Problem Solving

Techniques for visualizing non-trivial problems and coming up with solutions

Recall

- Computer Science is about problem solving.
- There isn't an exact prescribed formula or series of steps that you can learn and apply.
- This section cannot/will not provide you with a simple set of steps that you can memorize and apply in any situation.
  - Think about the many ways in which your assignments could have been implemented!
- What will be provided is a set of approaches that may be used to solve some types of problems.
  - Afterwards you may be able to apply these techniques when faced with problems that you face in future.
  - OR
  - You will be able to develop your own approaches.

How To Come With An Algorithm/Solution

- An algorithm is the series of steps (not necessarily linear!) that provide the solution to your problem.
- If there is a physical analogy to the problem then try visualizing the problem using real world objects or scenarios.

How To Come With An Algorithm/Solution (2)

- If the problem is more abstract (e.g., mathematical and no obvious physical model can be created)
  - For simple problems this may not be an issue (you can immediately see the solution).
  - For more complex problems you may be unable to come with the general solution for the program.
  - Try working out a solution for a particular example and see if that solution can be extended from that specific case to a more generalized formula.

Algorithm != Program Code

- Although program code is an example of an algorithm not all algorithms must take the form of a computer program.
- Algorithms due include code but it can take other forms:
  - Pseudo code
  - Flowchart diagrams
- The advantage of specifying an algorithm in pseudo code or a diagram is that you don't have worry about all the details of syntax.
  - In the beginning you can focus on just one thing: finding answer to the problem rather than issues such as "How do I do this in Python?"…reduces complexity.
  - Then when you have created a solution you can worry about issues such as syntax.

James Tam

Already covered this term: lecture

How To Find the Algorithm: Pseudo Code

Already covered this term: lecture

Pseudo code: investment program

```
count off ten years repeat the body
  # All of the below is done each year

  # This repeats while user input not valid
  while user selection is not valid repeat the body
      Get user selection for investment

  if (investment selected is high risk)
  determine loss/gain
  # check this if case only if first case doesn't
  apply
  If (investment is medium risk)
  determine loss/gain
  # check this case only if first two cases don't
  apply
  if (investment is low risk)
  determine gain
  # If all three cases don't apply
  If (user input is not 1 - 3)
  Display error message
```

James Tam

Specifying Pseudo-Code

- It can specified at different levels of detail (very general to very specific).
- There is not a hard and fast rule for the detail level.
- Guidelines:
  - When creating pseudo code the level of detail must not be as specific as program code (otherwise issues like syntax can take over the problem-solving process).
  - Yet pseudo code must provide enough detail so that you can eventually use it to write actual program code.

James Tam

Already covered this term: tutorial

Problem #1: Change Making

Already covered this term: tutorial

- (Paraphrased from the book "Pascal: An introduction to the Art and Science of Programming" by Walter J. Savitch.

Problem statement:

Design a program to make change. Given an amount of money, the program will indicate how many quarters, dimes and pennies are needed. The cashier is able to determine the change needed for values of a dollar or less.

Actions that may be needed:

•Action 1: Prompting for the amount of money

•Action 2: Computing the combination of coins needed to equal this amount

•Action 3: Output: Display the number of coins needed

Already covered this term: lecture

Problem #1: Change Making

Already covered this term: lecture

- How do you come up with the algorithm/solution for determining the amount of change owed?
  - Don't just shout the solution!
    - The solution is just the answer to one specific problem: which really isn't that important).
  - Instead think about *how to come up with* the solution
    - This is the ability to come up with solutions to ANY problem as opposed to just knowing the solution to ONE problem.
  - In this case you can try working out the algorithm for a specific example (i.e., how much change do you give back when a specific amount is owed and extrapolate that specific case to the general formula which will work for any amount owed).

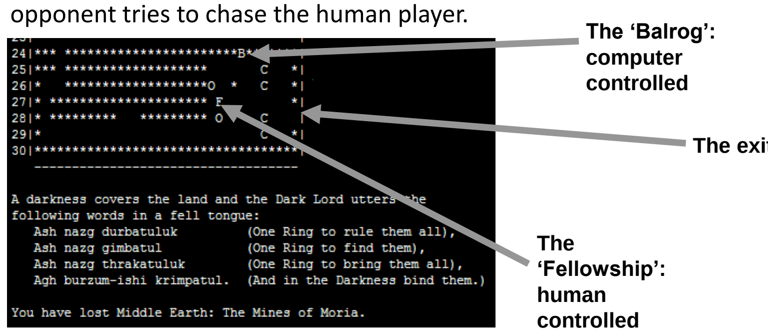
Problem #1: Change Making

- Illustrative key point
- When you are unable to come up with an algorithm that can be applied to many cases then try applying the second problem solving technique:
  - Try working out a solution for a particular example and see if that solution can be extended from that specific case to a more generalized formula.
  - Work out change making for different but specific amounts e.g., $0.25, $0.26, $0.30, $0.32 etc.
  - Use these specific examples to help you discover the general formula for the problem (in this case making change).

