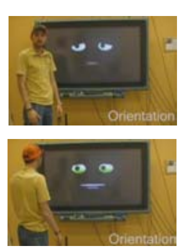
Introduction To Problem Solving

This section will focus on problem solving strategies.

What Is Computer Science?

•Computer Science is about problem solving

Graphics



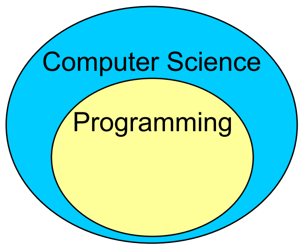
Interactive displays

Robotics: acceptance of domesticated robots

Artificial Intelligence
FIFA © Electronic Arts.

Computer Science Is Not The Same As Computer Programming

•Computer Science does require the creation of computer programs (‘programming’) but goes beyond that.



Computer Science Vs. Computer Programming

•Computer programming
- The focus on how to do different things with a given programming language

•Computer science
- Learning computer programming is a necessary part of computer science
- But it also includes extensive details about how a computer program works (not just how to write one in a given language).
- Also problem solving and reasoning skills must be developed and applied.

Problems Vs. Solutions

- Problem
  - Specifies '*what*' needs to be accomplished.
  - Includes a description of inputs and outputs.
- Solution
  - The way in which the problem is solved (the specific steps that specifies '*how*' the inputs are converted into outputs).
  - Algorithm: the name of a solution in Computer Science.

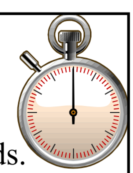
More On Problem Solving

- The process of problem solving is the development of an algorithm that fulfills the requirements of a problem).
- Typically there will multiple algorithms that could solve the problem.
  - Some solutions could be better than others (depending upon the criterion used).
- This means that there isn't a fixed series of steps that can be followed in order to solve the problem (the person who solves the problem – in this course it will be YOU – must come up with the algorithm).

Practice Examples For Working Out Problems

- Average running times
- Robotic movements

Simple Problem: Average Running Times



- Running times are typically recorded in minutes and seconds.
- Problem: Find a way to allow for an easier calculation of an average running time.
- Inputs: Time in minutes and the time in seconds.
- Output: A running time using one time unit.
- Solution: Derive an algorithm that will convert the composite time in minutes and in seconds to a single value in minutes so an average can be determined.

Another Problem: Robotic Movement[1]

- Develop an algorithm for a simple robot (similar in movement capabilities to a Roomba™).
- Movement:
  - The robot can move forward one distance unit (a 'square').
- Rotation:
  - If forward motion is not possible then the robot can rotate left or right by 90 degrees.
- Short range sensors:
  - One is mounted forwards, the other is mounted on the right.
  - The sensors check for obstacles in the next square.

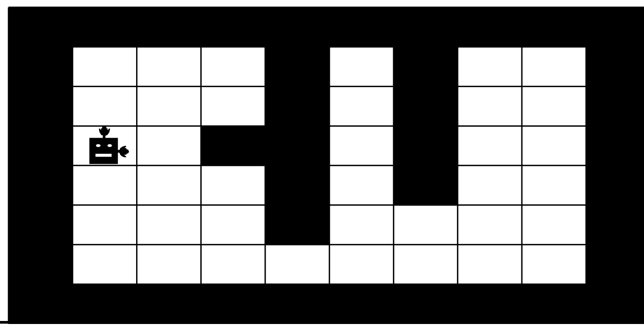
1 From "Peeking into Computer Science" and the lecture notes of Jalal Kawash

Specifying The Problem

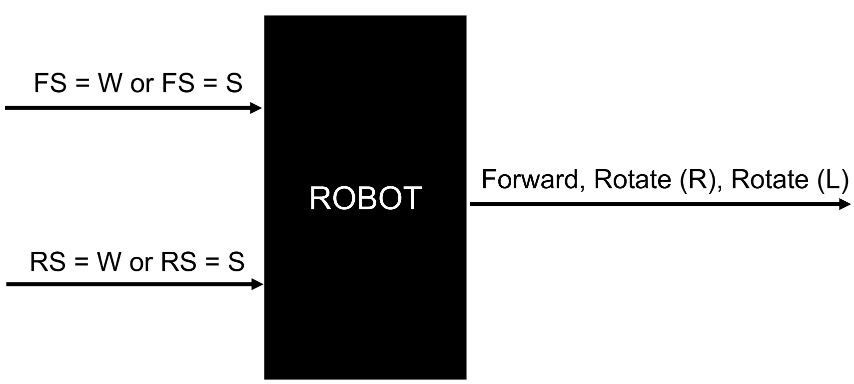
- What does the robot need to do:
  - Find a wall/obstacle.
  - Hug the wall, indefinitely moving forward.
- Input:
  - Whatever is detected by the sensors (front, right).
- Output:
  - The robot's movement

