CS1010 Laboratory 03

CS1010 Coding Style, Assert Library, Exercise 2

Zhang Puyu

Group BD07

September 12, 2024

Plan of the Day

CS1010 Code Style

- It is a common practice for professional programmers to follow a certain set of **style conventions** when coding.

CS1010 Code Style

- It is a common practice for professional programmers to follow a certain set of **style conventions** when coding.
- The purpose of doing so is to standardise the structure, logical flow, naming conventions, etc. of the program so as to **increase code readability**.

CS1010 Code Style

- It is a common practice for professional programmers to follow a certain set of **style conventions** when coding.
- The purpose of doing so is to standardise the structure, logical flow, naming conventions, etc. of the program so as to **increase code readability**.
- You should read the following (very important) items on your own time:
  1. **BANNED** syntax: `https://nus-cs1010.github.io/2425-s1/guides/c-in-cs1010.html#banned-in-cs1010`.
  2. **Discouraged** syntax: `https://nus-cs1010.github.io/2425-s1/guides/c-in-cs1010.html#discouraged-in-cs1010`.
  3. **Good** practices and general guide: `https://nus-cs1010.github.io/2425-s1/guides/style.html`.

Exercise 1 Review: General Remarks

- Improve conciseness of code:

Exercise 1 Review: General Remarks

- Improve conciseness of code:
  1. Removing redundant parentheses. Example:

     ```
     if (((a + b > c) && (a + c > b) && (b + c > a)))
     ```
     is the same as
     ```
     if (a + b > c && a + c > b && b + c > a)
     ```

Exercise 1 Review: General Remarks

- Improve conciseness of code:
  1. Removing redundant parentheses. Example:
     `if (((a + b > c) && (a + c > b) && (b + c > a)))`
     is the same as
     `if (a + b > c && a + c > b && b + c > a)`
  2. For boolean functions with only one conditional judgment, you can consider returning the boolean expression directly.

Exercise 1 Review: General Remarks

- Improve conciseness of code:
  1. Removing redundant parentheses. Example:
     `if (((a + b > c) && (a + c > b) && (b + c > a)))`
     is the same as
     `if (a + b > c && a + c > b && b + c > a)`
  2. For boolean functions with only one conditional judgment, you can consider returning the boolean expression directly.
  3. Avoid unnecessary nesting of `if` statements — some can just be connected with &&.
- Various issues with coding style:
  - Declare but not initialise a variable.
  - Declare but not implement a function immediately.
  - Inconsistent opening braces style.

Exercise 1 Review: An Example


```
bool is_ok(long x, long y) {
    if ((((x/y+10)*2)+(x*y))>100) {
        if(x>0) {
            return true;}
    } else {
        return false;}}
```

Exercise 1 Review: An Example


```
bool is_ok(long x, long y) {
    if ((x / y + 10) * 2 + x * y > 100) {
        if (x > 0) {
            return true;
        }
    } else {
        return false;
    }
}
```

Exercise 1 Review: An Example


```c
bool is_ok(long x, long y) {
    return (x / y + 10) * 2 + x * y > 100 && x > 0;
}
```

A Sequal to Our Debugging Techniques

- **Core skill as a programmer: Ask the right questions.**

A Sequal to Our Debugging Techniques

- **Core skill as a programmer: Ask the right questions**.
- Saying things like "my code gives the wrong answer" helps little when trying to seek debugging advice from others.

A Sequal to Our Debugging Techniques

- **Core skill as a programmer: Ask the right questions**.
- Saying things like "my code gives the wrong answer" helps little when trying to seek debugging advice from others.
- A good framework for reporting a bug:
  1. Describe the task you intend the code to do.

A Sequal to Our Debugging Techniques

- **Core skill as a programmer: Ask the right questions**.
- Saying things like "my code gives the wrong answer" helps little when trying to seek debugging advice from others.
- A good framework for reporting a bug:
  1. Describe the task you intend the code to do.
  2. Describe the test case you used and other relevant constraints about the input.

A Sequal to Our Debugging Techniques

- **Core skill as a programmer: Ask the right questions.**
- Saying things like "my code gives the wrong answer" helps little when trying to seek debugging advice from others.
- A good framework for reporting a bug:
  1. Describe the task you intend the code to do.
  2. Describe the test case you used and other relevant constraints about the input.
  3. Describe the discrepancy between expected output and actual output.

