BLM5207
Computer Organization

Prof. Dr. Nizamettin AYDIN

naydin@yildiz.edu.tr

http://www3.yildiz.edu.tr/~naydin

Programming

1

1

Solving Problems using a Computer

- Methodologies for creating computer programs that perform a desired function.
  - Problem Solving
    - How do we figure out what to tell the computer to do?
    - Convert problem statement into algorithm, using stepwise refinement.
    - Convert algorithm into machine instructions.
  - Debugging
    - How do we figure out why it didn't work?
    - Examining registers and memory, setting breakpoints, etc.
- Time spent on the first can reduce time spent on the second!

2

2

Stepwise Refinement

- Also known as systematic decomposition.
- Start with problem statement:
  - "We wish to count the number of occurrences of a character in a file. The character in question is to be input from the keyboard; the result is to be displayed on the monitor."
- Decompose task into a few simpler subtasks.
- Decompose each subtask into smaller subtasks, and these into even smaller subtasks, etc.... until you get to the machine instruction level.

3

3

Problem Statement

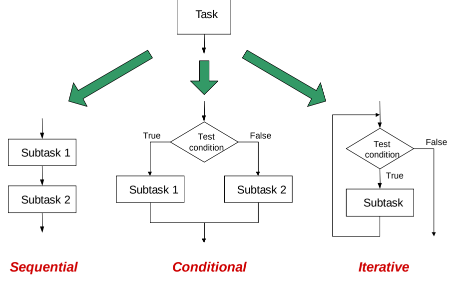
- Because problem statements are written in English, they are sometimes ambiguous and/or incomplete.
  - Where is "file" located?
  - How big is it, or how do I know when I've reached the end?
  - How should final count be printed? A decimal number?
  - If the character is a letter, should I count both upper-case and lower-case occurrences?
- How do you resolve these issues?
  - Ask the person who wants the problem solved, or
  - Make a decision and document it.

4

4

Three Basic Constructs

- There are three basic ways to decompose a task:

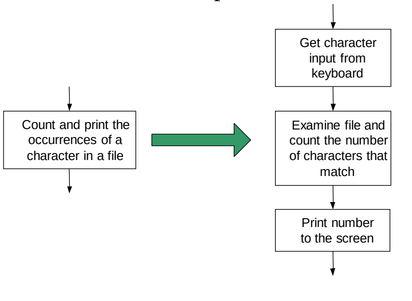


5

5

Sequential

- Do Subtask 1 to completion, then do Subtask 2 to completion, etc.

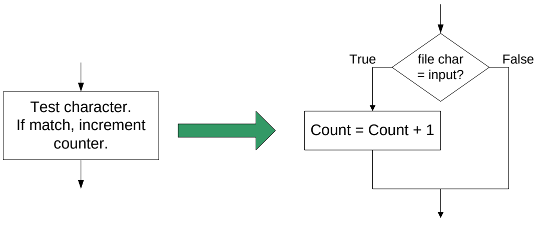


6

6

Conditional

- If condition is true, do Subtask 1;
  else, do Subtask 2.

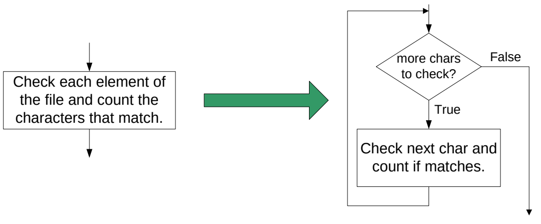


7

7

Iterative

- Do Subtask over and over,
  as long as the test condition is true.



8

8

Problem Solving Skills

- Learn to convert problem statement
  into step-by-step description of subtasks.
  - Like a puzzle, or a "word problem" from grammar school math.
    - What is the starting state of the system?
    - What is the desired ending state?
    - How do we move from one state to another?
  - Recognize English words that correlate to three basic constructs:
    - "do A then do B" ⇒ sequential
    - "if G, then do H" ⇒ conditional
    - "for each X, do Y" ⇒ iterative
    - "do Z until W" ⇒ iterative

9

9

Debugging

- You've written your program and it doesn't work.
  - Now what?
- What do you do when you're lost in a city?
  - Drive around randomly and hope you find it?
  - Return to a known point and look at a map?
- In debugging, the equivalent to looking at a map is tracing your program.
  - Examine the sequence of instructions being executed.
  - Keep track of results being produced.
  - Compare result from each instruction to the expected result.