The Contents Of The Robot's World

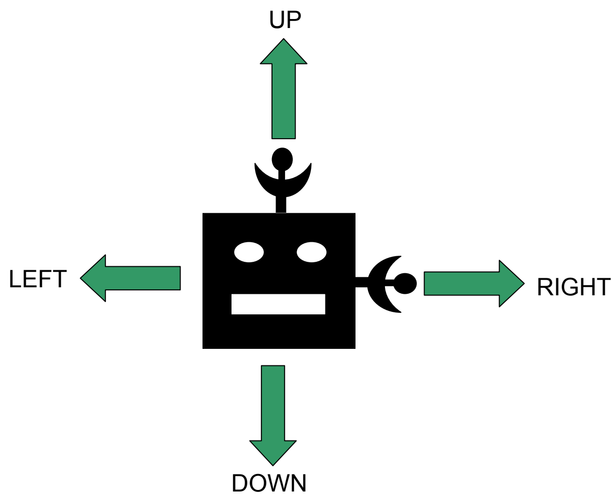
- Since the robot can either move onto a square that's empty or avoid a square that is occupied, the world can be simplified into two cases:
  - The destination square is empty: 'space'.
  - The destination square is not empty: 'wall' (contains a wall, furniture, person, pet etc.)
    - Details about exactly why the destination isn't empty isn't important so simplify the problem.



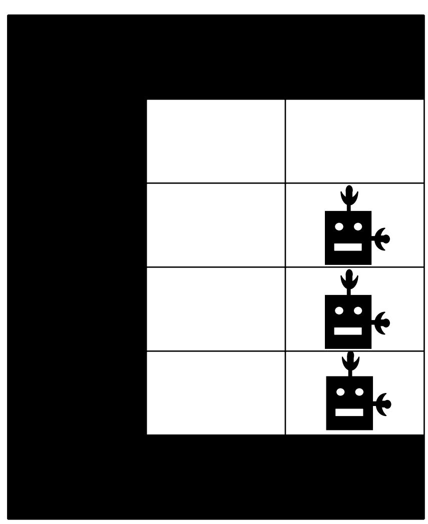
Inputs/Outputs Based On The World



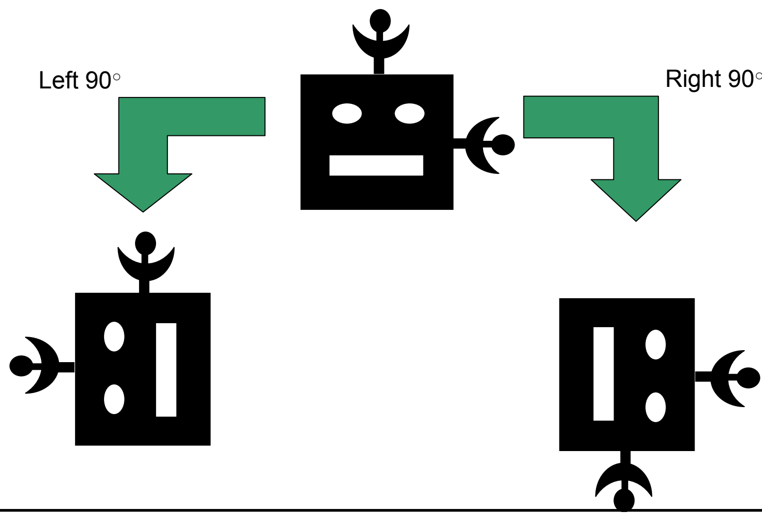
Robot's Orientation



Robot: Moving Forward



Robot: Rotations



Solution: The Generic Algorithm For Movement

- Search for the wall
- Once found, keep it to the robot's right
- Move forward
  - Each move, make sure the wall is still to the robot's right.
  - (This means there should be a space in front and the wall to the right).

- Robot's modes:
  - Search for the wall
  - Hug the wall

Algorithm: Search For The Wall

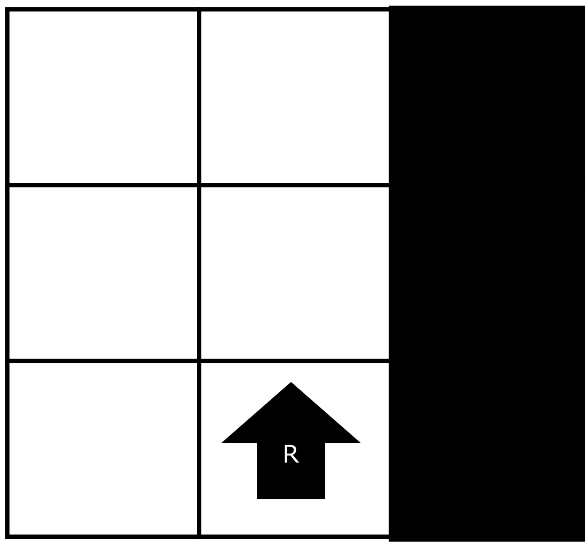
Repeat the following steps, until this phase is done (wall found, change to the wall hugging mode)