A Sequal to Our Debugging Techniques

- **Core skill as a programmer: Ask the right questions.**
- Saying things like "my code gives the wrong answer" helps little when trying to seek debugging advice from others.
- A good framework for reporting a bug:
  1. Describe the task you intend the code to do.
  2. Describe the test case you used and other relevant constraints about the input.
  3. Describe the discrepancy between expected output and actual output.
  4. List down the things you have tried to resolve the issue, if any.

A Sequal to Our Debugging Techniques

- **Core skill as a programmer: Ask the right questions**.
- Saying things like "my code gives the wrong answer" helps little when trying to seek debugging advice from others.
- A good framework for reporting a bug:
  1. Describe the task you intend the code to do.
  2. Describe the test case you used and other relevant constraints about the input.
  3. Describe the discrepancy between expected output and actual output.
  4. List down the things you have tried to resolve the issue, if any.
  5. List down the suspected causes for the issue, if any.

The `assert` Library

- `assert.h` is a library designed to help with debugging procedures.

The `assert` Library

- `assert.h` is a library designed to help with debugging procedures.
- Usage: `assert(p)` where `p` is a boolean expression.

The `assert` Library

- `assert.h` is a library designed to help with debugging procedures.
- Usage: `assert(p)` where `p` is a boolean expression.
- When the condition `p` fails during program execution, the program will halt with an error message.
- Let's do a short demonstration with `odd.c`.

Exercise 1: `odd.c`

Let m be the smallest odd number strictly greater than n, what can you say about the numerical relationship between m and n?

Exercise 1: `odd.c`

Let m be the smallest odd number strictly greater than n, what can you say about the numerical relationship between m and n?

$m \geq n + 1$.

Exercise 1: `odd.c`

Let m be the smallest odd number strictly greater than n, what can you say about the numerical relationship between m and n? $m \geq n + 1$.

So why not we let $m = n + 1 + k$ for some non-negative integer k? What can you say about k?

$$k = \begin{cases} 1, & \text{if } n \text{ is odd} \\ 0, & \text{if } n \text{ is even} \end{cases} .$$

Exercise 1: `odd.c`

Let m be the smallest odd number strictly greater than n, what can you say about the numerical relationship between m and n?
$m \geq n + 1$.

So why not we let $m = n + 1 + k$ for some non-negative integer k? What can you say about k?

$$k = \begin{cases} 1, & \text{if } n \text{ is odd} \\ 0, & \text{if } n \text{ is even} \end{cases} .$$

In other words, k is determined completely by the parity of n.

Exercise 1: `odd.c`

Let m be the smallest odd number strictly greater than n, what can you say about the numerical relationship between m and n? $m \geq n + 1$.

So why not we let $m = n + 1 + k$ for some non-negative integer k? What can you say about k?

$$k = \begin{cases} 1, & \text{if } n \text{ is odd} \\ 0, & \text{if } n \text{ is even} \end{cases} .$$

In other words, k is determined completely by the parity of n. So why not we write k as a function of n? What is this function?

$$k(n) = (n \,\%\, 2)^2 .$$

Exercise 1: `odd.c`

Let m be the smallest odd number strictly greater than n, what can you say about the numerical relationship between m and n? $m \geq n + 1$.

So why not we let $m = n + 1 + k$ for some non-negative integer k? What can you say about k?

$$k = \begin{cases} 1, & \text{if } n \text{ is odd} \\ 0, & \text{if } n \text{ is even} \end{cases} .$$

In other words, k is determined completely by the parity of n. So why not we write k as a function of n? What is this function?

$$k(n) = (n \,\%\, 2)^2 .$$

So it suffices to return `n + (n % 2) * (n % 2) + 1`.

Exercise 1: `sum.c`

Some of you used unnecessarily convoluted `if-else` checks.

Exercise 1: `sum.c`

Some of you used unnecessarily convoluted `if`-`else` checks.

Note that to compute `x + y` in this question is equivalent to the following:

If the number is positive, add it to the sum, otherwise do nothing.

Exercise 1: `sum.c`

Some of you used unnecessarily convoluted `if-else` checks. Note that to compute `x + y` in this question is equivalent to the following:

If the number is positive, add it to the sum, otherwise do nothing.

```
long compute_sum_if_positive(long x, long y){
    long sum = 0;
    if (x > 0){
        sum += x;
    }
    if (y > 0){
        sum += y;
    }
    return sum;
}
```

Exercise 1: multiple.c

```
bool is_multiple(long x, long y) {
    return x % y == 0 || y % x == 0;
}
```

Is this code correct?

