Computer Science 6

JavaScript

Dive deeper into more advanced algorithms, data structures, and computation.

#1. Misty Island Mine

Level Overview and Solutions

Intro

Command a troupe of peasants to collect their best coin.

Command them to `"buildXY"` a `"decoy"` when an enemy gets too close! Check for under **10m**.

Default Code

```
// Collect gold efficiently by commanding peasants wisely!
// Peasants should collect coins and build decoys.

// The function should return the best item per target
// Use an array of ids to ensure no two peasants target the same item.
function findBestItem(friend, excludedItems) {
    var items = friend.findItems();
    var bestItem = null;
    var bestItemValue = 0;
    for(var i = 0; i < items.length; i++) {
        var item = items[i];
        // indexOf searches and array for a certain element:
        var idx = excludedItems.indexOf(item);
        // If the array doesn't contain it, it returns -1
        // In that case, skip over that item as another peasant is targeting it.
        if(idx != -1) { continue; }
        // Finish the function!
        // Remember bestItemValue should be the highest item.value / distanceTo

    }
    return bestItem;
}

while(true) {
    var peasants = hero.findByType("peasant");
    // Create a new array every loop.
    var claimedItems = [];
    for(var i = 0; i < peasants.length; i++) {
        var peasant = peasants[i];
        var enemy = peasant.findNearestEnemy();
        if(enemy) {
            // If the distance to enemy is < 10
            // AND the hero has enough gold for a decoy

                // Command a peasant to build a "decoy":

                // Add a continue so the peasant doesn't collect coins when building.
        }
        var item = findBestItem(peasant, claimedItems);
        if(item) {
            // After an item is claimed, stick it in the claimedItems array.
            claimedItems.push(item);
            // Command the peasant to collect the coin:

        }
    }
}
```

Overview

To command a peasant to build, use: `command(peasant, "buildXY", "decoy", x, y)`

For this level, build your decoys at `peasant.pos.x - 2`, so the decoy will lead the ogres into the arrow towers.

Misty Island Mine Solution

```
// Collect gold efficiently by commanding peasants wisely!
// Peasants should collect coins and build decoys.

// The function should return the best item per target
// Use an array of ids to ensure no two peasants target the same item.
function findBestItem(friend, excludedItems) {
    var items = friend.findItems();
    var bestItem = null;
    var bestItemValue = 0;
    for(var i = 0; i < items.length; i++) {
        var item = items[i];
        // indexOf searches and array for a certain element:
        var idx = excludedItems.indexOf(item);
        // If the array doesn't contain it, it returns -1
        // In that case, skip over that item as another peasant is targeting it.
        if(idx != -1) { continue; }
        // Finish the function!
        // Remember bestItemValue should be the highest item.value / distanceTo
        var value = item.value / friend.distanceTo(item);
        if(value > bestItemValue) {
            bestItem = item;
            bestItemValue = value;
        }
    }
    return bestItem;
}

while(true) {
    var peasants = hero.findByType("peasant");
    // Create a new array every loop.
    var claimedItems = [];
    for(var i = 0; i < peasants.length; i++) {
        var peasant = peasants[i];
        var enemy = peasant.findNearestEnemy();
        if(enemy) {
            // If the distance to enemy is < 10
            // AND the hero has enough gold for a decoy
            if(peasant.distanceTo(enemy) < 10 && hero.gold >= 25) {
                // Command a peasant to build a "decoy":
                hero.command(peasant, "buildXY", "decoy", peasant.pos.x - 2, peasant.pos.y);
                // Add a continue so the peasant doesn't collect coins when building.
                continue;
            }
        }
        var item = findBestItem(peasant, claimedItems);
        if(item) {
            // After an item is claimed, stick it in the claimedItems array.
            claimedItems.push(item);
            // Command the peasant to collect the coin:
            hero.command(peasant, "move", item.pos)
        }
    }
}
```

#2. Grim Determination

Level Overview and Solutions

Intro

Paladins are capable of casting a spell! They can cast the spell: `heal` on nearby targets, when it's off cooldown.

```
var paladin = hero.findNearest(hero.findByType("paladin"))
if paladin.canCast("heal", hero) {
    hero.command(paladin, "cast", "heal", hero);
}
```

Default Code

```
// Your goal is to protect Reynaldo

// Find the paladin with the lowest health.
function lowestHealthPaladin() {
    var lowestHealth = 99999;
    var lowestFriend = null;
    var friends = hero.findFriends();
    for(var f=0; f < friends.length; f++) {
        var friend = friends[f];
        if(friend.type != "paladin") { continue; }
        if(friend.health < lowestHealth && friend.health < friend.maxHealth) {
            lowestHealth = friend.health;
            lowestFriend = friend;
        }
    }
    return lowestFriend;
}

function commandPaladin(paladin) {
    // Heal the paladin with the lowest health using lowestHealthPaladin()
    // You can use paladin.canCast("heal") and command(paladin, "cast", "heal", target)
    // Paladins can also shield: command(paladin, "shield")
    // And don't forget, they can attack, too!

}

function commandFriends() {
    // Command your friends.
    var friends = hero.findFriends();
    for(var i=0; i < friends.length; i++) {
        var friend = friends[i];
        if(friend.type == "peasant") {
            //commandPeasant(friend);
        } else if(friend.type == "griffin-rider") {
            //commandGriffin(friend);
        } else if(friend.type == "paladin") {
            commandPaladin(friend);
        }
    }
}

while(true) {
    commandFriends();
    // Summon griffin riders!

}
```

Overview

This level introduces the Paladin ally. Paladins can cast a healing spell, and can use a shield to withstand more damage then usual.

The sample code gives you a `lowestHealthPaladin` function, which returns the paladin with the lowest health, if any are damaged.

Fill in `commandPaladin` using `lowestHealthPaladin` along with `canCast("heal",target)` and `command(paladin, "cast", "heal", target)` and `command(paladin, "shield")` to keep your paladins alive.

Write `commandPeasant` to have your peasants collect coins.

Write `commandGriffin` to have your Griffin Riders attack the enemy.

Finally, in the main loop, you should summon Griffin Riders when you have enough gold.

Tip: Paladins also carry a mace to `attack` with!

