I. Software Development (Chap. 1 — read)

5 phases of software life cycle

A. Problem Analysis and Specification (§1.1)

— Easy in courses, not always in real world

CPSC 185 - Assignment 4

Due : Wednesday, March 11

One method of calculating depreciation is the sum-of-the-years digits method. It is illustrated by the following example.
Suppose that $15,000 is to be depreciated over a five-year period. We first calculate the "sum-of-the-years digits," 1 + 2 + 3 + 4 + 5 = 15. Then 5/15 of $15,000 ($5,000) is depreciated the first year, 4/15 of $15,000 ($4,000) is depreciated the second year, 3/15 the third year, and so on.

Write a program that reads the amount to be depreciated and the number of years over which it is to be depreciated. Then for each year from 1 through the specified number of years, print the year number and the amount of depreciation for that year under appropriate headings. Execute the program with the following data: $15,000 for 3 years; $7,000 for 10 years; $500 for 20 years; $100 for 1year.

To: Bob Byte, Director of Computer Center

From: Chuck Cash, V.P. of Scholarships and Financial Aid

Date: Wednesday, March 11

Because of new government regula-tions,we must keep more accurate records of all students currently receiv-ing financial aid and submit regular reports to FFAO (Federal Financial Aid Office). Could we get the computer to do this for us?

CC

— Statement of specifications becomes:
- the formal statement of the problem's requirements
- the major reference document
- a benchmark used to evaluate the final system

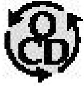
The program should display on the screen a prompt for an amount to be depreciated and the number of years over which it is to be depreciated. It should then read these two values from the keyboard. Once it has the amount and the number of years, it should compute the sum of the integers from 1, 2, . . . , up to the number of years. It should then display on the screen a table with appropriate headings that shows the year number and the depreciation for that year, for the specified number of years.

Sometimes stated precisely using a *formal method*

B. Design (§1.2)

Programs, libraries, classes:

In CS courses	In the real world
small — a few hundred lines of code	large systems — thousands of lines of code
simple, straightforward	complex



Object-centered design:

1. Identify the **objects** in the problem's specification and their types.
2. Identify the **operations** needed to solve the problem.
3. Arrange the operations in a sequence of steps, called an **algorithm**, which, when applied to the objects, will solve the problem.

Data types:

- Simple
- Structured — **data structures**

Algorithms

- Different ones may work, but may not be equally efficient (pp. 7-8)
  - O(n) — grows linearly with size (n) of the input
  - O(1) — is constant — independent of size of input

  More later about measuring efficiency
- Can't separate data structures and algorithms
  Algorithms + Data Structures = Programs
- Properties of instructions (p. 9)
  - — Definite and unambiguous
  - — Simple
  - — Finiteness
- Usually written in pseudocode
- ~~Can be unstructured~~
  Should be structured (pp. 10-12)

ALGORITHM (UNSTRUCTURED VERSION)

/* Algorithm to read and count several triples of distinct numbers and print the largest number in each triple. */

