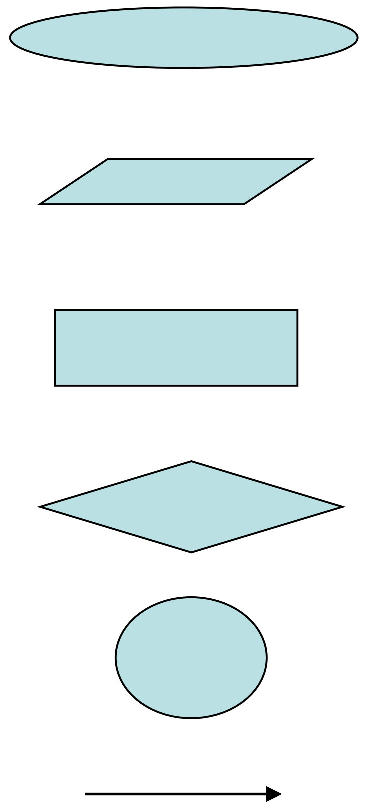
- It should be **accurate**:
  - It shouldn’t be ambiguous (each step in the algorithm should exactly determine the action to be done).
- It should be **effective**:
  - It shouldn’t include a step which is difficult to follow by a person (or the computer) in order to perform it.
- The algorithm should be **finite**:
  - the algorithm has to end up in a point at which the task is complete

Flowchart

- It is another way to display the algorithm.
- It is composed of special geometric symbols connected by lines and contain the instructions.
- You can follow the lines from one symbol to another, executing the instructions inside it.

Flowchart Symbols

- Start / End symbol
- Input/Output symbol
- Processing symbol
- Condition & decision symbol
- Continuation (connection symbol)
- Links



Problem Solving and Programming Strategy

- Programming is a process of problem solving.

- The problem is solved according to the problem domain (e.g. students, money).

- To be a good problem solver and hence a good programmer, you must follow good problem solving technique.

One problem solving technique to solve the problem includes

- **analyzing** the problem & outlining the problem's requirements,
- **designing** steps (writing an algorithm)
- **implementing** the algorithm in a programming language (e.g. Python) and verify that the algorithm works,
- **maintaining** the program by using and modifying it if the problem domain changes.

(a) Problem Analysis

1- Thoroughly understand the problem

2- Understand the problem **specifications.**

3- If the problem is complex, you need to divide it into **sub-problems** and repeat steps (1& 2). That is, you need to analyze each sub-problem and understand its requirements.

(b) Algorithm Design and Testing

- Design an algorithm for the problem.
- If the problem is divided into smaller sub-problems, then design an algorithm for each sub-problem.

How to write an Algorithm?

An algorithm is a sequence of statements to perform some operations. You can write an algorithm with the following layout:

```
ALGORITHM   algorithm_name
        statements of algorithm
END  algorithm_name
```

- The algorithm would consist of at least the following tasks:

  1- **Input** (Read the data)
  2- **Processing** (Perform the computation)
  3- **Output** (Display the results)

***Structured Design*:**

Dividing the problem into smaller sub-problems is called “**structured design**”, “**top-down design**”,

***Structured Programming*:**

In the structured design

- The problem is divided into smaller sub-problems
- Each sub-problem is analyzed
- A solution is obtained to solve the sub-problem
- The solutions of all sub-problems are combined to solve the overall problem.

This process of implementing a structured design is called “**structured programming**”

(c) Coding

- **Coding:**
  - After verifying that the algorithm is correct, you can code it in any high-level programming language
  - The algorithm is now converted into a **program**

(d) Executing the Program

- Compile the program (to check for syntax error)

- Run the program. If the execution doesn’t go well, then reexamine the code, the algorithm, or even the problem analysis.

Advantages of Structured Programming

- Easy to discover errors in a program that is well analyzed and well designed.
- Easy to modify a program that is thoroughly analyzed and carefully deigned.

Example: Search

Suppose we have data showing the number of humpback whales sighted off the coast of British Columbia over the past ten years:

809 834 477 478 307 122 96 102 324 476

We want to know how changes in fishing practices have impacted the whales’ numbers.

Our first question is, which year had the lowest number of sightings during those years?