Exercise 1: multiple.c

```
bool is_multiple(long x, long y) {
    return x % y == 0 || y % x == 0;
}
```

Is this code correct? **No, because modulo 0 is undefined**.

Exercise 1: multiple.c


```c
bool is_multiple(long x, long y) {
    return x % y == 0 || y % x == 0;
}
```

Is this code correct? **No, because modulo 0 is undefined**.
So you need to check for `y == 0` separately.

```c
bool is_multiple(long x, long y) {
    return (y == 0) || (x % y == 0) || (y % x == 0);
}
```

Exercise 1: date.c


```
bool is_dates_increasing(long m1, long m2, long d1, long d2) {
    return m1 < m2 || (m1 == m2 && d1 < d2);
}
```

In main, conduct the following check:

```
if (is_dates_increasing(m1, m2, d1, d2)
    && is_dates_increasing(m2, m3, d2, d3)) {
    cs1010_println_string("yes");
} else {
    cs1010_println_string("no");
}
```

Exercise 1: date.c


```c
bool is_dates_increasing(long m1, long m2, long d1, long d2) {
    return m1 < m2 || (m1 == m2 && d1 < d2);
}
```

In `main`, conduct the following check:

```c
if (is_dates_increasing(m1, m2, d1, d2)
    && is_dates_increasing(m2, m3, d2, d3)) {
    cs1010_println_string("yes");
} else {
    cs1010_println_string("no");
}
```

Question: Why is it not good to write a function to check for 3 dates directly?

Exercise 1: date.c


```
bool is_dates_increasing(long m1, long m2, long d1, long d2) {
    return m1 < m2 || (m1 == m2 && d1 < d2);
}
```

In main, conduct the following check:

```
if (is_dates_increasing(m1, m2, d1, d2)
    && is_dates_increasing(m2, m3, d2, d3)) {
    cs1010_println_string("yes");
} else {
    cs1010_println_string("no");
}
```

Question: Why is it not good to write a function to check for 3 dates directly?
Answer: Less extensible. If we can check for a pair of dates, we can surely check for any collection of *n* dates, but the converse is not true.

Exercise 1: date.c

```
bool is_dates_increasing(long m1, long m2, long d1, long d2) {
    return m1 < m2 || (m1 == m2 && d1 < d2);
}
```

In `main`, conduct the following check:

```
if (is_dates_increasing(m1, m2, d1, d2)
    && is_dates_increasing(m2, m3, d2, d3)) {
    cs1010_println_string("yes");
} else {
    cs1010_println_string("no");
}
```

Question: Why is it not good to write a function to check for 3 dates directly?
Answer: Less extensible. If we can check for a pair of dates, we can surely check for any collection of *n* dates, but the converse is not true.

Bonus Question: Can we do this question without the `is_dates_increasing` function?

Exercise 1: date.c


```
bool is_dates_increasing(long m1, long m2, long d1, long d2) {
    return m1 < m2 || (m1 == m2 && d1 < d2);
}
```

In `main`, conduct the following check:

```
if (is_dates_increasing(m1, m2, d1, d2)
    && is_dates_increasing(m2, m3, d2, d3)) {
    cs1010_println_string("yes");
} else {
    cs1010_println_string("no");
}
```

Question: Why is it not good to write a function to check for 3 dates directly?
Answer: Less extensible. If we can check for a pair of dates, we can surely check for any collection of n dates, but the converse is not true.

Bonus Question: Can we do this question without the `is_dates_increasing` function? **Yes.** Just map each (m, d) date to the integer $100m + d$.

Exercise 1 Review: `power.c`

Note that this is essentially

$$x^y = x^{y-1} \cdot x$$

where x and y are integers and $y \geq 0$.

Exercise 1 Review: `power.c`

Note that this is essentially

$$x^y = x^{y-1} \cdot x$$

where x and y are integers and $y \geq 0$. A baseline solution:

```c
long compute_power(long x, long y) {
    if (y == 0) {
        return 1;
    }
    return compute_power(x, y - 1) * x;
}
```

Exercise 1 Review: `power.c`

Note that this is essentially

$$x^y = x^{y-1} \cdot x$$

where x and y are integers and $y \geq 0$. A baseline solution:

```c
long compute_power(long x, long y) {
    if (y == 0) {
        return 1;
    }
    return compute_power(x, y - 1) * x;
}
```

This has a high chance of not being able to pass all test cases. Why?