Grim Determination Solution

```
// Your goal is to protect Reynaldo

// Find the paladin with the lowest health.
function lowestHealthPaladin() {
    var lowestHealth = 99999;
    var lowestFriend = null;
    var friends = hero.findFriends();
    for(var f=0; f < friends.length; f++) {
        var friend = friends[f];
        if(friend.type != "paladin") { continue; }
        if(friend.health < lowestHealth && friend.health < friend.maxHealth) {
            lowestHealth = friend.health;
            lowestFriend = friend;
        }
    }
    return lowestFriend;
}

function commandPaladin(paladin) {
    // Heal the paladin with the lowest health using lowestHealthPaladin()
    // You can use paladin.canCast("heal") and command(paladin, "cast", "heal", target)
    // Paladins can also shield: command(paladin, "shield")
    // And don't forget, they can attack, too!
    var needyPaladin = lowestHealthPaladin();
    var enemies = hero.findEnemies();
    var enemy = null;
    if(needyPaladin && needyPaladin.health < 300) {
        if(paladin.canCast('heal')) {
            hero.command(paladin, "cast", "heal", needyPaladin);
        } else {
            enemy = paladin.findNearest(enemies);
            if(enemy) {
                hero.command(paladin, "attack", enemy);
            }
        }
    } else {
        enemy = paladin.findNearest(enemies);
        if(enemy) {
            hero.command(paladin, "attack", enemy);
        }
    }
}

function commandPeasant(peasant) {
    var items = peasant.findItems();
    var item = peasant.findNearest(items);
    if (gPrevCoin && item == gPrevCoin && items.length) {
        item = items[0];
    }
    if(item) {
        gPrevCoin = item;
        hero.command(peasant, "move", {x:item.pos.x, y:item.pos.y});
    }
}

function commandGriffin(griffin) {
    var warlocks = hero.findByType('warlock');
    if(warlocks.length) {
        hero.command(griffin, "attack", griffin.findNearest(warlocks));
    } else {
        var enemy = griffin.findNearestEnemy();
        if (enemy) {
            hero.command(griffin, "attack", enemy);
        } else {
            hero.command(griffin, "move", {x:73, y:37});
        }
    }
}

function commandFriends() {
    // Command your friends.
    var friends = hero.findFriends();
    for(var i=0; i < friends.length; i++) {
        var friend = friends[i];
        if(friend.type == "peasant") {
            commandPeasant(friend);
        } else if(friend.type == "griffin-rider") {
            commandGriffin(friend);
        } else if(friend.type == "paladin") {
            commandPaladin(friend);
        }
    }
}
var gPrevCoin = null;
while(true) {
    gPrevCoin = null;
    commandFriends();
    // Summon griffin riders!
    if(hero.costOf('griffin-rider') <= hero.gold) {
        hero.summon('griffin-rider');
    }
}
```

#3. Yeti Eater

Level Overview and Solutions

Intro

Yetis surround us, so it will be a good hunt. Luckily, the wizard had time to cast the sleep spell.

Your hero can devour yetis' vital powers when they are defeated. So you have to defeat them in the order from the weakest to the strongest. Luckily (again), the wizard can sort enemies from the strongest to the weakest.

Reverse that list and act!

Default Code

```
// Yetis surround us and we need to defeat them.
// Luckily the wizard had time to cast the sleep spell.
// Your hero can devour the yetis' vital powers when they are defeated.
// Defeat them in the order from weakest to the strongest.

// The wizard sorted enemies, but in the order from the strongest to the weakest.
var wizard = hero.findNearest(hero.findFriends());
var yetis = wizard.findEnemies();

// You need iterate the yetis list in the reverse order with a 'for-loop'.
// The start value should be 'yetis.length - 1'.
// Iterate while the index greater than -1.
// Use the negative step -1.

    // Attack each enemy while its health greater than 0.


// Look at the guide to get hints.
```

Overview

We know the ordered list of the yetis, but we need it in the reversed order. To get the reversed list of the given you just need to read it from the end to the start.

`for-loop` can be useful for it. We just need to change start, end and step values:

```
var array = [1, 2, 3];
// Don't forget about use ">" instead "<"
for (var i = array.length - 1; i > -1; i -= 1) {
    hero.say(array[i]);
}
```

We use the end value `-1`, because we shouldn't skip the 0-th index. Also, be careful with the start value: it should be less than the array's length by one.

It can be written with `while-loop`:

```
var index = array.length - 1;
while (index > -1) { // or 'index >= 0'
    hero.say(array[i]);
    i -= 1; // or 'i--'
}
```

Yeti Eater Solution

```
// Yetis surround us and we need to defeat them.
// Luckily the wizard had time to cast the sleep spell.
// Your hero can devour the yetis' vital powers when they are defeated.
// Defeat them in the order from weakest to the strongest.

// The wizard sorted enemies, but in the order from the strongest to the weakest.
var wizard = hero.findNearest(hero.findFriends());
var yetis = wizard.findEnemies();

// You need iterate the yetis list in the reverse order with a 'for-loop'.
// The start value should be 'yetis.length - 1'.
// Iterate while the index greater than -1.
// Use the negative step -1.
for (var i = yetis.length - 1; i > -1; i--) {
    // Attack each enemy while its health greater than 0.
    while (yetis[i].health > 0) {
        hero.attack(yetis[i]);
    }
}

// Look at the guide to get hints.
```

#4. Antipodes

Level Overview and Solutions

Intro

Match archers with their antipode (the enemy archer with the reversed named.)

Once all the antipodes are defeated, defeat the warlock.

P.S.: The warlock looks peaceful, but he can feel any aggression.

Default Code

```
// The warlock used the "clone" spell and created evil antipodes of our archers.
// But even that evil spell has weakness.
// If your archer touches his antipode, then it will disappear.
// If an archer touches the wrong clone or attacks one of them, then the clones start to fight.
// We can find antipodes by their names - they are each other's reverse.

// This function check two units whether they are antipodes or not.
function areAntipodes(unit1, unit2) {
    var reversed1 = unit1.id.split("").reverse().join("");
    return reversed1 === unit2.id;
}

var friends = hero.findFriends();
var enemies = hero.findEnemies();

// Find antipodes for each of your archers.
// Iterate all friends.

    // For each of friends iterate all enemies.

        // Check if the pair of the current friend and the enemy are antipodes.

            // If they are antipodes, command the friend move to the enemy.


// When all clones disappears, attack the warlock.
```

Overview

This level is about reversing strings. Consider arrays and strings as equivalent, so think of each **character** in a *string* as an **element** in an *array*.

Here is an algorithm to help get you started:

1. Create an empty string to store the reversed value.
2. Start at the last index of the string
3. Add that character to the initial string.
4. Decrement the index by 1.
5. Repeat 2-4 until reaching the first index.
6. Now you have a reversed string!

