Course Overview

The chief goal of this course is to help you develop **computational thinking**, a way of looking at problems that plays an important role in computer science. Research has shown that people with strong computational thinking skills make better problem solvers, even in fields unrelated to computers.

The secondary goal is to give an introduction to the three major areas of computer science:

- **Systems:** The art and science of making physical devices effective engines for abstract computation.
- **Theory:** The art and mathematics of solving abstract problems with abstract computation.
- **Applications:** The art and science of effectively framing concrete problems in abstract terms.

We will cover all three in the context of web scripting. By the end of the class, students should demonstrate

- a working knowledge of imperative programming, specifically in-browser JavaScript (systems),
- an ability to create and formally communicate systematic problem-solving strategies (theory), and
- the use of such knowledge and skills to solve practical problems (applications).

Lesson Overview

Most introductory computer science courses begin with systems, applications, or a mix of both. However, it's not possible to teach much of either without—at least implicitly—expecting some proficiency in theory. So this approach tends to leave students having to pick up foundational theory skills on their own, which, in turn, disadvantages students with primarily non-computational styles of thinking. Instead, we'll spend our first unit specifically on foundational theory, and later units will work towards to concrete.

We'll start with the ways that abstract problems factor into subproblems and what that factoring tells us about the structure of their solutions. There's the simplest case, where a problem can't be factored:

- trivial problems

plus four others where it can:

- into alternative subproblems,
- into parallel subproblems,
- into sequential subproblems, and
- into quotient subproblems,

Trivial Problems

Computers are built to solve certain kinds of problems in (effectively) one step; there's usually no need to factor these problems into smaller pieces. Here's an example:

> There are two **variables** (pieces of computer memory), `original` and `duplicate`, each holding one number. The goal is to replace the number held by `duplicate` with the one held by `original`.

For our purposes, the computer can just be told to solve this problem; the solution is called **assignment**.

When computer scientists are writing an algorithm just for humans to read, they typically use a format called **pseudocode**. Here is how the assignment of `original` to `duplicate` might look in pseudocode:

> **let** `duplicate` ← `original`

The full list of trivial problems depends on how the computer and the programming language were designed. Fortunately, most designs only have a handful that are easy to memorize. We'll see a few others as we proceed, but assignment is by far the most important.

Factoring into Alternative Subproblems

Consider a slightly harder problem:

> `first_number`, `second_number`, and `result` each hold a number. The goal is to replace the number held by `result` with the maximum of the numbers held by `first_number` and `second_number`.

Replacing with a maximum, while fairly simple, is not normally among a computer's basic operations; we have to build the process out of smaller pieces.

The basic operation we do have is to replace one value with another. It's clear that `result`'s value is the one that we want to replace, but which value should we replace it with? It depends on which is larger. Whenever we find ourselves answering a question about what the computer should do with "it depends", we have a case of alternative subproblems.

There are three possibilities here: `first_number` is less than `second_number`, `first_number` is equal to `second_number`, or `first_number` is greater than `second_number`. In case of equality, it doesn't matter which value we copy, so let's combine the last two possibilities, factoring the problem this way:

- Finding the maximum
  Splits into alternative subproblems:
  - Finding the maximum when `first_number` is less than `second_number`
  - Finding the maximum when `first_number` is greater than or equal to `second_number`

Notice that "finding the maximum when `first_number` is less than `second_number`" is just a fancy way of saying "replace the value of `maximum` with the value of `second_number`", which we know how to do. Likewise for the other case.

However, there are still two steps that have to happen but are not part of either subproblem: determining whether `first_number` is less than `second_number`, and choosing between the two possible subsolutions. Fortunately, both of these things are among computers' basic operations.

The pseudocode for our algorithm would look like this:

```
if first_number < second_number then
 |  let maximum ← second_number
else
 |  let maximum ← first_number
end
```


The part "`first_number` < `second_number`" is the first step; it computes the value `true` if `first_number` is less than `second_number`, `false` otherwise. (The values `true` and `false` are called **booleans**.) The pattern

if ⟨condition...⟩ **then**
| ⟨Case I...⟩
else
| ⟨Case II...⟩
end

is the second step and means that the computer should follow the steps in Case I and ignore Case II if the condition evaluates to `true`, but do the opposite if the condition is `false`.