Exercise 1 Review: power.c

Note that this is essentially

$$x^y = x^{y-1} \cdot x$$

where x and y are integers and $y \geq 0$. A baseline solution:

```
long compute_power(long x, long y) {
    if (y == 0) {
        return 1;
    }
    return compute_power(x, y - 1) * x;
}
```

This has a high chance of not being able to pass all test cases. Why? Consider $x = 0$ and $y = 10^8$.

```
long compute_power(long x, long y) {
    if (y == 0) {
        return 1;
    }
    if (x == 0) {
        return 0;
    }
    return compute_power(x, y - 1) * x;
}
```

Exercise 1 Review: `power.c`

Note that this is essentially

$$x^y = x^{y-1} \cdot x$$

where x and y are integers and $y \geq 0$. A baseline solution:

```
long compute_power(long x, long y) {
    if (y == 0) {
        return 1;
    }
    return compute_power(x, y - 1) * x;
}
```

This has a high chance of not being able to pass all test cases. Why? Consider $x = 0$ and $y = 10^8$.

```
long compute_power(long x, long y) {
    if (y == 0) {
        return 1;
    }
    if (x == 0) {
        return 0;
    }
    return compute_power(x, y - 1) * x;
}
```

But can we do better?

Exercise 1 Review: power.c

```
long compute_power(long x, long y) {
    if (y == 0) {
        return 1;
    }
    if (x == 0) {
        return 0;
    }
    return compute_power(x, y - 1) * x;
}
```

Consider $x = 1$ and $y = 10^8$. There's no need to compute this either.

```
long compute_power(long x, long y) {
    if (y == 0) {
        return 1;
    }
    if (x == 0 || x == 1) {
        return x;
    }
    return compute_power(x, y - 1) * x;
}
```

Exercise 1 Review: power.c


```
long compute_power(long x, long y) {
    if (y == 0) {
        return 1;
    }
    if (x == 0) {
        return 0;
    }
    return compute_power(x, y - 1) * x;
}
```

Consider $x = 1$ and $y = 10^8$. There's no need to compute this either.

```
long compute_power(long x, long y) {
    if (y == 0) {
        return 1;
    }
    if (x == 0 || x == 1) {
        return x;
    }
    return compute_power(x, y - 1) * x;
}
```

But can we do better?

Exercise 1 Review: power.c

```
long compute_power(long x, long y) {
    if (y == 0) {
        return 1;
    }
    if (x == 0 || x == 1) {
        return x;
    }
    return compute_power(x, y - 1) * x;
}
```

Consider $x = -1$. There are only 2 possible answers.

```
long compute_power(long x, long y) {
    if (y == 0) {
        return 1;
    }
    if (x == 0 || x == 1) {
        return x;
    }
    if (x == -1) {
        return y % 2 == 0 ? 1 : -1;
    }
    return compute_power(x, y - 1) * x;
}
```

Exercise 1 Review: power.c


```c
long compute_power(long x, long y) {
    if (y == 0) {
        return 1;
    }
    if (x == 0 || x == 1) {
        return x;
    }
    return compute_power(x, y - 1) * x;
}
```

Consider $x = -1$. There are only 2 possible answers.

```c
long compute_power(long x, long y) {
    if (y == 0) {
        return 1;
    }
    if (x == 0 || x == 1) {
        return x;
    }
    if (x == -1) {
        return y % 2 == 0 ? 1 : -1;
    }
    return compute_power(x, y - 1) * x;
}
```

But can we do better?

Exponentiation by Squaring

Recall that

$$x^y = \begin{cases} \left(x^2\right)^{\frac{y}{2}} & \text{if } y \text{ is even} \\ \left(x^2\right)^{\frac{y-1}{2}} x & \text{if } y \text{ is odd} \end{cases}.$$

Exponentiation by Squaring

Recall that

$$x^y = \begin{cases} \left(x^2\right)^{\frac{y}{2}} & \text{if } y \text{ is even} \\ \left(x^2\right)^{\frac{y-1}{2}} x & \text{if } y \text{ is odd} \end{cases}.$$

```
long compute_power(long x, long y) {
    if (y == 0) {
        return 1;
    }
    if (x == 0 || x == 1) {
        return x;
    }
    if (x == -1) {
        return y % 2 == 0 ? 1 : -1;
    }
    if (y % 2 == 1) {
        return compute_power(x * x, (y - 1) / 2) * x;
    }
    return compute_power(x * x, y / 2);
}
```

What's the advantage of using this implementation?

