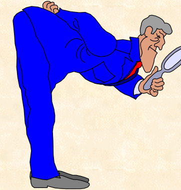
Problem Solving Techniques & Algorithms - Introduction

- Given: Clear statement of the problem, necessary input and necessary output



  - Not the case in real life situations
  - After analyzing the problem, an algorithm is needed
  - Most of the experience with algorithms is in the context of following them
  - In the problem-solving phase of computer programming, you will be designing algorithms
  - Strategies you should use to solve problems in order to apply them to programming problems

Ask Questions

- What do I have to work with => what are my data?
- What do the data look like?
- How much data is there?
- How will I know when I have processed all the data?
- How should my output look like?
- How many times is the process going to be repeated?
- What special error conditions might come up?
- *Example:* Spelling Checker

Example : University (fiction)

- Year 0: Keep a track on who study and who works here!
  - List of all students at University
  - List of all staff members at University
- Year 5: "I want to hire students to do Master thesis at our department. Tell me who has enough credits to start work!"
- Year 7: "I want to tell government how many students are working here, so they will give us more money to hire more students to do research!"
- Year 10: "Our company would like to hire some of your students that just graduated. Could you give us list of students that made thesis about XY?"
- Over a time, demands are changing, new algorithms are implemented to answer requests. Data are the same.

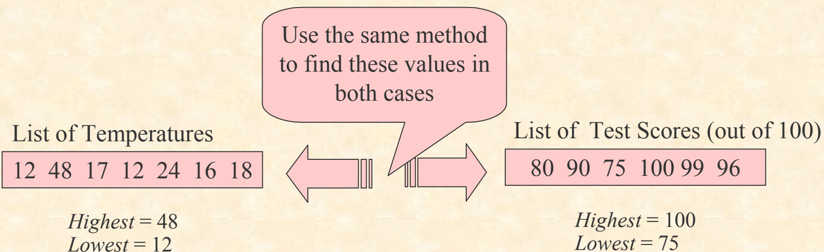
Look for Things That Are Familiar

- If a solution exists, use it!

  Solving same or similar problems before

  A good programmer immediately recognizes a subtask he or she has solved before and plugs in the solution (software reuse)

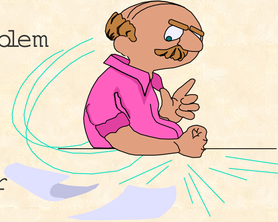
  Example: finding the daily high and low temperatures is the same problem as finding the highest and lowest grades on a test: the problem of finding the largest and smallest values in a set of numbers



Highest = 48
Lowest = 12

Highest = 100
Lowest = 75

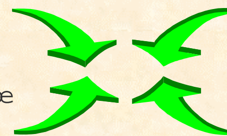
Solve by Analogy



- Often a problem reminds you of a similar problem you have seen before
- Draw an analogy between the two problems
- Analogy is looking for things that are familiar
- Don't worry if your analogy does not match perfectly, it is a place to start
- Details concerning different things between the two problems are dealt with one at a time
- Don't limit yourself to computer-oriented problems
- The best programmers are people who have broad experience solving all kinds of problems

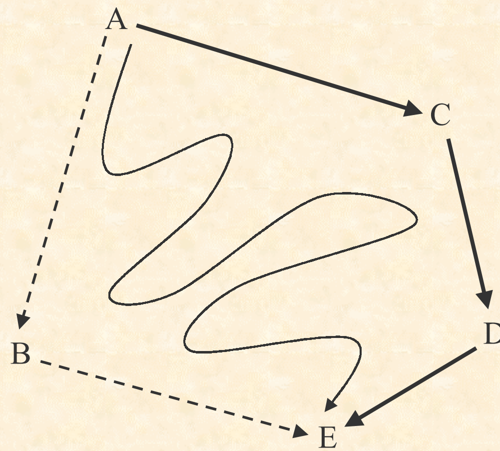
  Example: Library organization and a database search engine