Antipodes Solution

```
// The warlock used the "clone" spell and created evil antipodes of our archers.
// But even that evil spell has weakness.
// If your archer touches his antipode, then it will disappear.
// If an archer touches the wrong clone or attacks one of them, then the clones start to fight.
// We can find antipodes by their names - they are each other's reverse.

// This function check two units whether they are antipodes or not.
function areAntipodes(unit1, unit2) {
    var reversed1 = unit1.id.split("").reverse().join("");
    return reversed1 === unit2.id;
}

var friends = hero.findFriends();
var enemies = hero.findEnemies();

// Find antipodes for each of your archers.
// Iterate all friends.
for (var i = 0; i < friends.length; i++) {
    // For each of friends iterate all enemies.
    for (var j = 0; j < enemies.length; j++) {
        // Check if the pair of the current friend and the enemy are antipodes.
        if (areAntipodes(friends[i], enemies[j])) {
            // If they are antipodes, command the friend move to the enemy.
            hero.command(friends[i], "move", enemies[j].pos);
        }
    }
}
// When all clones disappears, attack the warlock.
while (true) {
    var nearestEnemy = hero.findNearest(hero.findEnemies());
    if (nearestEnemy && nearestEnemy.type === "warlock") {
        for (i = 0; i < friends.length; i++) {
            hero.command(friends[i], "attack", nearestEnemy);
        }
        hero.attack(nearestEnemy);
    }
}
```

#5. The Hunt Begins

Level Overview and Solutions

Intro

A Burl migration! The perfect time to acquire some information about the sizes of Burls!

Tell Senick, the Trapper the average size of this Burl migration.

Default Code

```
// Senick is trying to find the elusive Burleous Majoris!
// But he doesn't know how big a Burleous Majoris would be...
// Find the average size of this burl population to use as a baseline!

// This function returns average size of all the burls in an array.
function averageSize(burls) {
    var sum = sumSize(burls);
    // Remember the average is the sum of the parts divided by the amount!
    return sum / burls.length;
}

// This function should return the sum of all the burls sizes.
function sumSize(burls) {
    // Implement the sum function using the burls 'size':

}

while(true) {
    // Find the average size of the burls by calling the 'averageSize' function.

    // Say the average size of the seen burls!

}
```

Overview

Senick is on the hunt. He is searching for a Burleous Majorus! A fabled member of the Burl species whose size is unmatched! But, he has no idea where to start...

Help Senick conduct a little 'field-research'. Find the average size of the Burl population, so it will become obvious when they stumble across a real Burelous Majorus.

The `average` is the `sum divided by` the `number of elements` counted. Written simply:

```
var sum = 1 + 2 + 5 + 1; // 4 elements!
var average = sum / 4; // The average of these 4 elements is "2"!
```

Don't forget how to write a sum function for an array:

```
function sumHealth(enemies) {
    // Start with a sum variable initialized to 0
    // Iterate over each element in the array
        // Add its health to the sum variable
    // Return the sum variable!
}
```

The Hunt Begins Solution

```
// Senick is trying to find the elusive Burleous Majoris!
// But he doesn't know how big a Burleous Majoris would be...
// Find the average size of this burl population to use as a baseline!

// This function returns average size of all the burls in an array.
function averageSize(burls) {
    var sum = sumSize(burls);
    // Remember the average is the sum of the parts divided by the amount!
    return sum / burls.length;
}

// This function should return the sum of all the burls sizes.
function sumSize(burls) {
    // Implement the sum function using the burls 'size':
    var sum = 0;
    for(var i = 0; i < burls.length; i++) {
        var burl = burls[i];
        sum += burl.size;
    }
    return sum;
}

while(true) {
    // Find the average size of the burls by calling the 'averageSize' function.
    var burls = hero.findEnemies();
    var avgSize = averageSize(burls);
    // Say the average size of the seen burls!
    hero.say(avgSize);
}
```

#6. Yak Heist

Level Overview and Solutions

Intro

Senick needs to lay the trap for the Burleous Majorus!

And a big burl needs a big trap... With big bait! Find the closest, above-average sand-yak for Senick to swindle.

Default Code

```
// Senick needs big bait for a big burl!
// Help Senick find an above average yak!
// Don't pick one too deep in the herd, or risk angering the group.

// This function should return the average size of all the yaks:
function averageSize(yaks) {
    var sum = 0;
    // Go through each yak and add its size to the sum.

    return sum / yaks.length;
}

var yaks = hero.findEnemies();
var avgSize = averageSize(yaks);

var bestYak = null;
var closestDist = Infinity;
for(var i = 0; i < yaks.length; i++) {
    var yak = yaks[i];
    var yakDistance = hero.distanceTo(yak);
    var yakSize = yak.size;
    // Check if the yak is:
    // The distance is closer than the current 'closestDist' AND
    // The size is bigger than the 'avgSize'.

        // Update the 'bestYak' and 'closestDist'

}
// Say the 'bestYak':
```

Overview

In this level you'll need to find the average size of yaks AND find the 'best' yak.

Finding the best yak is no longer a matter of finding the closest, or largest, but the hero must combine these two properties together to find an optimal choice! Instead of checking the distances exclusively, be sure to check the `avgSize` of the Yak as well so Senick doesn't end up grabbing a tiny Yak.

To find the `avgSize` be sure to fill out the `avg` function! Remember how to find an average:

```
var enemies = hero.findEnemies();
var sum = 0;
for(var i = 0; i < enemies.length; i++) {
    sum += enemies[i].health;
}
var avg = sum / enemies.length;
hero.say("The average health of my enemies is: " + avg);
```

Yak Heist Solution

```
// Senick needs big bait for a big burl!
// Help Senick find an above average yak!
// Don't pick one too deep in the herd, or risk angering the group.

// This function should return the average size of all the yaks:
function averageSize(yaks) {
    var sum = 0;
    // Go through each yak and add its size to the sum.
    for(var i = 0; i < yaks.length; i++) {
        var yak = yaks[i];
        sum += yak.size;
    }
    return sum / yaks.length;
}

var yaks = hero.findEnemies();
var avgSize = averageSize(yaks);

var bestYak = null;
var closestDist = Infinity;
for(var i = 0; i < yaks.length; i++) {
    var yak = yaks[i];
    var yakDistance = hero.distanceTo(yak);
    var yakSize = yak.size;
    // Check if the yak is:
    // The distance is closer than the current 'closestDist' AND
    // The size is bigger than the 'avgSize'.
    if(yakSize > avgSize && yakDistance < closestDist) {
        // Update the 'bestYak' and 'closestDist'
        closestDist = yakDistance;
        bestYak = yak;
    }
}
// Say the 'bestYak':
hero.say(bestYak);
```

#7. Slumbering Sample

Level Overview and Solutions

Intro

One last thing needs to be done before Senick's plan can be put into motion!

Report the average size of this gathering of Yetis and Senick will formulate the last of the plot!

Default Code

```
// One last stop before the plan can be set into motion!
// It's up to you to help Senick get the average size of these yetis.

// Find all the Yetis using 'findByType':

// Implement the averageSize function from scratch:

// Say the average size of the yetis:
```

Overview

In this level you are required to write the `average` function from scratch! If you are having a hard time remembering, use these comments to help:

```
function average(array) {
    // Initialize a sum variable
    // Iterate over each element
        // Add the element's size to the sum variable
    // Return the sum variable divided by the length of the array
}
```

Slumbering Sample Solution

```
// One last stop before the plan can be set into motion!
// It's up to you to help Senick get the average size of these yetis.

// Find all the Yetis using 'findByType':
var yetis = hero.findByType("yeti");
// Implement the averageSize function from scratch:
function averageSize(yetis) {
    var sum = 0;
    for(var i = 0; i < yetis.length; i++) {
        var yeti = yetis[i];
        sum += yeti.size;
    }
    return sum / yetis.length;
}
// Say the average size of the yetis:
hero.say(averageSize(yetis));
```

#8. Circle Walking

Level Overview and Solutions

Intro

Coming soon!