Exponentiation by Squaring

Recall that

$$x^y = \begin{cases} \left(x^2\right)^{\frac{y}{2}} & \text{if } y \text{ is even} \\ \left(x^2\right)^{\frac{y-1}{2}} x & \text{if } y \text{ is odd} \end{cases}.$$

```
long compute_power(long x, long y) {
    if (y == 0) {
        return 1;
    }
    if (x == 0 || x == 1) {
        return x;
    }
    if (x == -1) {
        return y % 2 == 0 ? 1 : -1;
    }
    if (y % 2 == 1) {
        return compute_power(x * x, (y - 1) / 2) * x;
    }
    return compute_power(x * x, y / 2);
}
```

What's the advantage of using this implementation? This will **reduce the depth of your recursion** from y to approximately $\log_2 y$ (significant improvement).

Exercise 1 Review: `taxi.c`

General idea:

- Each taxi ride consists of two parts: **base fare** and **surcharge**.

Exercise 1 Review: `taxi.c`

General idea:

- Each taxi ride consists of two parts: **base fare** and **surcharge**.
- There are three cases for the base fare: $d > 10000$, $1000 < d \leq 10000$ and $d \leq 1000$.

Exercise 1 Review: `taxi.c`

General idea:

- Each taxi ride consists of two parts: **base fare** and **surcharge**.
- There are three cases for the base fare: $d > 10000$, $1000 < d \leq 10000$ and $d \leq 1000$.
- For the previous two cases, you need a **ceiling function** to compute the number of additional charges applicable to the ride.

Exercise 1 Review: `taxi.c`

General idea:

- Each taxi ride consists of two parts: **base fare** and **surcharge**.
- There are three cases for the base fare: $d > 10000$, $1000 < d \leq 10000$ and $d \leq 1000$.
- For the previous two cases, you need a **ceiling function** to compute the number of additional charges applicable to the ride.
- The surcharge is a multiplicative coefficient determined by the **starting time** of the ride. You need to check both **the day of the week** and **the time of the day**.

Extra: a Smart Way to Compute $\left\lceil \frac{n}{m} \right\rceil$

For any positive integers n and m, the standard way to compute $\left\lceil \frac{n}{m} \right\rceil$ is:

```
long ceil_of_quotient(long n, long m) {
    if (n % m == 0) {
        return n / m;
    }
    return n / m + 1;
}
```

Extra: a Smart Way to Compute $\left\lceil \frac{n}{m} \right\rceil$

For any positive integers n and m, the standard way to compute $\left\lceil \frac{n}{m} \right\rceil$ is:

```
long ceil_of_quotient(long n, long m) {
    if (n % m == 0) {
        return n / m;
    }
    return n / m + 1;
}
```

However, let us write $n = mq + r$ for some integers $q \geq 0$ and $0 \leq r \leq m - 1$.

Extra: a Smart Way to Compute $\left\lceil \frac{n}{m} \right\rceil$

For any positive integers n and m, the standard way to compute $\left\lceil \frac{n}{m} \right\rceil$ is:

```
long ceil_of_quotient(long n, long m) {
    if (n % m == 0) {
        return n / m;
    }
    return n / m + 1;
}
```

However, let us write $n = mq + r$ for some integers $q \geq 0$ and $0 \leq r \leq m - 1$. Now let's consider

$$\frac{n + m - 1}{m} = \frac{mq + r + m - 1}{m} = \frac{m(q + 1) + r - 1}{m}.$$

What can you notice?

Extra: a Smart Way to Compute $\left\lceil \frac{n}{m} \right\rceil$

For any positive integers n and m, the standard way to compute $\left\lceil \frac{n}{m} \right\rceil$ is:

```
long ceil_of_quotient(long n, long m) {
    if (n % m == 0) {
        return n / m;
    }
    return n / m + 1;
}
```

However, let us write $n = mq + r$ for some integers $q \geq 0$ and $0 \leq r \leq m - 1$. Now let's consider

$$\frac{n + m - 1}{m} = \frac{mq + r + m - 1}{m} = \frac{m(q + 1) + r - 1}{m}.$$

What can you notice? If $r = 0$, then the above numerator is the biggest integer strictly less than $m(q + 1)$ and so the quotient evaluates to q.

