How much toast

Tony Beaumont

3 June 2025

Problem Solving: How much Toast?

Introduction

This series of problem solving exercises are based on problems found on Codewars https://www.codewars.com/ or Hackerank https://www.hackerrank.com/dashboard.

What you should already know

This is the first problem in the course. There is no need for any prior knowledge.

What you'll learn

- How to use **abstraction** to identify the key elements of the problem
- How to use **decomposition** to break the problem down into smaller, easier problems.
- How to manage **complexity**.

What you'll need

No specific tools are required, although you will need to make notes so either:

- pencil and paper
- text editor on your laptop.

What you'll do

You will be presented with a problem. You will learn how to understand the problem and break the problem down into steps that help you work towards a solution.

Problem Description

Todays problem can be found here: https://www.codewars.com/kata/5834fec22fb0ba7d080000e8.

After the class finishes, you should submit a solution to the problem using the link above.

Story

You are going to make toast fast, you think that you should make multiple pieces of toast at once. So, you try to make 6 pieces of toast at once using all the toasters available.

Problem

You forgot to count the number of toast you put in there, you don't know if you put exactly six pieces of toast into the toasters.

Define a function that counts how many more (or less) pieces of toast you need in the toasters. Even though you need more or less, the number will still be positive, not negative.

Examples:

You must return the number of toast the you need to put in (or to take out). In case of `5` you can still put `1` toast in:

```
5 --> 1
```

And in case of `12` you need `6` toasts less (but **not** `-6`):

```
12 --> 6
```

Code Template

```
function sixToast(num) {
  // you code goes below this line

}
```

Understanding the language used in the problem description

The first step in solving any problem is understanding that problem.

1. Are there any words or phrases in the problem description that you don't understand? If so, write them down and find out what they mean.

When you are sure that you know the meaning of all the words and phrases in the problem desicription, move on to the next step.

Identify what is important

This next step is called using **abstraction**. That is just a fancy word for the process of filtering out - ignoring - the characteristics of problems that are not needed in order to concentrate on those that are needed. It is also the filtering out of specific details. From this, we can create an idea of what is to be solved (see https://www.bbc.co.uk/bitesize/guides/zmhpfcw/revision/1 for more information).

1. Identify the parts of the problem description that are important for our solution and the parts we can just ignore. Here are some hints to guide you:
    1. Is the toast important?
    2. Are the toasters important?
    3. Are there any input values we need to use, if so what?
    4. What is the output value we are being asked to create?
    5. Are there any values apart from the input and output that are important?
    6. Is there anything that might be difficult?

Make a note of each important piece of information you have identified that is important to the problem description.

Discuss your solution to this step as a group. Does everyone have the same list of important elements or are there differences? Resolve your differences so everyone agrees what is important.

Important characteristics of the problem

When we solve problems with code, there is often **input** that is required. Input is data that changes depending on the circumstances.

For each set of input, our solution should `return` an **output** value. The output is what you might describe as the "correct answer given the input".

- We have an input value which is referred to in the code template as `num`
- We are comparing the input to the value '6'.
- Our output is a number which tells by how much the value of `num` is bigger than, or smaller than, `6`.
- The output is always a positive number. Actually this aspect is the "difficult" part of this problem.

Understand the examples

The problem description almost always has some examples that show various different problem **inputs** and the **output** that goes with each input.

1. Look at the examples that are provided. Can you explain how the output was generated from the input?
2. Make a new version of each example that has different input and provide the output that should result from that input.

Explanation of the examples.

Example one

The first example is this one:

```
5 --> 1
```

The **input** value that we are calling `num`, is `5` and the **output** value is `1`. We can compute the output by calculating the result of `6 - num` which becomes `6 - 5` which gives the output value `1`.

Example two

The second example is:

```
12 --> 6
```

The **input** value that we are calling `num`, is `12` and the **output** value is `6`. If we follow the same process as the first example we get `6 - num` which becomes `6 - 12` which gives us the (incorrect) result `-6`. To get to the correct answer we need to turn `-6` into `6` (multiply by `-1`).