- If **RS = W**, then done this phase
  - Right sensor detects a wall
- If **FS = W**, then **L,** done this phase
  - Front sensor detects a wall, rotate left
- If **FS = S**, then **F**
  - Right sensor senses a space, take a step forward

Algorithm: Hug The Wall

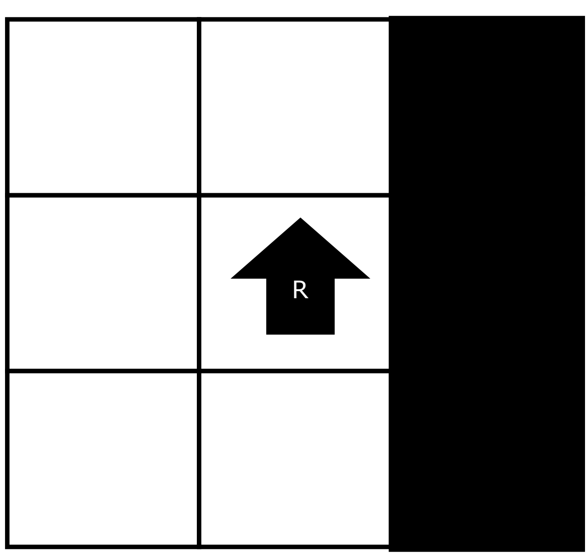
- Need to make sure that the wall is not "lost" during movement.
- Complexity: all cases must be considered.