Extra: a Smart Way to Compute $\left\lceil \frac{n}{m} \right\rceil$

For any positive integers n and m, the standard way to compute $\left\lceil \frac{n}{m} \right\rceil$ is:

```
long ceil_of_quotient(long n, long m) {
    if (n % m == 0) {
        return n / m;
    }
    return n / m + 1;
}
```

However, let us write $n = mq + r$ for some integers $q \geq 0$ and $0 \leq r \leq m - 1$. Now let's consider

$$\frac{n + m - 1}{m} = \frac{mq + r + m - 1}{m} = \frac{m(q + 1) + r - 1}{m}.$$

What can you notice? If $r = 0$, then the above numerator is the biggest integer strictly less than $m(q + 1)$ and so the quotient evaluates to q. If $0 < r \leq m - 1$, then the above numerator is at least $m(q + 1)$ but strictly less than $m(q + 2)$, so the quotient evaluates to $q + 1$.

Extra: a Smart Way to Compute $\left\lceil \frac{n}{m} \right\rceil$

For any positive integers n and m, the standard way to compute $\left\lceil \frac{n}{m} \right\rceil$ is:

```
long ceil_of_quotient(long n, long m) {
    if (n % m == 0) {
        return n / m;
    }
    return n / m + 1;
}
```

However, let us write $n = mq + r$ for some integers $q \geq 0$ and $0 \leq r \leq m - 1$. Now let's consider

$$\frac{n + m - 1}{m} = \frac{mq + r + m - 1}{m} = \frac{m(q + 1) + r - 1}{m}.$$

What can you notice? If $r = 0$, then the above numerator is the biggest integer strictly less than $m(q + 1)$ and so the quotient evaluates to q. If $0 < r \leq m - 1$, then the above numerator is at least $m(q + 1)$ but strictly less than $m(q + 2)$, so the quotient evaluates to $q + 1$. Therefore,

```
long ceil_of_quotient(long n, long m) {
    return (n + m - 1) / m;
}
```

Live: We'll demonstrate the coding style using `digits.c` and `suffix.c`

My thought process for `digits.c`:

Live: We'll demonstrate the coding style using `digits.c` and `suffix.c`

My thought process for `digits.c`:

- I'm given a **non-negative** integer.

Live: We'll demonstrate the coding style using `digits.c` and `suffix.c`

My thought process for `digits.c`:

- I'm given a **non-negative** integer.
- I can always write it as $a_1 a_2 a_3 \cdots a_n$ where each of the a_i's is a non-negative integer less than 10.

Live: We'll demonstrate the coding style using `digits.c` and `suffix.c`

My thought process for `digits.c`:

- I'm given a **non-negative** integer.
- I can always write it as $a_1 a_2 a_3 \cdots a_n$ where each of the a_i's is a non-negative integer less than 10.
- My task is essentially to compute

$$\sum_{i=1}^{n} a_i^3 = a_n^3 + \sum_{i=1}^{n-1} a_i^3 \quad (\textbf{A recursive pattern!}).$$

Live: We'll demonstrate the coding style using `digits.c` and `suffix.c`

My thought process for `digits.c`:

- I'm given a **non-negative** integer.
- I can always write it as $a_1 a_2 a_3 \cdots a_n$ where each of the a_i's is a non-negative integer less than 10.
- My task is essentially to compute

$$\sum_{i=1}^{n} a_i^3 = a_n^3 + \sum_{i=1}^{n-1} a_i^3 \quad (\textbf{A recursive pattern!}).$$

- So I will chop the integer into $a_1 \cdots a_{n-1} | a_n$. And since a_n^3 is trivial to compute, I only need to deal with $a_1 a_2 \cdots a_{n-1}$ now.

Live: We'll demonstrate the coding style using `digits.c` and `suffix.c`

My thought process for `digits.c`:

- I'm given a **non-negative** integer.
- I can always write it as $a_1 a_2 a_3 \cdots a_n$ where each of the a_i's is a non-negative integer less than 10.
- My task is essentially to compute

$$\sum_{i=1}^{n} a_i^3 = a_n^3 + \sum_{i=1}^{n-1} a_i^3 \quad (\textbf{A recursive pattern!}).$$

- So I will chop the integer into $a_1 \cdots a_{n-1} | a_n$. And since a_n^3 is trivial to compute, I only need to deal with $a_1 a_2 \cdots a_{n-1}$ now.
- Suppose I continue chopping in this way, there will come a time where only a_1 remains in the un-chopped portion.