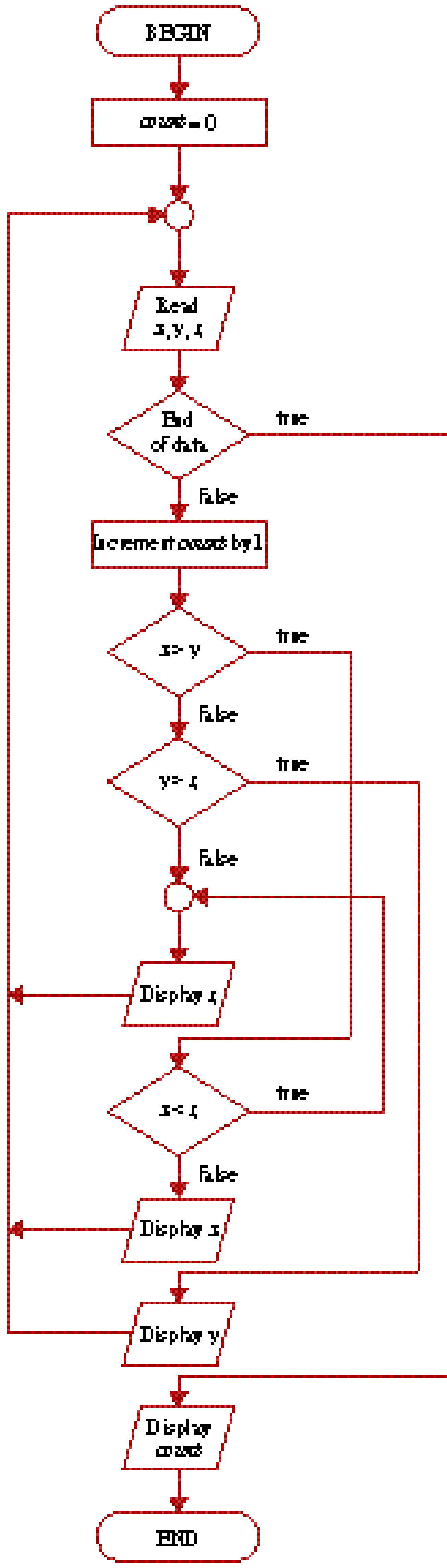
1. Initialize *count* to 0.
2. Read a triple x, y, z.
3. If x is the end-of-data flag then go to step 14.
4. Increment *count* by 1.
5. If $x > y$ then go to step 9.
6. If $y > z$ then go to step 12.
7. Display z.
8. Go to step 2.
9. If $x < z$ then go to step 7.
10. Display x.
11. Go to step 2.
12. Display y.
13. Go to step 2.
14. Display *count.*

Note the *spaghetti logic*!

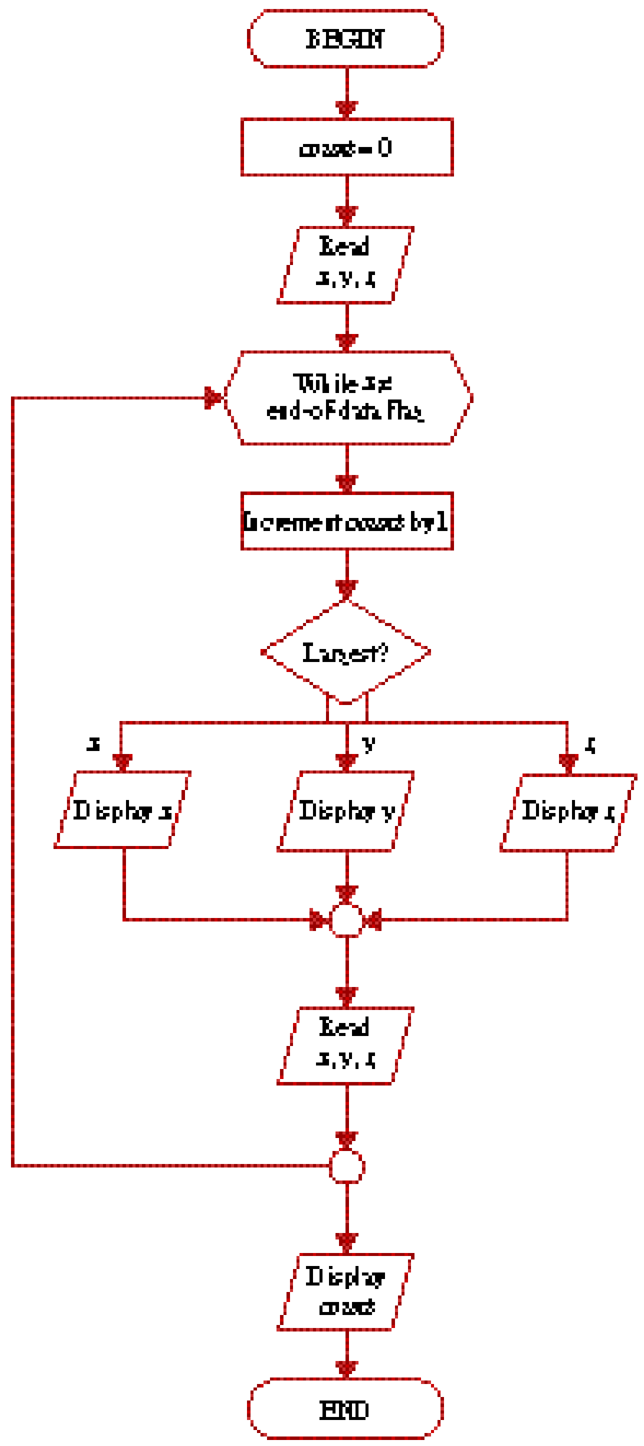
ALGORITHM (STRUCTURED VERSION)