Means-Ends Analysis



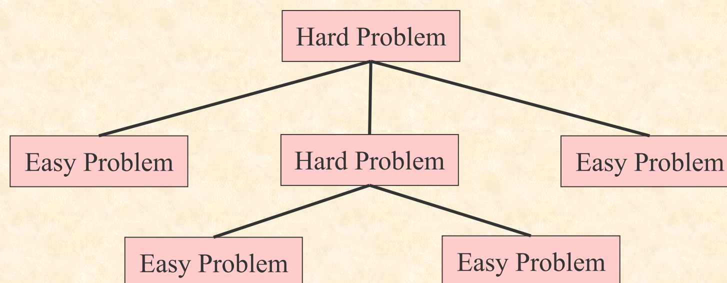
- The beginning state and the ending state are given
- The problem is to define a set of actions that can be used to get from one to the other
- Narrowing the set of actions, then working out the details
  Example: Travelling between two cities
- The overall strategy is to define the ends and then analyze the means of getting between them
- The process translates easily to computer programming
- Writing down the input and the desired output
- Choosing a sequence of computer actions that can transform the data into the results

Means-Ends Analysis (cont'd)



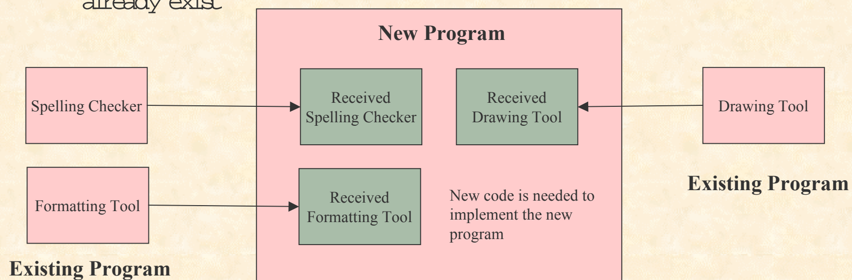
Divide and Conquer

- Breaking up large problems into smaller units that are easier to handle
- Example: Cleaning the house, implementing a word processor
- Top-down methodology and object-oriented methodology are based on the principle of divide and conquer



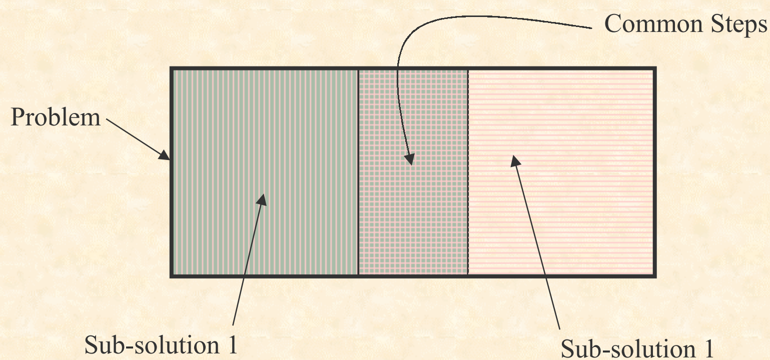
The Building-Block Approach

- Finding existing solutions for smaller pieces of the program
- It may be possible to put some of these solutions together end to end to solve most of the big problem
- A combination of look-for-familiar-things and divide-and-conquer approaches
- Dividing the big problem into smaller problems for which solutions already exist



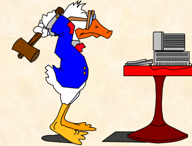
Merging Solutions

- Merging solutions on a step-by-step basis
- *Example:* Computing the average of a list of values, we must both sum and count the values
- Saving steps in merging solutions instead of joining them end to end



Mental Blocks: The Fear of Starting

- Not knowing where to begin, a big problem seems to be overwhelming and hard to tackle
- Writing down the problem on paper in own words in order to understand it
- Paraphrasing the problem helps in identifying the subparts
  - pieces analogous to other problems
  - pieces with existing solution
  - need more information
  - etc.



- Most mental blocks are caused by not really understanding the problem

Example: Let X be a person and A(X) denotes sleeping

Not (there exists X) such that not A(X)

is equivalent to

For any X, A(X) is always true

Algorithmic Problem Solving

- Coming up with a step-by-step procedure for solving a particular problem is not always cut-and-dried
- A trial-and-error process requires several attempts and refinements
- Test each attempt. If it really solves the problem, it is fine. Otherwise, try again

- A possible good solution is to combine the attempts
- Use a combination of all the techniques we have described to solve any nontrivial problem
- If you remember that the computer can only do certain things. You won't design an algorithm that is difficult or impossible to code

Algorithms: Importance of Design

- Once the specification of a problem has been given, a plan for developing a program or a system of modules, libraries, and programs that solve the problem must be designed (algorithm)

  An **algorithm** is a step-by-step set of instructions for solving a problem

- Various design methodologies have been developed over the years. One of the most promising is **object-centered design (OCD)** which leads to **object-oriented programming (OOP):**

- Object-centered design involves several stages:
  - Identify the objects in the problem's specification and their types
  - Identify the operations needed to solve the problem
  - Arrange the operations in a sequence of steps, called an algorithm, which, when applied to the objects, will solve the problem

Object-Centered Design: Case Study

- Problem Specification:

  Design a computer program that reads an integer *n* from the keyboard, then computes and displays the sum 1 + 2 + … + *n* to the screen.