10

10

Debugging Operations

- Any debugging environment should provide means to:
  - Display values in memory and registers.
  - Deposit values in memory and registers.
  - Execute instruction sequence in a program.
  - Stop execution when desired.
- Different programming levels offer different tools.
  - High-level languages (C, Java, ...)
    usually have source-code debugging tools.
  - For debugging at the machine instruction level:
    - simulators
    - operating system "monitor" tools
    - in-circuit emulators (ICE)
      - plug-in hardware replacements that give instruction-level control

11

11

Types of Errors

- Syntax Errors
  - You made a typing error that resulted in an illegal operation.
  - Not usually an issue with machine language, because almost any bit pattern corresponds to some legal instruction.
  - In high-level languages, these are often caught during the translation from language to machine code.
- Logic Errors
  - Your program is legal, but wrong, so the results don't match the problem statement.
  - Trace the program to see what's really happening and determine how to get the proper behavior.
- Data Errors
  - Input data is different than what you expected.
  - Test the program with a wide variety of inputs.

12

12

Tracing the Program

- Execute the program one piece at a time, examining register and memory to see results at each step.
- Single-Stepping
  - Execute one instruction at a time.
  - Tedious, but useful to help you verify each step of your program.
- Breakpoints
  - Tell the simulator to stop executing when it reaches a specific instruction.
  - Check overall results at specific points in the program.
    - Lets you quickly execute sequences to get a high-level overview of the execution behavior.
    - Quickly execute sequences that you believe are correct.
- Watchpoints
  - Tell the simulator to stop when a register or memory location changes or when it equals a specific value.
  - Useful when you don't know where or when a value is changed.

13

13

Debugging: Lessons Learned

- Trace program to see what's going on.
  - Breakpoints, single-stepping
- When tracing, make sure to notice what's *really* happening, not what you think *should* happen.
- Test your program using a variety of input data.
  - Be sure to test extreme cases (all ones, no ones, ...).

14

14

The Concept of an Algorithm

- Some Algorithms:
  - Converting from one base to another
  - Correcting errors in data
  - Compression

    ⋮
  - …
- Many researchers believe that every activity of the human mind is the result of an algorithm

15

15

Formal Definition of Algorithm

- An algorithm is an ordered set of unambiguous, executable steps that defines a terminating process
  - The steps of an algorithm can be sequenced in different ways
  - Linear (1, 2, 3, …)
  - Parallel (multiple processors)
  - Cause and Effect (circuits)
- A Terminating Process
  - Culminates with a result
  - Can include systems that run continuously
    - Hospital systems
    - Long Division Algorithm
- A Non-terminating Process
  - Does not produce an answer
    - "Non-deterministic Algorithms"

16

16

The Abstract Nature of Algorithms

- There is a difference between an algorithm and its representation.
  - Analogy:
    - difference between a story and a book
- A Program
  - a representation of an algorithm.
- A Process
  - the activity of executing an algorithm.

17

17

Algorithm Representation

- Is done formally with well-defined Primitives
  - A collection of primitives constitutes a programming language.
- Is done informally with Pseudocode
  - Pseudocode is between natural language and a programming language.

18

18

Designing a Pseudocode Language

- Choose a common programming language
- Loosen some of the syntax rules
- Allow for some natural language
- Use consistent, concise notation
- We will use a Python-like Pseudocode

19

19

Pseudocode Primitives

- Assignment

  name = *expression*

  – example

  RemainingFunds = CheckingBalance + SavingsBalance

- Conditional selection

  if (*condition*):
  activity

  – example

  if (sales have decreased):
  lower the price by 5%

20

20

Pseudocode Primitives

- Conditional selection

```
if (condition):
        activity
else:
        activity
```

– example

```
if (year is leap year):
        daily total = total / 366
else:
        daily total = total / 365
```

21

21

Pseudocode Primitives

- Repeated execution

  while (*condition*):
  body

  – example

  while (tickets remain to be sold):
  sell a ticket

- Indentation shows **nested** conditions

```
if (not raining):
        if (temperature == hot):
                go swimming
        else:
                play golf
else:
watch television
```