/* Algorithm to read and count several triples of distinct numbers and print the largest number in each triple. */

1. Initialize *count* to 0.
2. Read the first triple of numbers x, y, z.
3. While x is not the end-of-data-flag do the following:
   a. Increment *count* by 1.
   b. If $x > y$ and $x > z$ then
         Display x.
      Else if $y > x$ and $y > z$ then
         Display y.
      Else
         Display z.
   c. Read the next triple x, y, z.
4. Display *count.*



(a)



(b)

C. Coding (§1.3): Implementing the design plan in some programming language.

Integration: Combining program units into a complete software system.

— What language?
— Programs must be correct, readable, and understandable
(therefore, must be well-structured, documented,
written in good style — read guidelines on pp. 15-18)
Why? see page 15

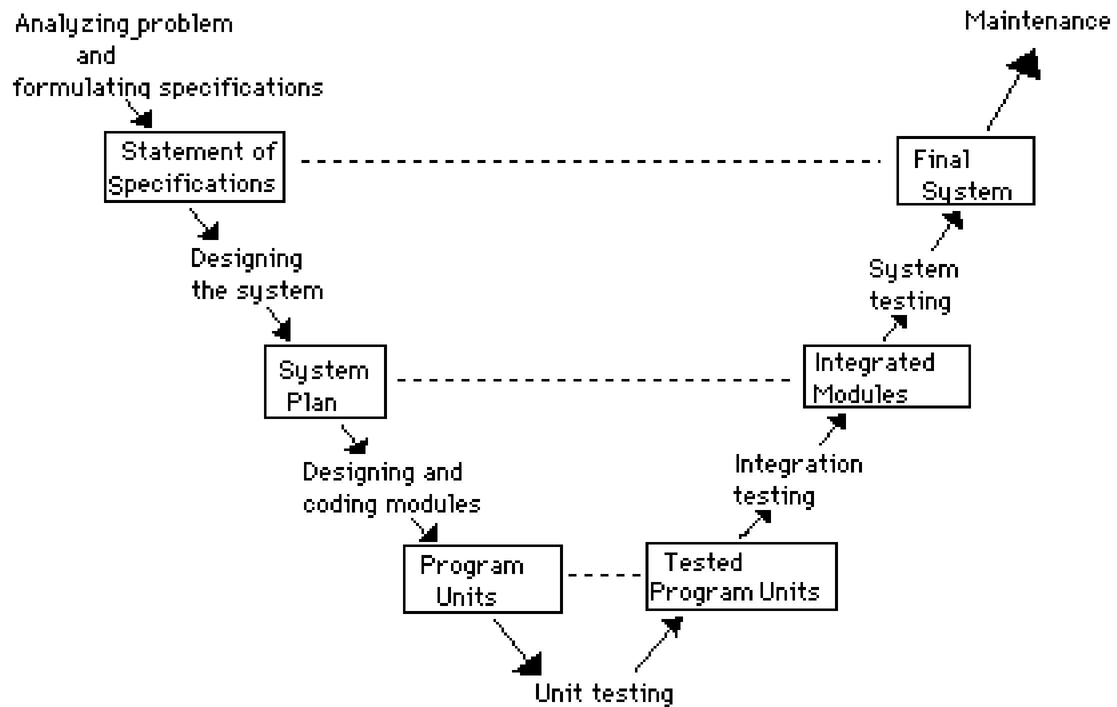
D. Testing, Execution, and Debugging

Validation: checking that the documents, program modules, etc. produced match the customer's requirements.
Verification: checking that products are correct, complete, consistent with each other and with those of the preceding phases.
Validation: "Are we building the right product?"
Verification: "Are we building the product right?"