Default Code

```
// Mirror your partner's movement around the center X mark.
// Vectors can be thought of as an x, y position
// Vectors can also represent the distance and direction between two positions

// use Vector.subtract(vector1, vector2) to find the direction and distance from vector2 to vector1
// use Vector.add(vector1, vector2) to find the position you get when you start from vector1 and follow
    vector2

// Create a new Vector at the center X point
var center = new Vector(40, 34);

// A unit's position is actually a Vector!
var partner = hero.findByType("peasant")[0];

while(true) {
    // First, you want to find the Vector (distance and direction) of the partner's position to the
        center X.
    var vector;

    // Second, find the position your hero should moveTo starting from center, and following vector.
    var moveToPos;

    // Third, move to the moveToPos position.

}
```

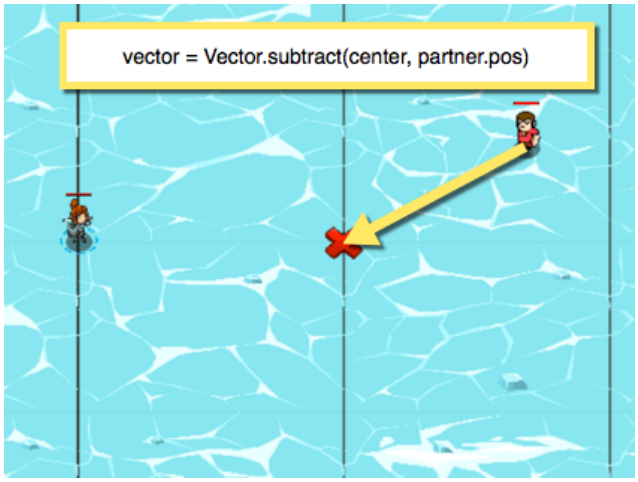
Overview

You should be familiar with unit position objects, like `hero.pos`. We've used them to specify fixed points in the 2D space using `pos.x` and `pos.y`.

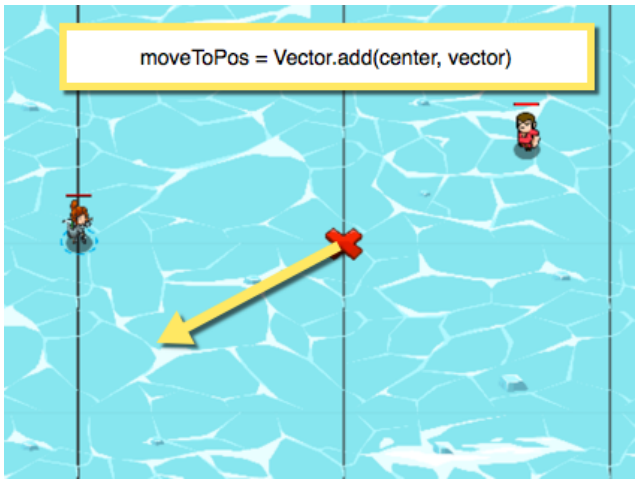
Vectors can be used to represent fixed points in the world, and in fact `hero.pos` is a vector.

Vectors can also be used to represent information **relative** to points in the world, such as the **distance** (or magnitude) and **direction** from one point to another.

In this level, you first need to use `Vector.subtract(center, partner.pos)` to get a vector that represents the distance and direction from your partner to the center X:



Then you take the vector you got from that, and use it to find the point your hero needs to move to, relative to the center X with `Vector.add(center, vector)`



The result, `moveToPos` is also a vector, which we will use as a fixed point with `move` to move the hero to the correct spot!

Circle Walking Solution

```
// Mirror your partner's movement around the center X mark.
// Vectors can be thought of as an x, y position
// Vectors can also represent the distance and direction between two positions

// use Vector.subtract(vector1, vector2) to find the direction and distance from vector2 to vector1
// use Vector.add(vector1, vector2) to find the position you get when you start from vector1 and follow
    vector2

// Create a new Vector at the center X point
var center = new Vector(40, 34);

// A unit's position is actually a Vector!
var partner = hero.findByType("peasant")[0];

while(true) {
    // First, you want to find the Vector (distance and direction) of the partner's position to the
        center X.
    var vector = Vector.subtract(center, partner.pos);
    // Second, find the position your hero should moveTo starting from center, and following vector.
    var moveToPos = Vector.add(center, vector);
    // Third, move to the moveToPos position.
    hero.move(moveToPos);
}
```

#9. Skating Away

Level Overview and Solutions

Intro

Coming soon!

Default Code

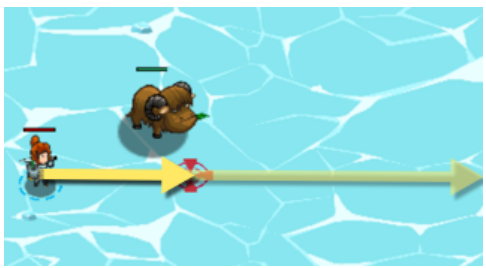
```
// Move to the red X mark while avoiding the yaks.
// use Vector.normalize(vector1) to create a vector in the same direction as vector1, but with a
    distance of 1
// use Vector.multiply(vector1, X) to create a vector in the same direction as vector1, but with its
    distance multiplied by X

// The point you want to get to.
var goalPoint = new Vector(78, 34);

while(true) {
    // This creates a vector that will move you 10 meters toward the goalPoint
    // First, create a vector from your hero to the goal point.
    var goal = Vector.subtract(goalPoint, hero.pos);
    // Then, normalize it into a 1m distance vector
    goal = Vector.normalize(goal);
    // Finally, multiply the 1m vector by 10, to get a 10m long vector.
    goal = Vector.multiply(goal, 10);

    // To avoid the yaks, if you get within 10 meters of a yak, you should vector away from it.
    var yak = hero.findNearest(hero.findEnemies());
    var distance = hero.distanceTo(yak);
    if(distance < 10) {
        // First, make a Vector from the yak to you

        // Now use Vector.normalize and Vector.multiply to make it 10m long

        // Once you have the 10m vector away from the yak, use Vector.add to add it to your goal vector!

    }
    // Finally, determine where to move by adding your goal vector to your current position.
    var moveToPos = Vector.add(hero.pos, goal);
    hero.move(moveToPos);
}
```

Overview

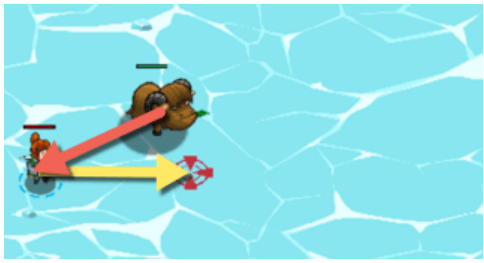
A Vector can represent the direction and distance between two points. Sometimes you'll want to keep the direction of a vector, but change it's distance.