Hug The Wall: Case 1



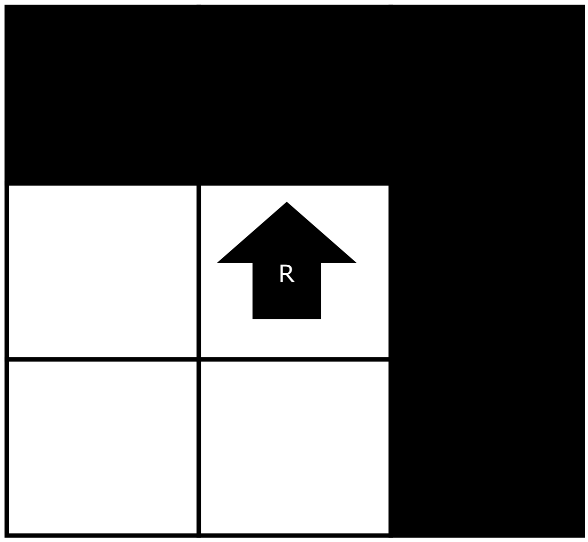
Input
RS = W
FS = S

Hug The Wall: Case 1



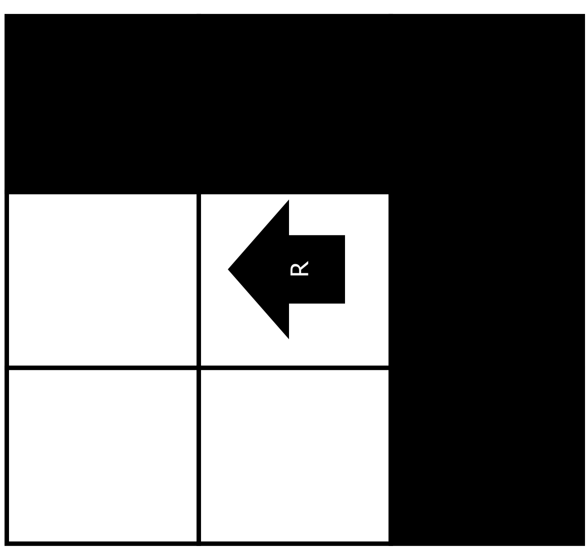
Output
Movement: forward

Hug The Wall: Case 2



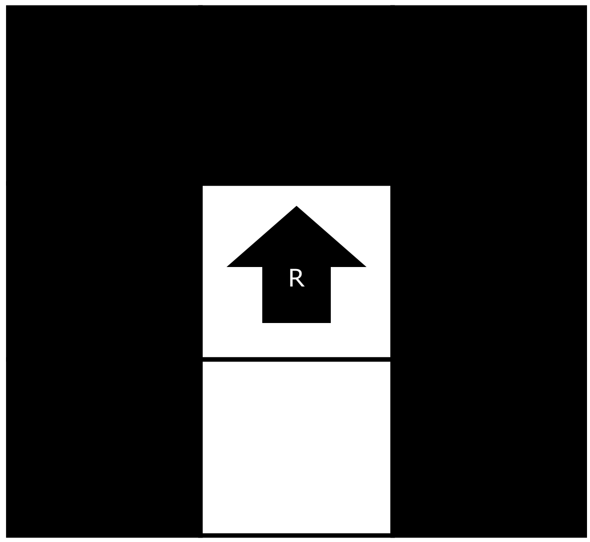
Input
FS = W

Hug The Wall: Case 2



Output
Rotate: left

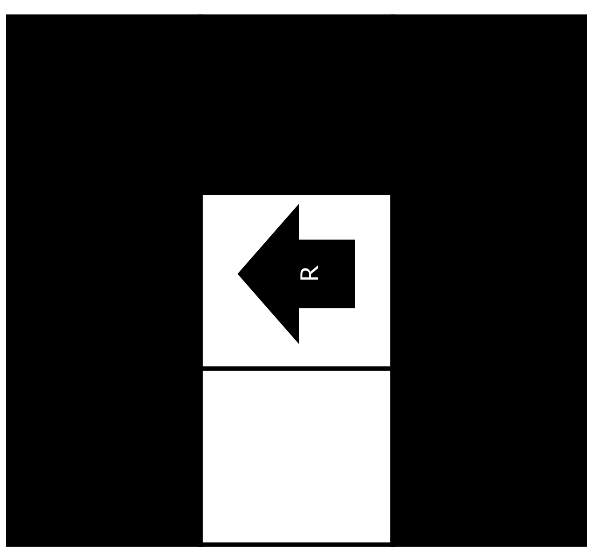
Hug The Wall: Case 3



Input
FS = W

James Tam

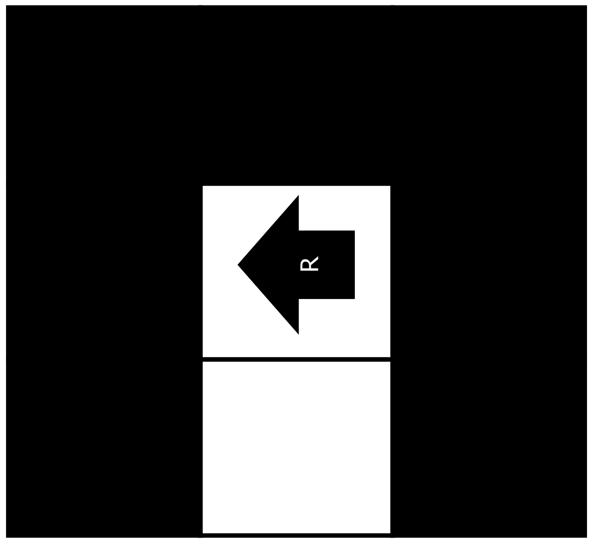
Hug The Wall: Case 3



Output
Rotate: Left

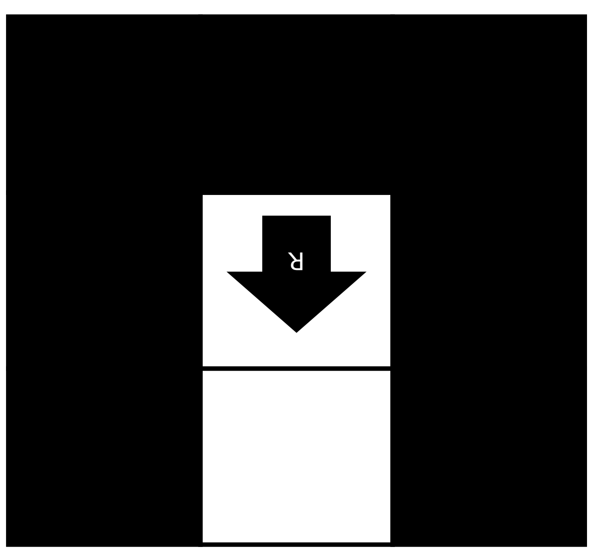
James Tam

Hug The Wall: Case 3



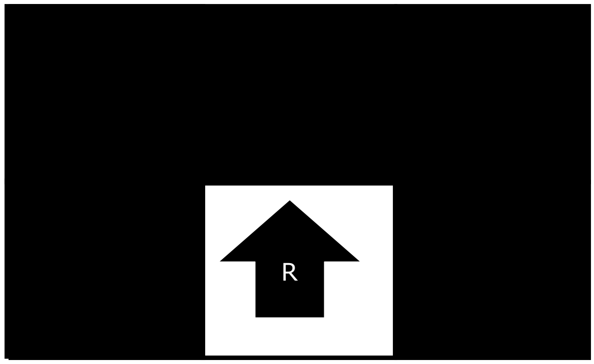
Input
FS = W

Hug The Wall: Case 3



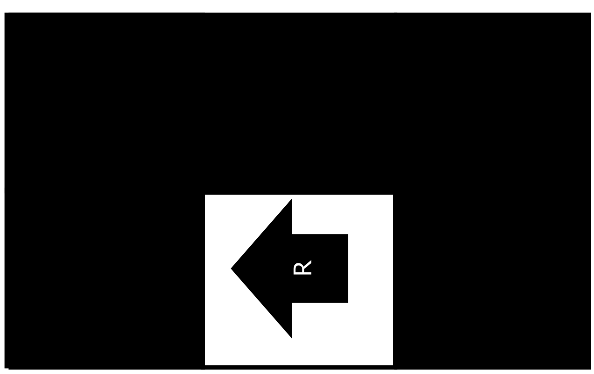
Output
Rotate: Left

Hug The Wall: Case 3



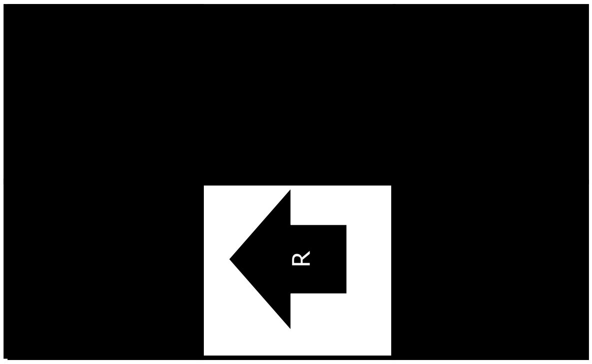
Input
FS = W

Hug The Wall: Case 3



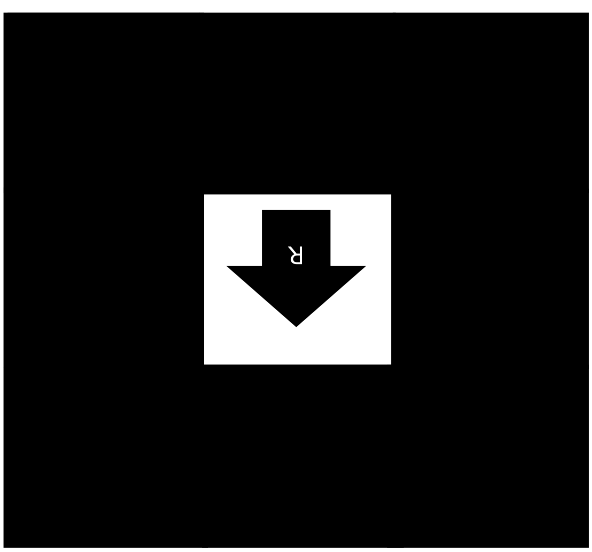
Output
Rotate left

Hug The Wall: Case 3



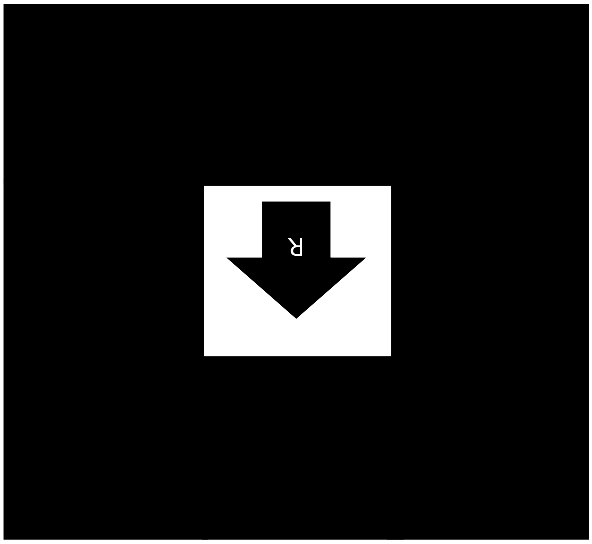
Input
FS = W

Hug The Wall: Case 3



Output
Rotate left

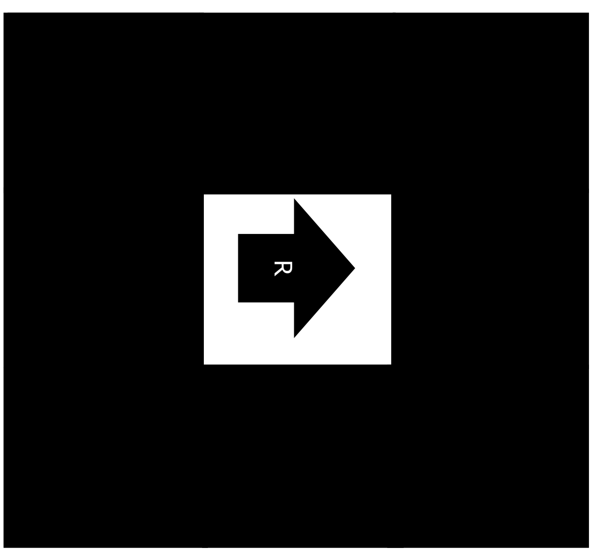
Hug The Wall: Case 3



Input

FS = W

Hug The Wall: Case 3

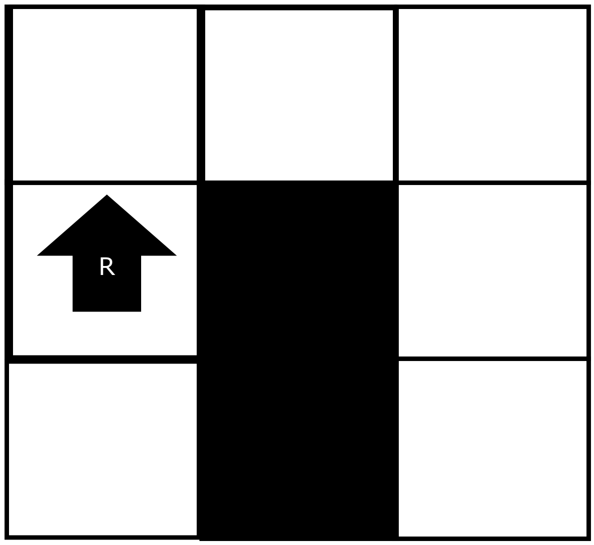