  If integer *n* is less than 0, the program displays “Error in input” to the screen.

Object-Centered Design: Objects

- Once the specification is formulated, we need to identify the objects.

  One approach is to go through the description and identify all the nouns:

Description	Type	Name
Integer	*int*	Number
Keyboard	*istream*	*cin*
Sum	*int*	Total
Screen	*ostream*	*cout*

Object-Centered Design: Operations

- Data is useless if it cannot be processed, we need to identify which operations are needed to process the data and solve the problem.

  One approach is to go through the description and focus on the actions:

Description	Name
Prompt the user	<<
Read an int (*Number*)	>>
Display *Total*	<<
Calculate *Total*	??
Display "Error in input"	<<

Case Study: Algorithm 1

1. Prompt the user for input

2. Accept an integer for *Number*

3. If Number < 0 then

Display “Error in input”

Else

a. Initialize *Total* to 0

b. For each integer *k* in the range 1 to *Number*

Assign *Total* + *k* to *Total*

c. Display the value of *Total*

Case Study: Algorithm 1 (example)

1. “Input number:” 5

2. Number = 5

3. 5 > 0

Total = 0

Total = *Total* + *1*

Total = *Total* + *2*

Total = *Total* + *3*

Total = *Total* + *4*

Total = *Total* + *Number*

“Total is 15”

Case Study: Algorithm 2

1. Prompt the user for input

2. Accept an integer for *Number*

3. If Number < 0 then

Display “Error in input”

Else

a. Assign *Total* to $\frac{Number * (Number + 1)}{2}$

b. Display the value of *Total*

Case Study: Algorithm 2 (example)

1. Prompt the user for input

“Input number:” 5

2. Accept an integer for *Number*

$Number = 5$

3. If Number < 0 then

5 > 0

Else

a. Assign *Total* to $\frac{Number * (Number + 1)}{2}$

$\frac{5 * (5 + 1)}{2}$

b. Display the value of *Total*

“Total is 15”

More on Algorithms

- The world **algorithm** is derived from the name of the mathematician, Abu Ja'far Mohammed ibn Musa al **Khowarizmi**
- Algorithms must be:
  - **definite** and **unambiguous** so that it is clear what each instruction is meant to accomplish
  - **simple** enough that they can be carried out by a computer
  - **finite**; that is, the algorithm must terminate after a finite number of operations
- Algorithms are usually written in **pseudocode**, a pseudo-programming language that is the mixture of natural language and symbols, terms and other features used in high level programming languages.

Structured versus Unstructured Algorithms

- The analysis and verification of algorithms and of the programs that implement them are much easier if they are well structured, that is they are designed using three basic control structures:
  - **Sequence:** Steps are performed in a strictly sequential manner
  - **Selection:** One of the several alternative actions is selected and executed
  - **Repetition:** One or more steps is performed repeatedly
- These three control mechanisms are individually quite simple, but together they are sufficiently powerful that any algorithm can be constructed using them

Case Study: Algorithm 1 (Unstructured Version)

1. Prompt the user for input

2. Accept an integer for *Number*

3. If *Number* < 0 then
    Display “Error in input”
  Else
    a. Initialize *Total* to 0
    b. Initialize k to 0
    c. If k > Number then go to **step g**
    d. Assign *Total* + (*k* + 1) to *Total*
    e. Increment k by 1
    f. Go to **step c**
    g. Display the value of *Total*

Case Study: Algorithm 1 (Structured Version)

1. Prompt the user for input

2. Accept an integer for *Number*

3. If Number < 0 then
    Display “Error in input”
  Else
    a. Initialize *Total* to 0
    b. For each integer *k* in the range 1 to *Number*
        Assign *Total* + *k* to *Total*
    c. Display the value of *Total*