The sample code gets the Vector from your hero to the end goal (the Red X), then makes it **1 meter** long with `Vector.normalize(goal)`. Then it uses `Vector.multiply(goal, 10)` to make it **10 meters**.

So far so good, but this will take your hero right past a yak!



So now, you need to use `Vector.subtract`, `Vector.normalize`, and `Vector.multiply` to get a 10m vector from the Yak to the hero:



Then add the vector from the yak to you, to the original vector to the goal:

That will give you a final moveToPos that takes you toward the goal, but also away from the yak!

Skating Away Solution

```
// Move to the red X mark while avoiding the yaks.
// use Vector.normalize(vector1) to create a vector in the same direction as vector1, but with a
    distance of 1
// use Vector.multiply(vector1, X) to create a vector in the same direction as vector1, but with its
    distance multiplied by X

// The point you want to get to.
var goalPoint = new Vector(78, 34);

while(true) {
    // This creates a vector that will move you 10 meters toward the goalPoint
    // First, create a vector from your hero to the goal point.
    var goal = Vector.subtract(goalPoint, hero.pos);
    // Then, normalize it into a 1m distance vector
    goal = Vector.normalize(goal);
    // Finally, multiply the 1m vector by 10, to get a 10m long vector.
    goal = Vector.multiply(goal, 10);

    // To avoid the yaks, if you get within 10 meters of a yak, you should vector away from it.
    var yak = hero.findNearest(hero.findEnemies());
    var distance = hero.distanceTo(yak);
    if(distance < 10) {
        // First, make a Vector from the yak to you
        // Now use Vector.normalize and Vector.multiply to make it 10m long
        // Once you have the 10m vector away from the yak, use Vector.add to add it to your goal vector!
        var toYak = Vector.subtract(hero.pos, yak.pos);
        toYak.normalize();
        toYak.multiply(10);
        goal = Vector.add(goal, toYak);
    }
    // Finally, determine where to move by adding your goal vector to your current position.
    var moveToPos = Vector.add(hero.pos, goal);
    hero.move(moveToPos);
}
```

#10. Guess My Number

Level Overview and Solutions

Intro

You have 10 tries to figure out which number the Riddler is thinking of.

He will tell you higher or lower, and you must divide and conquer this riddle!

Default Code

```
// You need to guess the number from 1 to 999 (0 < n < 1000).
// You have 10 attempts!
// For each attempt Riddler will say if the number is less or greater.
// If the guessed number is less than your try, then a munchkin ogre appears.
// Else a scout ogre appears.

var lowestPossible = 1;
var highestPossible = 999;

while (true) {
    // You need to defeat the enemy before the next attempt.
    var enemy = hero.findNearest(hero.findEnemies());
    if (enemy) {
        // "scout" is 'greater', "munchkin" is less.
        // Prepare for the next attempt and wipe out the ogre.

        hero.attack(enemy);
    }
    else {
        // Make your next attempt and say it.

        hero.say(42);
    }
}
```

Overview

Let's play the game. It's simple: Riddler thought of a number, for you to try to guess. If the number is less or greater than your guess, the Riddler will tell you higher or lower. Riddler is an infamous cheater, but with 10 attempts you can win anyway.

For each attempt an ogre appears . If it's a Scout (`ogre.type == 'scout'`), then the Riddler's number is less than your guess. If an ogre is a Munchkin (`ogre.type == 'munchkin'`), then the Riddler's number is greater than your guess.

You need to defeat the ogre before saying the next guess!

Guess My Number Solution

```
// You need to guess the number from 1 to 999 (0 < n < 1000).
// You have 10 attempts!
// For each attempt Riddler will say if the number is less or greater.
// If the guessed number is less than your try, then a munchkin ogre appears.
// Else a scout ogre appears.

var lowestPossible = 1;
var highestPossible = 999;

var lowBorder = lowestPossible - 1;
var highBorder  = highestPossible + 1;

while (true) {
    // You need to defeat the enemy before the next attempt.
    var enemy = hero.findNearest(hero.findEnemies());

    if (enemy) {
        // "scout" is 'greater', "munchkin" is less.
        // Prepare for the next attempt and wipe out the ogre.
        if (enemy.type === "scout") {
            lowBorder = guess;
        }
        else if (enemy.type === "munchkin") {
            highBorder = guess;
        }
        hero.attack(enemy);
    }
    else {
        // Make your next attempt and say it.
        var guess = Math.floor(lowBorder + (highBorder - lowBorder) / 2);
        hero.say(guess);
    }
}
```

#11. Form Up!

Level Overview and Solutions

Intro

Those soldiers before the gate are new recruits, and aren't trained. We should form up them by their height. Their 'health' property is proportional to their height.

Form up them from the smallest to the biggest (from left to right). Use `command` and predefined functions to make some formation from those recruits.

Default Code

```
// Those soldiers before the gate are new recruits, and aren't trained.
// We should order them by their height.
// Their 'health' property is proportional to their height.
// Form up them from the smallest to the biggest (from left to right).
// Command those 8 recruits to move on the marks in the right order.

// This function returns the coordinates of the i-th mark.
function markPos(i) {
    return {x: 12 + i * 8, y: 12};
}

// This function returns a list of the soldiers who should be sorted.
function findUnplacedSoldiers() {
    var friends = hero.findFriends();
    var result = [];
    for (var i = 0; i < friends.length; i++) {
        var friend = friends[i];
        // Recruits' actions are readable.
        if (friend.pos.y < 30 && friend.pos.y > 18 && friend.action === "idle") {
            result.push(friend);
        }
    }
    return result;
}

// This function should return the unit with the least health.
function minHealthUnit(units) {
    // Write this function to use it for the sorting.

}

// Soldiers who should be sorted.
var recruits = findUnplacedSoldiers();
var markIndex = 0;

// While there are soldiers who aren't in the position.
while (recruits.length) {
    hero.say(recruits.length + " recruits aren't in the positions.");
    // Find the recruit with the minimum health.

    // Command him to move to the mark (use 'markPos' function and 'markIndex').

    // Increase markIndex by 1.

    // Update the list of soldiers who should be sorted - 'recruits'.

}
```

Overview

In this level, we'll implement the selection sort algorithm (not-in-place). That algorithm is not effective, but simple.