Output

Rotate left

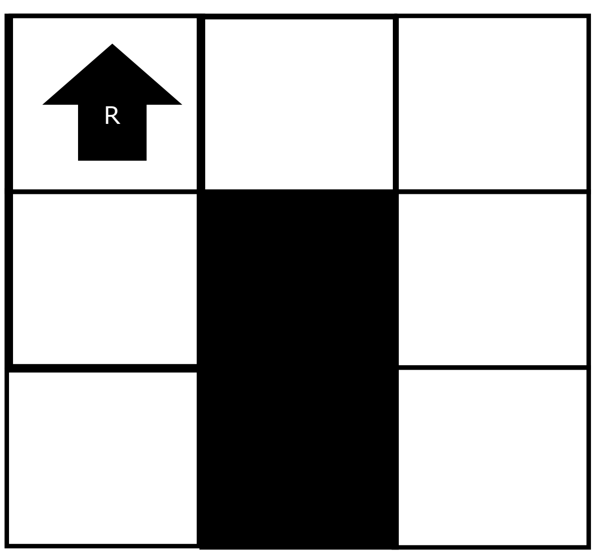
http://adorablay.wordpress.com/

Hug The Wall: Case 4



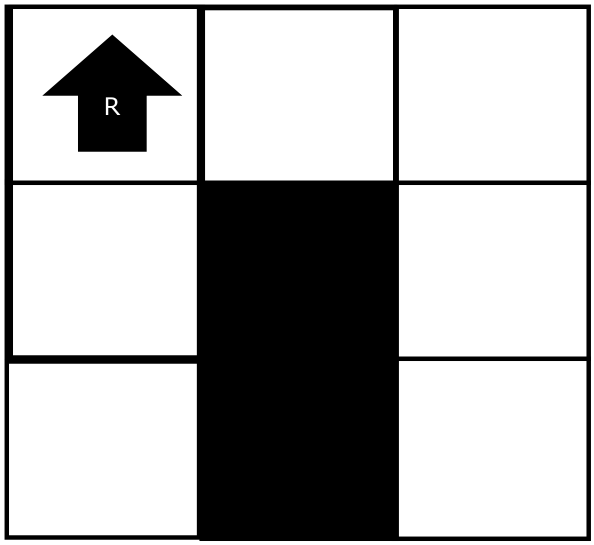
Input
FS = S
RS = W

Hug The Wall: Case 4



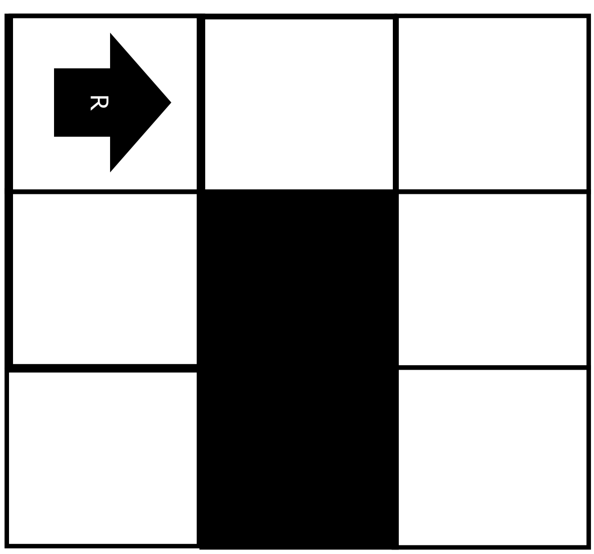
Output
Movement:
Forward

Hug The Wall: Case 4



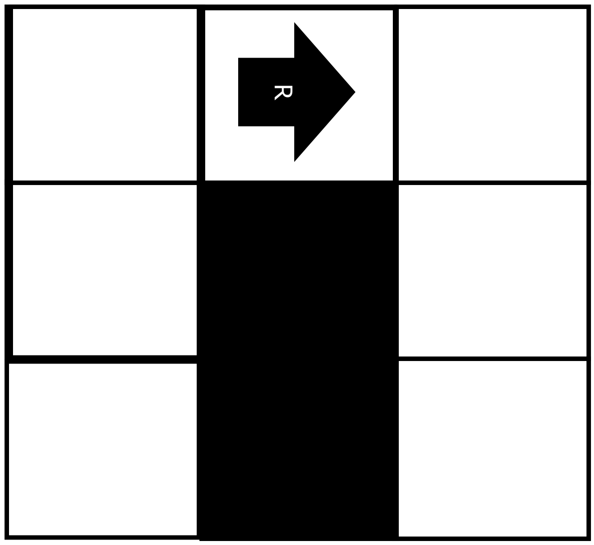
Input
FS = S
RS = S

Hug The Wall: Case 4



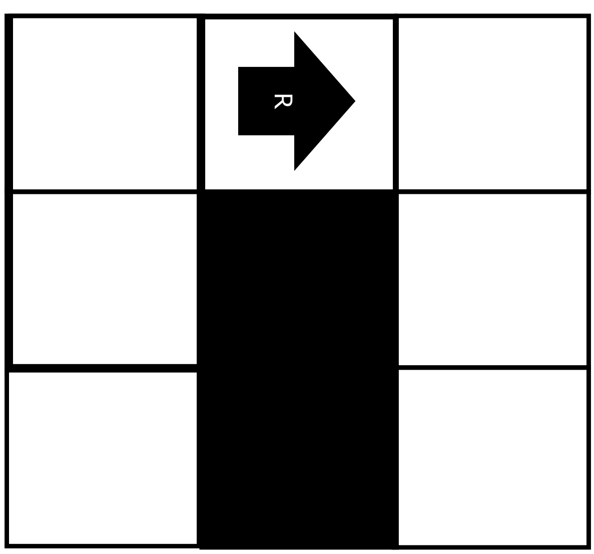
Output (Step 1)
Rotate: right

Hug The Wall: Case 4



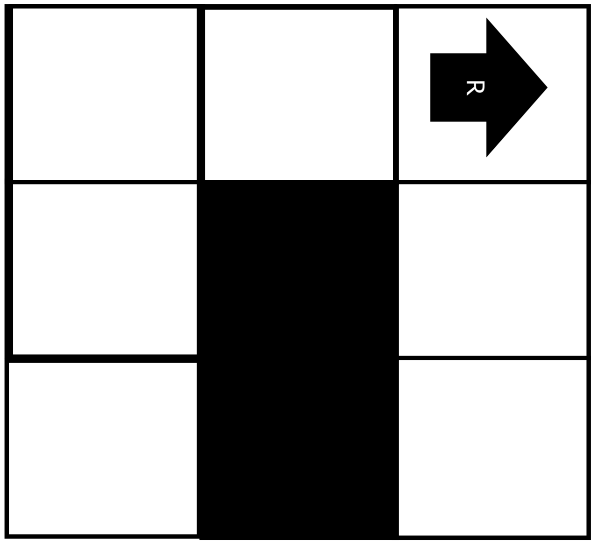
Output (Step 2):
Movement:
forward

Hug The Wall: Case 4



Input:
FS = S
RS = W

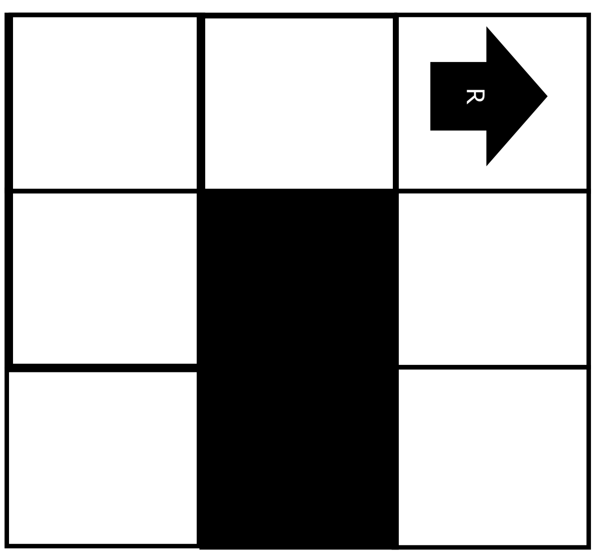
Hug The Wall: Case 4



Output:
Movement:
forward

James Tam

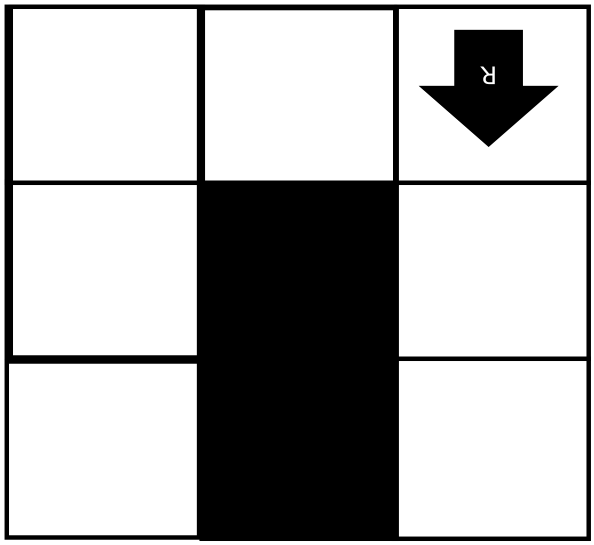
Hug The Wall: Case 4



Input:
FS = S
RS = S

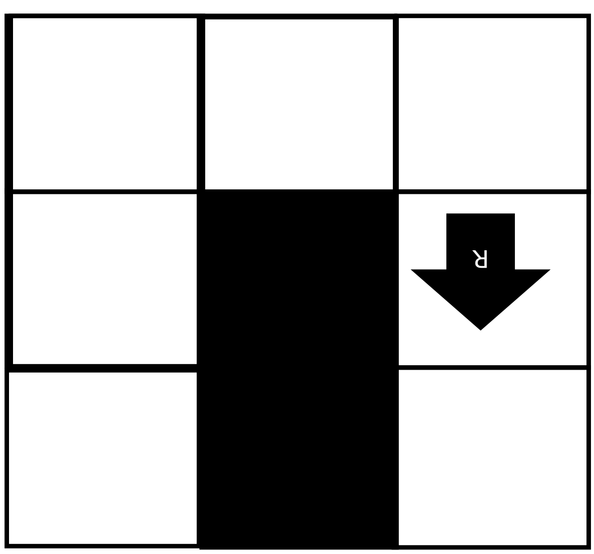
James Tam

Hug The Wall: Case 4



Output (step 1):
Rotate: right

Hug The Wall: Case 4



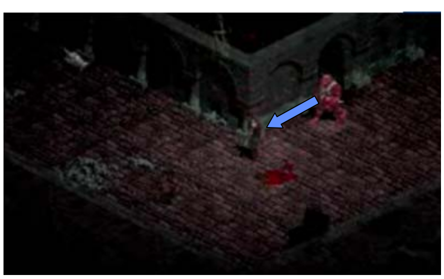
Output (step 2):
Movement: foward

Algorithm: Hug The Wall

Repeat the following steps:

1. If **RS = W** and **FS = S**, then **F**