22

22

Pseudocode Primitives

- Define a function

```
def name():
```

– example

```
def ProcessLoan():
```

- Executing a function

```
if (. . .):
    ProcessLoan()
else:
    RejectApplication()
```

23

23

The procedure Greetings in pseudocode

```
def Greetings():
    Count = 3
    while (Count > 0):
        print('Hello')
        Count = Count - 1
```

24

24

Pseudocode Primitives

- Using parameters

```
def Sort(List):
    .
    .
    .
```

- Executing Sort on different lists

```
Sort(the membership list)
Sort(the wedding guest list)
```

25

25

Algorithm Discovery

- The first step in developing a program
- More of an art than a skill
- A challenging task
- Polya's Problem Solving Steps
  1. Understand the problem.
  2. Devise a plan for solving the problem.
  3. Carry out the plan.
  4. Evaluate the solution for accuracy and its potential as a tool for solving other problems.

26

26

Getting a Foot in the Door

- Try working the problem backwards
- Solve an easier related problem
  - Relax some of the problem constraints
  - Solve pieces of the problem first
    - (bottom up methodology)
- Stepwise refinement:
  - Divide the problem into smaller problems
    - (top-down methodology)

27

27

Ages of Children Problem

- Person A is charged with the task of determining the ages of B's three children.
  - B tells A that the product of the children's ages is 36.
  - A replies that another clue is required.
  - B tells A the sum of the children's ages.
  - A replies that another clue is needed.
  - B tells desired triple must be one whose sum appears at least twice in the table
  - A replies that another clue is needed.
  - B tells A that the oldest child plays the piano.
  - A tells B the ages of the three children.
- How old are the three children?

a. Triples whose product is 36

(1,1,36)	(1,6,6)
(1,2,18)	(2,2,9)
(1,3,12)	(2,3,6)
(1,4,9)	(3,3,4)

b. Sums of triples from part (a)

1 + 1 + 36 = 38	1 + 6 + 6 = 13
1 + 2 + 18 = 21	2 + 2 + 9 = 13
1 + 3 + 12 = 16	2 + 3 + 6 = 11
1 + 4 + 9 = 14	3 + 3 + 4 = 10

28

28

Iterative Structures

- A collection of instructions repeated in a looping manner
- Examples include:
  - Sequential Search Algorithm
    - The sequential search algorithm in pseudocode

```
def Search (List, TargetValue):
    if (List is empty):
        Declare search a failure
    else:
        Select the first entry in List to be TestEntry
        while (TargetValue > TestEntry and entries remain):
            Select the next entry in List as TestEntry
        if (TargetValue == TestEntry):
            Declare search a success
        else:
            Declare search a failure
```

29

29

Iterative Structures

- Insertion Sort Algorithm
  - The insertion sort algorithm expressed in pseudocode

```
def Sort(List):
    N = 2
    while (N <= length of List):
        Pivot = Nth entry in List
        Remove Nth entry leaving a hole in List
        while (there is an Entry above the
                    hole and Entry > Pivot):
            Move Entry down into the hole leaving
            a hole in the list above the Entry
        Move Pivot into the hole
        N = N + 1
```

30

30

Components of repetitive control

- Initialize:
  - Establish an initial state that will be modified toward the termination condition
- Test:
  - Compare the current state to the termination condition and terminate the repetition if equal
- Modify:
  - Change the state in such a way that it moves toward the termination condition

31

Iterative Structures

- Pretest loop:

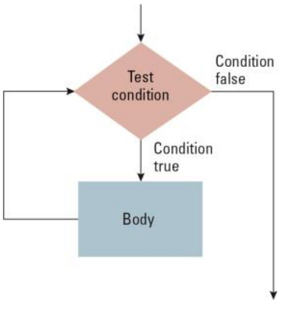
```
while (condition):
    body
```

- Posttest loop:

```
repeat:
    body
until(condition)
```

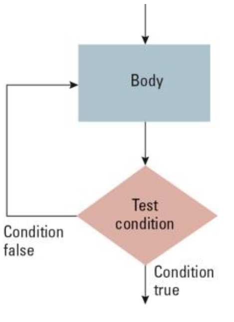
32

The while loop structure



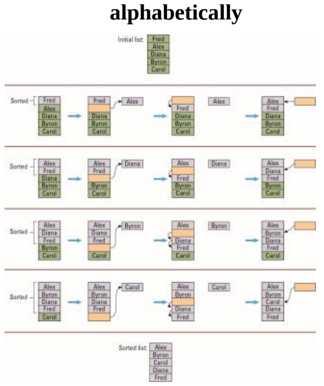
33

The repeat loop structure



34

Sorting the list Fred, Alex, Diana, Byron, and Carol alphabetically



35

The insertion sort algorithm expressed in pseudocode

```
def Sort(List):
    N = 2
    while (N <= length of List):
        Pivot = Nth entry in List
        Remove Nth entry leaving a hole in List
        while (there is an Entry above the
                   hole and Entry > Pivot):
            Move Entry down into the hole leaving
             a hole in the list above the Entry
        Move Pivot into the hole
        N = N + 1
```

36

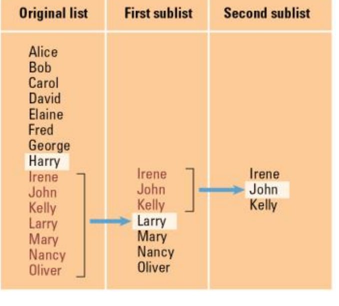
37

Recursive Structures

- Repeating the set of instructions as a subtask of itself.
- Multiple activations of the procedure are formed, all but one of which are waiting for other activations to complete.
- Example:
  – The Binary Search Algorithm

38

Applying our strategy to search a list for the entry John



39

A first draft of the binary search technique

```
if (List is empty):
    Report that the search failed
else:
    TestEntry = middle entry in the List
    if (TargetValue == TestEntry):
        Report that the search succeeded
    if (TargetValue < TestEntry):
        Search the portion of List preceding TestEntry for
        TargetValue, and report the result of that search
    if (TargetValue > TestEntry):
        Search the portion of List following TestEntry for
        TargetValue, and report the result of that search
```

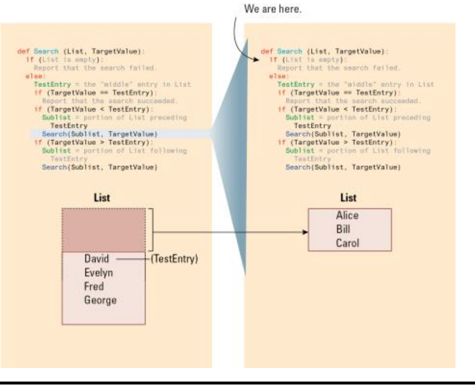
40

The binary search algorithm in pseudocode

```
def Search(List, TargetValue):
    if (List is empty):
        Report that the search failed
    else:
        TestEntry = middle entry in the List
        if (TargetValue == TestEntry):
            Report that the search succeeded
        if (TargetValue < TestEntry):
            Sublist = portion of List preceding TestEntry
            Search(Sublist, TargetValue)
        if (TargetValue < TestEntry):
            Sublist = portion of List following TestEntry
            Search(Sublist, TargetValue)
```

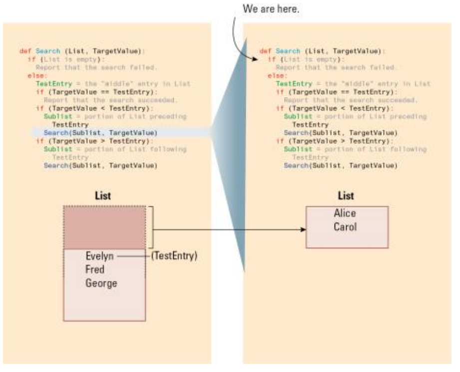
41

Recursively Searching



42

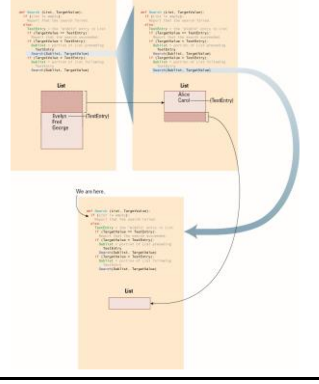
Second Recursive Search



43

43

Second Recursive Search, Second Snapshot



44

44

45

45

Recursive Control

- Requires initialization, modification, and a test for termination (base case)
- Provides the illusion of multiple copies of the function, created dynamically in a telescoping manner
- Only one copy is actually running at a given time, the others are waiting