Example: Search

Although what we did accomplish the task,

It seems a little inefficient to search through the data to find the minimum value and then search again to find that value’s index.

Intuitively speaking, there ought to be a way to find both in just one pass.

The Problem

Now what if we want to find the indices of the two smallest values?

Lists don’t have a method to do this directly, so we will have to write a function ourselves.

Top Down Design

There are at least three ways we could do this, each of which will be subjected to top-down design:

- *Find, remove, find.* Find the index of the minimum, remove that element from the list, and find the index of the new minimum element in the list.
-

Top Down Design

Sort, identify minimums, get indices. Sort the list, get the two smallest numbers, and then find their indices in the original list.

Walk through the list. Examine each value in the list in order, keep track of the two smallest values found so far, and update these values when a new smaller value is found.

Find, Remove, Find

```
def find_two_smallest(L):
1. find the index of the minimum element in L
2. remove that element from the list
3. find the index of the new minimum element in the list
4. return the two indices
```

Find, Remove, Find

```
def find_two_smallest(L):
1. get the minimum element in L
2. find the index of that minimum element
3. remove that element from the list
4. find the index of the new minimum element in the list
5. return the two indices
```

Find, Remove, Find

Since we removed the smallest element, we need to put it back.

Also, when we remove a value, the indices of the following values shift down by one.

So, when smallest was removed, in order to get the indices of the two lowest values in the original list, we need to add 1 to min2

Sort, Identify Minimums, Get Indices

```
def find_two_smallest(L):
1. sort a copy of L
2. get the two smallest numbers
3. find their indices in the original list L
4. return the two indices
```

Walk Through the List

```
def find_two_smallest(L):
1.examine each value in the list in order
2.keep track of the indices of the two smallest values found
so far
    3.update these values when a new smaller value is found
4.return the two indices
```

Walk Through the List

```
def find_two_smallest(L):
set min1 and min2 to the indices of the smallest and next-smallest values at the beginning of L
examine each value in the list in order
update these values when a new smaller value is found
return the two indices
```

Walk Through the List

```
# set min1 and min2 to the indices of the smallest and next-
smallest
# values at the beginning of L
if L[0] < L[1]:
   smallest, next_smallest = 0, 1
else:
smallest, next_smallest = 1, 0
examine each value in the list in order
     update these values when a new smaller value is found
return the two indices
```

Walk Through the List

```
def find_two_smallest(L):
   if L[0] < L[1]:
      smallest, next_smallest = 0, 1
   else:
      smallest, next_smallest = 1, 0
# examine each value in the list in order
for i in range(2, len(values)):
    update min1 and/or min2 when a new smaller value is  found
return the two indices
```

The Loop Part

```
# examine each value in the list in order
for i in range(2, len(L)):
L[i] is larger than both min1 and min2, smaller than both, or in between.
if L[i] is larger than both min1 and min2, skip it
if L[i] is smaller than min1 and min2, update them both
if L[i] is in between, update min2
return (min1, min2)
```

Timing

Profiling a program means measuring how long it takes to run and how much memory it uses.

These two measures—space and time— are fundamental to the theoretical study of algorithms.

They are also pretty important from a pragmatic point of view.

Fast programs are more useful than slow ones, and programs that need more memory than what your computer has aren’t particularly useful at all

Timing

The 3 functions developed were used on a list on 1,400 monthly readings of air pressure in Darwin, Australia, from 1882 to 1998. The execution times were as follows:

Algorithm	Running Time (ms)
Find, remove, find	1.117
Sort, identify, index	2.128
Walk through the list	1.472

Compare

No human being can notice the difference between one and two milliseconds;

if this code never has to process lists with more than 1,400 values, we would be justified in choosing an implementation based on simplicity or clarity, rather than speed.

But what if we wanted to process millions of values?

Find-remove-find outperforms the other two algorithms on 1,400 values, but how much does that tell us about how they will perform on data sets 1,000 times larger?

Summary

The most effective way to design algorithms is to use top-down design, in which goals are broken down into sub-goals until the steps are small enough to be translated directly into a programming language.

The performance of a program can be characterized by how much time and memory it uses. This can be determined experimentally by profiling its execution.

Next Class

Exception Handling