2. If **FS = W**, then **L**
3. If **RS = S** and **FS = S**, then **R** and **F**

How To Develop Solutions For Tougher Problems



1. Try to solve specific examples and from those cases extrapolate the general algorithm.
2. If there is a physical analogy: Specify the scenario in concrete terms to help you visualize what you're facing.
   - This doesn't necessarily involve a text description but could include something more physical or visual.
   - Note: this approach can also be used to help clarify the problem that you face.

First Problem: Change Calculator

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

Problem statement:

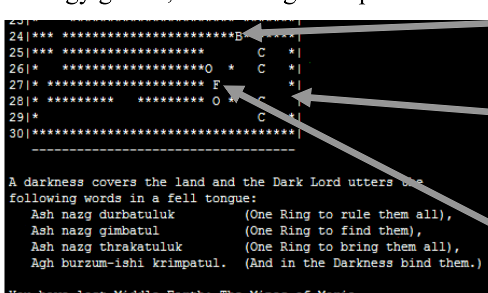
Write the algorithm to make change. Given an amount of money, the program will indicate how many quarters, dimes and pennies are needed. The cashier is able to determine the change needed for values of a dollar or less.

Second Problem: Simple Path Finding

•Path finding is a common problem in many programs.

•Examples:

- Websites that give directions from one location to anther e.g., MapQuest, Google Maps etc.