1. Errors may occur in any of the phases:
   — Specifications don't accurately reflect given information or the user's needs/requests
   — Logic errors in algorithms
   — Incorrect coding or integration

2. Different kinds of tests required to detect them:
   Unit tests: Each individual program unit works?
   Integration tests: Units combined correctly?
   System tests: Overall system works correctly?

The "V" Life Cycle Model.



Unit testing:
— probably the most rigourous and time-consuming
— surely the most fundamental and important

3. Kinds of errors
— syntax
— linking
— run-time
— logical

4. Kinds of tests:
— **Black box** or **functional test** : Outputs produced for various inputs are checked for correctness without considering the structure of the module itself. (Program unit is viewed as a black box that accepts inputs and produces outputs, but the inner workings of the box are not visible.)

—**White box** or **structural test**: Performance is tested by examining its internal structure. Test data is carefully selected so that the various parts of the program unit are exercised.

5. Example: Binary search (pp. 19-23)

```
    /* INCORRECT FUNCTION ----------------------------------
-
   BinarySearch() performs a binary search of a for item.

   Receive:   item and an array a having n items, arranged
              in ascending order
   Pass back: found and mid, where found is true
and
              mid is the position of item if the
search
              is successful; otherwise found is
false.
-----------------------------------------------------
--*/

void BinarySearch(NumberArray a, int n,  ElementType item,
                  bool & found, int & mid)
{
  int first = 0,     // first and last positions in sublist
      last = n - 1;  // currently being searched *)
  found = false;
  while (first <= last && !found)
  {
    mid = (first  + last ) / 2;
    if item < a[mid]
      last = mid;
    else if item  > a[mid]
      first = mid;
    else
      found = true
  }
}
```

Black box test: Use n = 7 and array a of integers:

a[0] = 45
a[1] = 64
a[2] = 68
a[3] = 77
a[4] = 84
a[5] = 90
a[6] = 96

Test with item = 77 returns found = true, mid = 4
Test with item = 90 returns found = true, mid = 6
Test with item = 64 returns found = true, mid = 2
Test with item = 76 returns found = false

But, . . ., **must consider special cases**:

e.g., searching at the ends of the list: item 45, item 96

item = 45: found = true and mid = 1 as it should.
item = 96: doesn't terminate; must "break" program.

White-box test would also find an error:

e.g., Use `item` < 45 to test a path in which the first condition `item < a[mid]`
is always true so first alternative `last = mid;` is always selected.
Use `item` > 96 to test a path in which the second condition `item > a[mid]`
is always true so second alternative `first = mid;` is always selected.

6. Techniques to locate error:
— Debugger (Project 1)

— Debug statements (p. 21): e.g.,

```
cerr << "DEBUG:  At top of while loop in BinarySearch()\n"
     << "first = " << first << ", last = " << last
     << ", mid = " << mid << endl;
```

Output:

```
DEBUG:  At top of while loop in BinarySearch()
first = 0, last = 6, mid = 3
DEBUG:  At top of while loop in BinarySearch()
first = 3, last = 6, mid = 4
DEBUG:  At top of while loop in BinarySearch()
first = 4, last = 6, mid = 5
DEBUG:  At top of while loop in BinarySearch()
first = 5, last = 6, mid = 5
DEBUG:  At top of while loop in BinarySearch()
first = 5, last = 6, mid = 5
DEBUG:  At top of while loop in BinarySearch()
first = 5, last = 6, mid = 5
        .
        .
        .
```

— Trace tables (p. 22 & Lab 1A)

— *Quick-and-dirty patches are bad!* (p. 23)

E. Maintenance — pp. 23-24

— Large % of computer center budgets
— Large % of programmer's time

Why? Poor structure, poor documentation, poor style.