Two more examples

- This is another version of the first example:

  ```
  2 --> 4
  ```

- Here is another version of the second problem:

  ```
  9 --> 3
  ```

Break down the problem

It often helps to **decompose** the problem into smaller problems. We call this step **decomposition**.

1. From your analysis of the examples, can you spot TWO different cases or sub-problems within this problem?
2. Discuss the two different cases within your class. Is one case easier than the other?

The two different cases

- In the first example, the **input** value `num` is less than `6`. The solution is obtained by the formula `6 - num`.
- In the second example, the **input** value `num` is greater than `6`. The solution is more complicated because we either need to calculate `num - 6` OR calculate `6 - num` and multiply the result by `-1`.

The second case is "more difficult" in that we need to do something different to the "easier" first case.

Are there any other cases? What if `num` is equal to `6`?

Starting to develop a solution

It is acceptable (and often important) to solve the decomposed smaller problems one by one. In this first step we will develop a solution that works ONLY in the case that `num` is less than `6`.

It is important to note that up until now there has been no programming code written or even mentioned. There was a bit of mathematics involved in calculating the result.

When we develop a solution we often use **pseudo-code**. This is a cross between program code where that code is easy to write and an English description when the step is not easy to write in one program code statement.

1. Here is a partly finished pseudo-code solution to the easier part of the problem:

   ```
   return __________
   ```

   Fill in the gap in the example above. Don't worry about getting the code correct, just fill in what the step needs to achieve using English.

A plan for the solution

Here is the pseudo-code with the gaps filled in:

```
return 6 - num
```

We could put this solution into the training step on Codewars and we may see that sometimes it produces the correct output and passes a test and other cases it produces the wrong output and fails a test. That behaviour is ok, we expected that.

Improving the solution

To improve our solution we need to include a check in our solution to identify whether we are solving the simple case when `num` is less than `6`, or not.

1. Write a new version of our plan that includes an `if` statement that checks whether `num` is less than 6 and when that condition is true returns the value of `num - 6`.

Improved solution

Here is some improved pseudo-code. **Note** that `is less than` is not valid Javascript but while we are beginners, we won't worry yet about the correct Javascript code for the statement :

```
if (num is less than 6) {
    return 6 - num
}
```

We can use this new version in the Codewars training screen after translating `is less than` into correct Javascript. Although the solution is improved, it may not pass more tests.

Solving the second case

Our solution so far detects the easier case and returns the correct value for that case. Here we develop a solution for the alternative case.

1. Complete the following:

   ```
   if (num is less than 6) {
       return 6 - num
   } else {
       const difference = 6 - num;
       return ________________________
   }
   ```

A plan for the second case

Here is our complete plan for a solution in pseudo-code:

```
if (num is less than 6) {
    return 6 - num
} else {
    const difference = 6 - num;
```

```
    return difference * -1;
}
```

We will stop here.

Later, in your own time:

- Turn this plan into actual code and test your solution code here: https://www.codewars.com/kata/5834fec22fb0ba7d080000e8
- Do some research into the function `Math.abs` and try to come up with a neater solution.
- **Challenge**: Can you use `Math.abs` to come up with a version that doesn't have an `if` statement?

Summary

There were a lot of steps into solving this problem. They were important.

- Abstraction was useful to identify the problem. It turns out toast and toasters are irrelevant.
- Decomposition allowed us to develop a pseudo-code solution in a sequence of steps.
- Our first "solution" was a simple one that only gives the correct solution in some case. Creating an interim partial solution like this is fine.
- Improving a partial solution often means adding complexity. If we add that complexity slowly, step by step, the complexity is easier to manage.
- We can write pseudo-code to plan our actual solution without worrying about correct code.
- Do not be tempted to add complexity too quickly.
- You can always improve a simple solution that gives correct answers sometimes. It is more difficult to fix a more complex and broken solution that doesn't produce any output.
- You might eventually be able to rewrite your solution to create a neat one line solution, but your aim must always be to create a **working** solution rather than a "smart one line" solution.