Live: We'll demonstrate the coding style using `digits.c` and `suffix.c`

My thought process for `digits.c`:

- I'm given a **non-negative** integer.
- I can always write it as $a_1a_2a_3\cdots a_n$ where each of the a_i's is a non-negative integer less than 10.
- My task is essentially to compute

$$\sum_{i=1}^{n} a_i^3 = a_n^3 + \sum_{i=1}^{n-1} a_i^3 \quad (\textbf{A recursive pattern!}).$$

- So I will chop the integer into $a_1\cdots a_{n-1}|a_n$. And since a_n^3 is trivial to compute, I only need to deal with $a_1a_2\cdots a_{n-1}$ now.
- Suppose I continue chopping in this way, there will come a time where only a_1 remains in the un-chopped portion.
- This is when my recursion should terminate!

Live: `digits.c` and `suffix.c`

My thought process for `suffix.c`:

Live: `digits.c` and `suffix.c`

My thought process for `suffix.c`:

- There are a lot more cases where the suffix is "-th", so I'll use that as my **default** fallback.

Live: `digits.c` and `suffix.c`

My thought process for `suffix.c`:

- There are a lot more cases where the suffix is ”-th”, so I’ll use that as my **default** fallback.
- An integer whose last digit is 1, 2 or 3 should use the special suffixes accordingly.

Live: `digits.c` and `suffix.c`

My thought process for `suffix.c`:

- There are a lot more cases where the suffix is "-th", so I'll use that as my **default** fallback.
- An integer whose last digit is 1, 2 or 3 should use the special suffixes accordingly.
- Wait, is my previous point correct?

Live: `digits.c` and `suffix.c`

My thought process for `suffix.c`:

- There are a lot more cases where the suffix is "-th", so I'll use that as my **default** fallback.
- An integer whose last digit is 1, 2 or 3 should use the special suffixes accordingly.
- Wait, is my previous point correct?
- **No**, because 11, 12 and 13 are somehow spelt irregularly in English :(

Live: `digits.c` and `suffix.c`

My thought process for `suffix.c`:

- There are a lot more cases where the suffix is "-th", so I'll use that as my **default** fallback.
- An integer whose last digit is 1, 2 or 3 should use the special suffixes accordingly.
- Wait, is my previous point correct?
- **No**, because 11, 12 and 13 are somehow spelt irregularly in English :(
- So I need to make sure that the numbers ending with 1, 2 and 3 are not 11, 12 and 13 (**edge cases!**).

Live: `digits.c` and `suffix.c`

My thought process for `suffix.c`:

- There are a lot more cases where the suffix is "-th", so I'll use that as my **default** fallback.
- An integer whose last digit is 1, 2 or 3 should use the special suffixes accordingly.
- Wait, is my previous point correct?
- **No**, because 11, 12 and 13 are somehow spelt irregularly in English :(
- So I need to make sure that the numbers ending with 1, 2 and 3 are not 11, 12 and 13 (**edge cases!**).
- For anything else, it will just use "-th".

Selected Problems from Exercise 2

`onigiri.c` (おにぎり)

`onigiri.c` (おにぎり)

- **We need to print n lines.**

Selected Problems from Exercise 2

onigiri.c (おにぎり)

- **We need to print *n* lines.**
- The following should be natural:

```c
for (long i = 0; i < n; i += 1) {
    // Print the (i + 1)-th line.
}
```

Selected Problems from Exercise 2

`onigiri.c` (おにぎり)

- **We need to print *n* lines.**
- The following should be natural:

```c
for (long i = 0; i < n; i += 1) {
    // Print the (i + 1)-th line.
}
```

- But within each line, we need to print three sections!