James Tam

Problem #2: Path Finding (Chase Algorithm)

– Write a program that implements some rudimentary artificial intelligence. The program is a game that takes place on a 2D grid. The human player tries to find the exit and the computer controlled opponent tries to chase the human player.



Problem #2: Hint

- What exactly does chase mean?
  - Move towards.
- In order to determine what exactly 'move towards' means try:
  - Applying first problem solving technique: Create a physical replica of the game world (numbered grid using a white/black board or graph paper).
  - Applying second problem solving technique: Use specific examples of 'chase' scenarios to help you determine the algorithm for scenario where a case must occur.

Problem Solving: Final Hint

- Keep in mind: a computer is dumb!
- Each step in an algorithm must be explicitly given to a computer.
  - Either the programmer writes the step in the form of a program instruction
  - Or else the programming language includes this information as part of it's syntax.
  - When specifying a solution to a problem don't skip steps!

Detail Level Required: Example

- What is *an algorithm* for finding the word in the following collection that comes first (alphabetical order):
  - Dog
  - Cat
  - Bird
  - Fish

Specifying Details: Example

- sometimes be more a nuisance than a benefit. This was found to be the case in my own investigation of potential change display mechanisms summarized in Chapter 5 and published as Tam, McCaffrey, Maurer, and Greenberg (2000). During this study, many test participants expressed a desire for useful abstractions that combine rudimentary change information into one higher-level conceptual change. For example, one participant noted while watching the animated replay of a class name being shown, "…I don't need to see each and every character being typed just to see a name change!" Of course, care must be taken to make these abstractions understandable, e.g., by using already familiar representations or notations. This minimizes the cost of acquiring information while maximizing its benefits due to the added structure and organization.
- Based upon my previous findings (to be discussed in Chapter 5), I add a third dimension, *persistenc*e, to Gutwin's classification. Persistence refers to how long the information is displayed (Figure 4.1 side pane). The display of information is *permanent* if it is always visible and *passing* if it only appears for a certain period. We noticed how study participants frequently complained when important information disappeared off the screen. Conversely, they also indicated that screen clutter might occur with the mechanisms that constantly displayed all changes. Thus, there's a need to classify change information according to how long it should stay visible.
- With permanent persistence, the effort needed to find changes i.e., the acquisition cost is low because the information is always there. Ideally, a person merely has to shift their gaze over to see the information. Because people can become accustomed to the occurrence of workspace events, they can also ignore things that do not interest them and pay closer attention to things that are of interest (Gutwin 1997).
- With passing persistence, information about changes is presented only for a limited duration. This is useful when the information applies only to a specific portion of the project (artifact or group of artifacts) being viewed, or when the change information otherwise becomes irrelevant. This is quite an important point for us.
- The matrix in Figure 4.1 suggests that these dimensions can be combined, giving eight possibilities. For example, a literal, situated and passing display of changes is depicted in Figure 4.2a. The figure shows an animation of a changed circle (by using a 'replay' technique) where the circle literally retraces the path that it took as it was moved. It is situated because the animation occurs in the same place that the change actually happened. The persistence is 'passing' because once an animation has replayed a change, the information is gone. Figure 4.2b shows two other examples within a concept map editor. The first illustrates the symbolic, situated and permanent octant, where color value (shades of gray) is used to indicate changed 'Jim' and 'Jack' nodes. Thus, it is symbolic because changes are mapped to a gray scale value, situated because the shading is applied directly to the node that was changed, and permanent because the color values are always on. Figure 4.2b also portrays an example of the symbolic, separate, and passing octant, where a person can raise a node's change details in a pop-up as a text description by mousing-over the node. Thus it is somewhat separate as the information appears outside the changed node, it is symbolic as it uses the text to describe the changes, and passing because the pop-up disappears when the person moves the mouse off the node (not quite on the node).