You have an unsorted array. Create the new empty array, which will contain our sorted data.

```
var unsorted = [10, 12, 3, 1, 2, 6];
var sorted = [];
```

Next, we iterate the unsorted array and find the smallest element. Remove it from the first array and put in the end of the result array.

```
var smallest = minFunction(unsorted)
result.push(smallest)
unsorted.splice(unsorted.indexOf(smallest), 1)
```

Repeat the last step while the unsorted array isn't empty.

```
while (unsorted.length) {
    // Find the smallest and move it
    ...
}
```

For the current level **you don't need to remove elements** (soldiers) from the unsorted array, because soldiers "disappear" from the `recruits` array when you command them.

Don't forget to implement the function for searching of the weakest soldier. You've done it already in previous levels, so it shouldn't be hard.

Optional (Advanced) information:

The complexity of that algorithm is O(N^2), where N is the number of soldiers. For the worst case you iterate the array N times and for each step it has lengths: N, N-1,... 2, 1.

```
N * (N + (N-1) + ... + 3 + 2 + 1) == N * (N + 1) / 2 < N * N
```

Form Up! Solution

```
// Those soldiers before the gate are new recruits, and aren't trained.
// We should order them by their height.
// Their 'health' property is proportional to their height.
// Form up them from the smallest to the biggest (from left to right).
// Command those 8 recruits to move on the marks in the right order.

// This function returns the coordinates of the i-th mark.
function markPos(i) {
    return {x: 12 + i * 8, y: 12};
}

// This function returns a list of the soldiers who should be sorted.
function findUnplacedSoldiers() {
    var friends = hero.findFriends();
    var result = [];
    for (var i = 0; i < friends.length; i++) {
        var friend = friends[i];
        // Recruits' actions are readable.
        if (friend.pos.y < 30 && friend.pos.y > 18 && friend.action === "idle") {
            result.push(friend);
        }
    }
    return result;
}

// This function should return the unit with the least health.
function minHealthUnit(units) {
    // Write this function to use it for the sorting.
    var weakest;
    var minHealth = 9001;
    for (var u = 0; u < units.length; u++) {
        if (units[u].health < minHealth) {
            minHealth = units[u].health;
            weakest = units[u];
        }
    }
    return weakest;
}

// Soldiers who should be sorted.
var recruits = findUnplacedSoldiers();
var markIndex = 0;

// While there are soldiers who aren't in the position.
while (recruits.length) {
    hero.say(recruits.length + " recruits aren't in the positions.");
    // Find the recruit with the minimum health.
    var weakestRecruit = minHealthUnit(recruits);
    // Command him to move to the mark (use 'markPos' function and 'markIndex').
    hero.command(weakestRecruit, "move", markPos(markIndex));
    // Increase markIndex by 1.
    markIndex++;
    // Update the list of soldiers who should be sorted - 'recruits'.
    recruits = findUnplacedSoldiers();
}
```

#12. Match Cord

Level Overview and Solutions

Intro

Ogres have prepared a minefield to protect their Cheiftain.

We can use the domino effect to our advantage to blow a match cord to the Cheftain!

Default Code

```
// Ogres mined the field to protect their Chieftain.
// But we can use the "domino" effect get our target.
// The scout has prepared the map of the minefield.
// All mines are placed the same distance apart.
// The map is an array of strings, where "x" is a mine and "." is nothing.
// The first row in the array is the row nearest to the hero.

// The map and helpful constants are listed below.
var fieldMap = hero.findFriends()[0].getMap();

var mine = "x";
var empty = ".";
var mineDistance = 5;
var firstXPos = 15;
var firstYPos = 40;

// Find which starting mine connects to the ogre Chieftain.


var resultColumn = 21; // Δ Change this to your actual result!

hero.say("I think it's column number: " + resultColumn);
hero.moveXY(resultColumn * mineDistance + firstXPos, firstYPos);
```

Overview

You are given a map of the minefield as an array of strings. "x" is a mine while "." is an empty spot. All mines are 5 meters apart. When a mine blows, it will chain react with all mines adjacent to itself (horizontally and vertically but not diagonally.)

The first row of the map is the row closest to the hero, and the first column is the left edge.

When you find which one will get the Cheiftain, step on it and blow her sky high!

Match Cord Solution

```
// Ogres mined the field to protect their Chieftain.
// But we can use the "domino" effect get our target.
// The scout has prepared the map of the minefield.
// All mines are placed the same distance apart.
// The map is an array of strings, where "x" is a mine and "." is nothing.
// The first row in the array is the row nearest to the hero.

// The map and helpful constants are listed below.
var fieldMap = hero.findFriends()[0].getMap();
var height = fieldMap.length;
var width = fieldMap[0].length;

var mine = "x";
var empty = ".";
var mineDistance = 5;
var firstXPos = 15;
var firstYPos = 40;

// Find which starting mine connects to the ogre Chieftain.
var stack = [];
var visited = [];

// Find which starting mine connects to the ogre Chieftain.
for (var j = 0; j < width; j++) {
    if (fieldMap[height - 1][j] == "x") {
        stack.push([height - 1, j]);
    }
}

// Possible neighbours (shifts)
var neighbourDiffs = [[-1, 0], [1, 0], [0, -1], [0, 1]];

var resultColumn = 1;

while (stack.length) {
    var state = stack.pop();
    var row = state[0];
    var col = state[1];

    // String representation of the current position for 'visited' search
    var strPos = row + "-" + col;
    if (row == 0) {
        resultColumn = col;
        break;
    }
    // If we checked the current position
    if (visited.indexOf(strPos) != -1) {
        continue;
    }
    visited.push(strPos);

    for (var d = 0; d < neighbourDiffs.length; d++) {
        var diff = neighbourDiffs[d];
        var neighRow = row + diff[0];
        var neighCol = col + diff[1];
        // Check if it's outside the map
        if (neighRow < 0 || neighCol < 0 || neighRow >= height || neighCol >= width) {
            continue;
        }
        // Don't put in stack the checked nodes
        var strNeighPos = neighRow + "-" + neighCol;
        if (visited.indexOf(strNeighPos) != -1){
            continue;
        }
        if (fieldMap[neighRow][neighCol] == "x") {
            stack.push([neighRow, neighCol]);
        }
    }
}

hero.say("I think it's column number: " + resultColumn);
hero.moveXY(resultColumn * mineDistance + firstXPos, firstYPos);
```

#13. Golden Choice

Level Overview and Solutions

Intro

The gate keeper will tell you how much gold you need to collect.

Move in the direction of the exit.

After each coin choose one of the two (or one for edges) nearest coins in the next row.

Don't stop and move exactly to the next coin. One wrong step and deadly beams will burn you.