- For games with rudimentary artificial intelligence and given two locations on a map, the game will plot a path from one location to another e.g., chase algorithms for computer controlled opponents, moving armies/fleets in strategy games, controlling multiple characters in RPG's.



Solving Large Problems

- Structure the problem so it becomes manageable.
- This can be done by *abstracting* (simplifying the problem).
- One approach to abstracting is to hide details that aren't immediately necessary or focusing on details that are more important.
  - Example: The robot example, either the destination was empty or it wasn't (the exact contents aren't important).
- Later the other details may be deal with as needed/possible.
- A commonly used approach is the top-down method.
- Start with a general approach to the problem (the "top").
- Decompose that approach into smaller portions (moving 'down').

Solving Large Problems (2)

- Continue decomposing the problem into smaller and smaller parts until each part alone is manageable and can be solved.
- The solution to each part is an algorithm.

Top Down Design

1. Start by outlining the major parts (structure)

My autobiography

Chapter 1: The humble beginnings

Chapter 2: My rise to greatness

…

Chapter 7: The end of an era

2. Then implement the solution for each part

Chapter 1: The humble beginnings

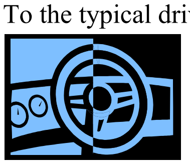
It all started ten and one score years ago with a log-shaped computer workstation…

Levels Of Abstraction

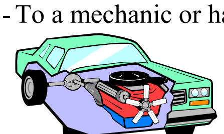
- The appropriate level of detail will depend upon the person and what they need to accomplish.
- Example a vehicle:
  - To a passenger

    A → B

  - To the typical driver



  - To a mechanic or hard-core enthusiast



You Should Now Know

- Computer Science is about problem solving
- How Computer Science differs from computer programming
- What is the definition of a problem
- What is the definition of a solution/algorithm
- How to work out the details of a problem of moderate difficulty
- Some techniques for solving challenging problems
- How to manage the complexity of larger problems through abstraction and top down decomposition