46

46

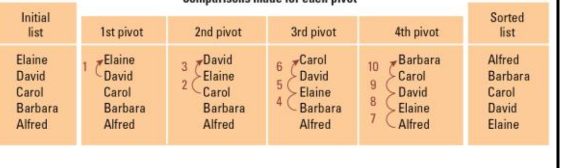
Correctness and Efficiency

- The choice between efficient and inefficient algorithms can make the difference between a practical solution and an impractical one
- The correctness of an algorithm is determined by reasoning formally about the algorithm, not by testing its implementation
- The efficiency is measured as number of instructions executed
  - Uses big theta notation:
    - Example: Insertion sort is in $\Theta(n^2)$
  - Incorporates best, worst, and average case analysis

47

47

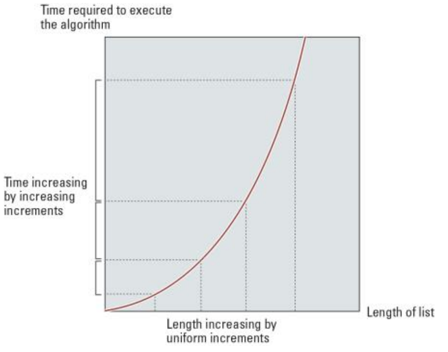
Applying the insertion sort in a worst-case situation



48

48

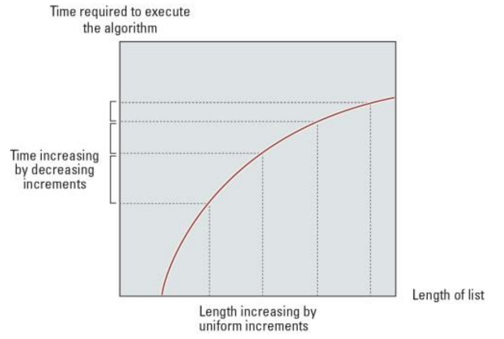
Graph of the worst-case analysis of the insertion sort algorithm



49

49

Graph of the worst-case analysis of the binary search algorithm



50

50

Software Verification

- Proof of correctness (with formal logic)
  - Assertions
    - Preconditions
    - Loop invariants
- Testing is more commonly used to verify software
- Testing only proves that the program is correct for the test cases used

51

51

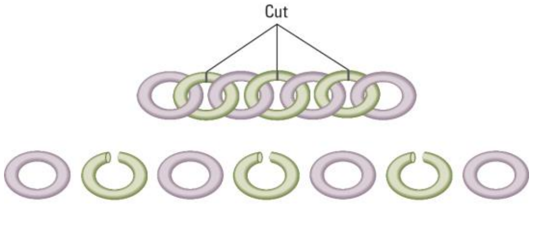
Chain Separating Problem

- A traveler has a gold chain of seven links.
- He must stay at an isolated hotel for seven nights.
- The rent each night consists of one link from the chain.
- What is the fewest number of links that must be cut so that the traveler can pay the hotel one link of the chain each morning without paying for lodging in advance?

52

52

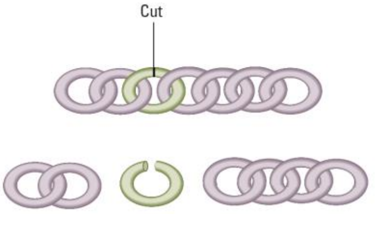
Separating the chain using only three cuts



53

53

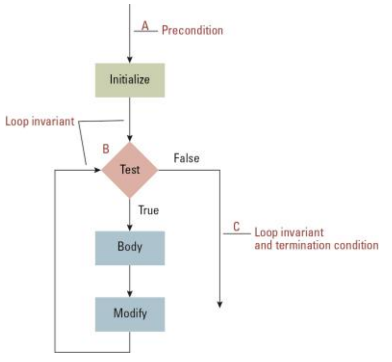
Solving the problem with only one cut



54

54

The assertions associated with a typical while structure



55

55

56

56