Default Code

```
// You must collect the required amount of gold.
// The gate keeper will tell you how much you need.
// Always move in the direction of the exit.
// For each row you can take only one coin.
// Choose only one from the nearest coins in the next row.

// Distance between rows and coins.
var distanceX = 4;
var distanceY = 6;
var zeroPoint = {x: 14, y: 14};
var coinLines = 10;

function makeGoldMap(coins) {
    var template = [];
    for (var i = 0; i < coinLines; i++) {
        template[i] = [];
        for (var j = 0; j < 2 * coinLines - 1; j++) {
            template[i][j] = 0;
        }
    }
    for (var c = 0; c < coins.length; c++) {
        var row = Math.floor((coins[c].pos.y - zeroPoint.y) / distanceY);
        var col = Math.floor((coins[c].pos.x - zeroPoint.x) / distanceX);
        template[row][col] = coins[c].value;
    }
    return template;
}

// Prepare the gold map. It looks like:
// [[1, 0, 9, 0, 4],
//  [0, 1, 0, 9, 0],
//  [8, 0, 2, 0, 9]]
var goldMap = makeGoldMap(hero.findItems());

// Find your path.
```

Overview

The algorithm required to beat this level is advanced! Be prepared to work for your victory.

1. Iterate across an entire row of coins and store their values
2. Check the next row, and sum each initial value with each neighbor.
3. Repeat until the last row.
4. Check which path has the highest value.
5. Navigate that path!

Golden Choice Solution

```
// You must collect the required amount of gold.
// The gate keeper will tell you how much you need.
// Always move in the direction of the exit.
// For each row you can take only one coin.
// Choose only one from the nearest coins in the next row.

// Distance between rows and coins.
var distanceX = 4;
var distanceY = 6;
var zeroPoint = {x: 14, y: 14};
var coinLines = 10;

function makeGoldMap(coins) {
    var template = [];
    for (var i = 0; i < coinLines; i++) {
        template[i] = [];
        for (var j = 0; j < 2 * coinLines - 1; j++) {
            template[i][j] = 0;
        }
    }
    for (var c = 0; c < coins.length; c++) {
        var row = Math.floor((coins[c].pos.y - zeroPoint.y) / distanceY);
        var col = Math.floor((coins[c].pos.x - zeroPoint.x) / distanceX);
        template[row][col] = coins[c].value;
    }
    return template;
}

function processMap(goldMap) {
    for (var i = goldMap.length - 2; i >= 0; i--) {
        var line = goldMap[i];
        for (var j = 0; j < line.length; j++) {
            if (line[j] === 0) continue;
            if (j === 0) {
                goldMap[i][j] += goldMap[i+1][j+1];
            } else if (j === line.length - 1) {
                goldMap[i][j] += goldMap[i+1][j-1];
            } else {
                goldMap[i][j] += Math.max(goldMap[i+1][j-1], goldMap[i+1][j+1]);
            }
        }
    }
    return goldMap;
}

function chooseStartCol(goldMap) {
    var bestCol = 0;
    var bestValue = 0;
    for (var i = 0; i < goldMap[0].length; i++) {
        if (goldMap[0][i] > bestValue) {
            bestValue = goldMap[0][i];
            bestCol = i;
        }
    }
    return bestCol;
}

function findPath(goldMap, startCol) {
    var path = [];
    var col = startCol;
    for (var i = 0; i < goldMap.length - 1; i++) {
        if (col === 0) {
            path.push("R");
            col += 1;
        } else if (col === goldMap[i].length - 1) {
            path.push("L");
            col -= 1;
        } else {
            if (goldMap[i+1][col-1] < goldMap[i+1][col+1]) {
                path.push("R");
                col += 1;
            } else {
                path.push("L");
                col -= 1;
            }
        }
    }
    return path;
}

// Prepare the gold map. It looks like:
// [[1, 0, 9, 0, 4],
//  [0, 1, 0, 9, 0],
//  [8, 0, 2, 0, 9]]
var myMap = makeGoldMap(hero.findItems());

var processedMap = processMap(myMap);

// Find your path.
var startColumn = chooseStartCol(processedMap);

hero.moveXY(startColumn * distanceX + zeroPoint.x, hero.pos.y);
```

```
hero.moveXY(startColumn * distanceX + zeroPoint.x, zeroPoint.y);

var route = findPath(processedMap, startColumn);

for (var p = 0; p < route.length; p++) {
    if (route[p] === "L") {
        hero.moveXY(hero.pos.x - distanceX, hero.pos.y + distanceY);
    } else {
        hero.moveXY(hero.pos.x + distanceX, hero.pos.y + distanceY);
    }
}

hero.moveXY(hero.pos.x, hero.pos.y + 5);
```

#14. Fragile Maze

Level Overview and Solutions

Intro

You must escape from the ice maze. Some doors are fragile and they will open just from the sound of your steps.

Explore the maze room by room and find the path to the exit.

Default Code

```
// Escape from the maze!
// Some doors are blocked, some open when you are near them.

var distanceBetweenRooms = 16;
var startRoom = {x: 18, y: 19};
```

Overview

Use the `isPathClear` function to find which doors are open as you navigate the maze.

Fragile Maze Solution

```
// Escape from the maze!
// Some doors are blocked, some open when you are near them.
// "Left hand rule" algorithm http://bit.ly/2eCpzWO

var distanceBetweenRooms = 16;

var DIRECTIONS = ["left", "up", "right", "down"];
var LEFT_FOR = {
    "left": "down",
    "up": "left",
    "down": "right",
    "right": "up"
};


function findNextDirection(current) {
    return DIRECTIONS[(DIRECTIONS.indexOf(current) + 1) % 4];
}

function pointInDirection(direction, pos) {
    if (direction == "left") {
        return Vector(pos.x - distanceBetweenRooms, pos.y);
    }
    else if (direction == "right") {
        return Vector(pos.x + distanceBetweenRooms, pos.y);
    }
    else if (direction == "down") {
        return Vector(pos.x, pos.y - distanceBetweenRooms);
    }
    else if (direction == "up") {
        return Vector(pos.x, pos.y + distanceBetweenRooms);
    }
}

var prev = "up";

while(true) {
    var tryDir = LEFT_FOR[prev];
    while(true) {
        var tryPoint = pointInDirection(tryDir, hero.pos);
        if (hero.isPathClear(hero.pos, tryPoint)) {
            prev = tryDir;
            // Use 2x moveXY to avoid sliding.
            hero.moveXY(tryPoint.x, tryPoint.y);
            hero.moveXY(tryPoint.x, tryPoint.y);
            hero.wait(0.1);
            break;
        }
        tryDir = findNextDirection(tryDir);
    }
}
```

#15. Treasured in Ice

Level Overview and Solutions

Intro

You must escape from the ice maz--... Wait, wait wait, I've said this all before.

This is the last time, but don't forget the goal of your adventure - the treasure.

Go through the maze, find the treasure, and return to the exit.

Step on the red cross mark to escape from the maze, but don't leave the treasure behind.

Some doors are fragile and will open by the sound of your steps.