Factoring into Parallel Subproblems

Alternatives are one way that we can have subproblems whose solutions do not depend on each other; parallel subproblems are the other. The difference lies in the number of subproblems that must be solved: with alternatives, the algorithm must solve one; with parallel subproblems, it must solve all.

A good way to recognize parallel subproblems is to ask whether the desired result can be split into two or more unrelated parts. If so, each part corresponds to a subproblem.

Here's an example:

> `first_number` and `second_number` each hold an input. `minimum` should, after the algorithm runs, hold their minimum, whereas `maximum` should hold their maximum.

Notice that the desired result comes in two parts, and computing the minimum has nothing to do with computing the maximum. Our problem factors this way:

- Finding the minimum and maximum
  Splits into parallel subproblems:
  - Finding the minimum
  - Finding the maximum

We already know how to tell a computer to solve these subproblems. All that remains is to tell it to do one first and the other second. (Sometimes the order will matter, but it doesn't here, and, in fact, most computers could do both parts at the same time. However, browsers make web scripts specify an order, so we'll also do that in our examples.) In pseudocode we write the consecutive tasks one after the other:

if `first_number` < `second_number` **then**
| **let** `minimum` ← `first_number`
else
| **let** `minimum` ← `second_number`
end
if `first_number` < `second_number` **then**
| **let** `maximum` ← `second_number`
else
| **let** `maximum` ← `first_number`
end

(You might notice that there's another way to factor this problem. See the example question on this lesson's worksheet.)

Factoring into Sequential Subproblems

Sequential subproblems are like parallel subproblems in that all of them must be solved. However, order matters with sequential subproblems. The first subproblem starts with only the data available to the original problem and asks for some additional useful data. The second subproblem assumes that this additional data has been computed and asks for even more useful data. The pattern continues until the last subproblem, whose answer is the desired result.

An example:

> `first_number` and `second_number` each hold an input, the two of which the algorithm should put in ascending order. `minimum` and `maximum` are available as temporary storage.

Notice how, as the problem wording suggestions, sorting two values is easier when we know which is the minimum and which is the maximum. So we can solve this problem in two steps, one that computes the extrema to make the problem easier and one that gives the sorted order with the help of this new data:

- Sorting two numbers
  Splits into sequential subproblems:
  - Finding the minimum and maximum
  - Replacing `first_number` and `second_number` with the minimum and maximum, respectively

The resulting algorithm:

```
if first_number < second_number then
 | let minimum ← first_number
else
 | let minimum ← second_number
end
if first_number < second_number then
 | let maximum ← second_number
else
 | let maximum ← first_number
end
let first_number ← minimum
let second_number ← maximum
```

(Again, you can find other factorings, some of which will produce shorter pseudocode.)

Factoring into Quotient Subproblems

Quotient subproblems are a common special case of sequential subproblems, one we treat separately. With quotient subproblems, each intermediate step's result is the answer to a smaller or simpler version of the overarching problem.

A good way to recognize quotient subproblems is to ask whether the problem would be easier if you could ignore or change part of the input. If so, the first subproblem is the one ignoring as much of the input as possible or changing the input the most, the second ignores or changes almost as much, etc.

Consider this example:

> `first_number`, `second_number`, and `third_number` each hold a number. `maximum` should, after the algorithm runs, hold their maximum.

If we take away one of the inputs, `third_number` say, we still have a maximum-finding problem, but a smaller one that we already know how to solve. Furthermore, knowing the maximum of `first_number` and `second_number` makes it easier to find the maximum of all three.

The consequent factoring, where we make sure to note how the subproblems are related:

- Finding the maximum of three numbers
  Splits into quotient subproblems:
  - Finding the maximum of the first two numbers
  - Finding the maximum of the first three numbers
    by finding the maximum of the previous result and the third number

And the pseudocode:

```
if first_number < second_number then
 |  let maximum ← second_number
else
 |  let maximum ← first_number
end
if maximum < third_number then
 |  let maximum ← third_number
else
 |
end
```

By convention, an "else" part is usually left out when it doesn't do anything:

```
if first_number < second_number then
 |  let maximum ← second_number
else
 |  let maximum ← first_number
end
if maximum < third_number then
 |  let maximum ← third_number
end
```

Homework

See Worksheets 1 and 2.