Explore the maze, room by room, to find the chest full of gems.

Default Code

```
// Find the treasure inside the maze.
// When you get the treasure, move to the exit.
// The exit is marked by the red cross. The level ends when you step on the mark.
// Some doors are blocked, some open when you are near them.

var exitPosition = {x: 150, y: 120};
var distanceBetweenRooms = 16;
var zeroShift = {x: 10, y: 10};
```

Overview

This is a more complicated version for navigating mazes! Explore the rooms using `isPathClear` and keep track of which rooms you've entered to ensure no doubling-back.

Treasured in Ice Solution

```
var inSearch = true;

var width = 19,
    height = 15;

function coorConvert(i, dist) {
    return i * 8 + 10;
}

var wall = "X",
    unknown = "?",
    empty = ".";

// Clear map of the maze.
var maze = [];
// left border
for (var i = 0; i < height; i++) maze.push([wall]);
// top border
for (var j = 1; j < width - 1; j++) maze[0].push(wall);
for (i = 1; i < height-1; i++) {
    if (i % 2) {
        for (j = 1; j < width - 1; j++) maze[i].push(unknown);
    } else {
        for (j = 1; j < width - 1; j++) maze[i].push(j % 2 ? unknown : wall);
    }
}
// bottom border
for (j = 1; j < width - 1; j++) maze[height-1].push(wall);
// right border
for (i = 0; i < height; i++) maze.push([wall]);

// Start position
hero.row = 13;
hero.col = 17;
// Exit
var exitRow = 13,
    exitCol = 17;

// function for find positions in various directions
function east() {
    return {x: coorConvert(hero.col - 2), y: coorConvert(hero.row), row: hero.row, col: hero.col - 2};
}
function west() {
    return {x: coorConvert(hero.col + 2), y: coorConvert(hero.row), row: hero.row, col: hero.col + 2};
}
function south() {
    return {x: coorConvert(hero.col), y: coorConvert(hero.row - 2), row: hero.row - 2, col: hero.col};
}
function north() {
    return {x: coorConvert(hero.col), y: coorConvert(hero.row + 2), row: hero.row + 2, col: hero.col};
}
function exit() {
    return {x: 150, y: 120};
}

// Move hero in the direction and update property positions
function moveDir(direction) {
    var newPosFunc = {
        "N": north,
        "S": south,
        "E": east,
        "W": west,
        "!": exit
    }[direction];
    var newPos = newPosFunc();
    hero.moveXY(newPos.x, newPos.y);
    if (direction === "!") {
        inSearch = false;
    }
    hero.row = newPos.row;
    hero.col = newPos.col;
}

// check near door and update map
function checkPaths(cMap) {
    cMap[hero.row][hero.col] = ".";
    var dirs = [[1, 0, north], [-1, 0, south], [0, 1, west], [0, -1, east]];
    for (var d = 0, D = dirs.length; d < D; d++) {
        var dx = dirs[d][0],
            dy = dirs[d][1],
            dPosFunc = dirs[d][2];
        if (cMap[hero.row+dx][hero.col+dy] === "?" && hero.isPathClear(hero.pos, dPosFunc())) {
            cMap[hero.row+dx][hero.col+dy] = ".";
        }
    }
    return cMap;
}

function findPath(m, needExit) {
    var stack = [[hero.row, hero.col, ""]];
    var visited = [];
    while (stack.length > 0) {
```

```
        var current = stack.shift();
        var row = current[0],
            col = current[1],
            path = current[2];
        var strPos = row + '-' + col;
        if (visited.indexOf(strPos) !== -1) {
            continue;
        }
        visited.push(strPos);

        if (!needExit && m[row][col] === "?") {
            return path;
        }
        if (needExit && row === exitRow && col === exitCol) {
            return path + "!";
        }
        if (m[row-1][col] === ".") {
            stack.push([row-2, col, path + "S"]);
        }
        if (m[row+1][col] === ".") {
            stack.push([row+2, col, path + "N"]);
        }
        if (m[row][col+1] === ".") {
            stack.push([row, col+2, path + "W"]);
        }
        if (m[row][col-1] === ".") {
            stack.push([row, col-2, path + "E"]);
        }
    }
}


var path = "";
var treasureFound = false;
while (inSearch) {
    maze = checkPaths(maze);
    if (hero.gold > 0) {
        treasureFound = true;
        inSearch = false;
    }
    path = findPath(maze, treasureFound);
    if (!path) {
        hero.say("Arrrr!");
    } else {
        for (var p = 0; p < path.length; p++) {
            moveDir(path[p]);
        }
    }
```

#16. Broken Circles

Level Overview and Solutions

Intro

Six circles of traps are gathering round the treasure chest. Each circle has an opening. Can you find a path to the inner circle?

Flying guards are keeping watch for the following rule violations:

- No jumping
- No summoning
- No flags
- No magic

Default Code

```
// Collect the treasure.
// No flags.
// No summons.
// No jumps.
// No magic.
// Infringes will be punished.

// Useful constants.
var trapAttackRange = 3;
var radiusStep = 10;
var center = {x: 68, y: 68};

// Go, Go, GO!
```

Overview

`findHazards()` returns an array of all `"fire-traps"`. Use it to navigate this polar maze!

Broken Circles Solution

```
// Collect the treasure.
// No flags.
// No summons.
// No jumps.
// No magic.
// Infringes will be punished.

// Useful constants.
var r = 10;
var diff = 0.5;
var threshold = 5.8;
var S = 3;
var circle = 5;
var cx = 68,
    cy = 68;

function prepareMap() {
    var map = [];
    for (var y = 0; y < 135; y++) {
        var row = [];
        for (var x = 0; x < 160; x++) {
            row.push(0);
        }
        map.push(row);
    }
    var hazards = hero.findHazards();
    for (var i = 0; i < hazards.length; i++) {
        var hx = Math.round(hazards[i].pos.x);
        var hy = Math.round(hazards[i].pos.y);
        map[hx][hy] = 1;
    }
    return map;
}

var rad = Math.PI * 2 / 360;

hero.moveXY(128 - r / 2 - diff, 68);
var angle = 0;
while(true) {
    if (!circle) {
        hero.moveXY(68, 67);
        break;
    }
    var R = r * circle + r / 2 - diff;
    var iR = R - r;
    while (true) {
        angle = (angle + 2) % 360;
        var x = cx + Math.cos(angle * rad) * R;
        var y = cy + Math.sin(angle * rad) * R;
        var ix = cx + Math.cos(angle * rad) * iR;
        var iy = cy + Math.sin(angle * rad) * iR;
        hero.moveXY(x, y);
        var nearest = hero.findNearest(hero.findHazards());

        if (hero.distanceTo(nearest) >= threshold) {
            hero.moveXY(ix, iy);
            circle--;
            break;
        }
    }
}

// Go, Go, GO!
```