Selected Problems from Exercise 2

`onigiri.c` (おにぎり)

- **We need to print *n* lines.**
- The following should be natural:

```
for (long i = 0; i < n; i += 1) {
    // Print the (i + 1)-th line.
}
```

- But within each line, we need to print three sections!
- Each of the three sections consists a number of repeated characters. (**Another loop?**)

`onigiri.c` (おにぎり)

- **We need to print *n* lines.**
- The following should be natural:

```c
for (long i = 0; i < n; i += 1) {
    // Print the (i + 1)-th line.
}
```

- But within each line, we need to print three sections!
- Each of the three sections consists a number of repeated characters. (**Another loop?**)
- ```c
  for (long i = 0; i < n; i += 1) {
      // Print " " for (h - i - 1) times.
      // Print "#" for (2i + 1) times.
      // Print " " for another (h - i - 1) ti
      // Line break.
  }
  ```
  ```

Selected Problems from Exercise 2

`binary.c`

Selected Problems from Exercise 2

`binary.c`

- How to convert an n-ary m-digit number $x_1 x_2 \cdots x_m$ to base 10:

$$n^0 x_m + n^1 x_{m-1} + n^2 x_{m-2} + \cdots + n^{m-1} x_1 = \sum_{i=0}^{m-1} n^i x_{m-i}.$$

Selected Problems from Exercise 2

`binary.c`

- How to convert an n-ary m-digit number $x_1 x_2 \cdots x_m$ to base 10:

$$n^0 x_m + n^1 x_{m-1} + n^2 x_{m-2} + \cdots + n^{m-1} x_1 = \sum_{i=0}^{m-1} n^i x_{m-i}.$$

- In the binary case, the binary number $b_1 b_2 \cdots b_n$ is converted to:

$$2^0 b_n + 2^1 b_{n-1} + \cdots + 2^{n-1} b_1 = \sum_{i=0}^{n-1} 2^i b_{n-i}.$$

Selected Problems from Exercise 2

binary.c

- How to convert an n-ary m-digit number $x_1 x_2 \cdots x_m$ to base 10:

$$n^0 x_m + n^1 x_{m-1} + n^2 x_{m-2} + \cdots + n^{m-1} x_1 = \sum_{i=0}^{m-1} n^i x_{m-i}.$$

- In the binary case, the binary number $b_1 b_2 \cdots b_n$ is converted to:

$$2^0 b_n + 2^1 b_{n-1} + \cdots + 2^{n-1} b_1 = \sum_{i=0}^{n-1} 2^i b_{n-i}.$$

- Recall from `digits.c` that we can chop the number to $b_1 \cdots b_{n-1} | b_n$.

Selected Problems from Exercise 2

binary.c

- Tempted to do `to_base_10(b / 10) + b % 10` like `digits.c`, but notice that:

$$b_1 \cdots b_{n-1} = \sum_{i=0}^{n-2} 2^i b_{n-1-i}$$

$$\begin{aligned} b_1 \cdots b_n &= \sum_{j=0}^{n-1} 2^j b_{n-j} \\ &= b_n + \sum_{j=1}^{n-1} 2^j b_{n-j} \\ &= b_n + 2 \sum_{i=0}^{n-2} 2^i b_{n-i-1} \end{aligned}$$

Selected Problems from Exercise 2

binary.c

```c
long to_base_10(long b) {
    if (...) {
        // Under some condition, the binary and
        // base-10 representations are the same.
    }
    // b1b2...bn == bn + 2 * b1b2...b{n-1}
}
```

Selected Problems from Exercise 2

binary.c

- But for such summation problems, a loop is always your saviour when you cannot understand the mystery behind recursion!

binary.c

- But for such summation problems, a loop is always your saviour when you cannot understand the mystery behind recursion!
- $2^0 b_n + 2^1 b_{n-1} + \cdots + 2^{n-1} b_1 = \sum_{i=0}^{n-1} 2^i b_{n-i}$. (Notice that the i here is actually your loop index?)

Selected Problems from Exercise 2

binary.c

- But for such summation problems, a loop is always your saviour when you cannot understand the mystery behind recursion!
- $2^0 b_n + 2^1 b_{n-1} + \cdots + 2^{n-1} b_1 = \sum_{i=0}^{n-1} 2^i b_{n-i}$. (Notice that the i here is actually your loop index?)
- What the summation implies: at the i-th iteration, chop off the last digit of the binary number B, multiply it by 2^i and add it to your sum. Repeat this until you have no more digits, i.e., $B = 0$.

Selected Problems from Exercise 2

binary.c

- But for such summation problems, a loop is always your saviour when you cannot understand the mystery behind recursion!
- $2^0 b_n + 2^1 b_{n-1} + \cdots + 2^{n-1} b_1 = \sum_{i=0}^{n-1} 2^i b_{n-i}$. (Notice that the i here is actually your loop index?)
- What the summation implies: at the i-th iteration, chop off the last digit of the binary number B, multiply it by 2^i and add it to your sum. Repeat this until you have no more digits, i.e., $B = 0$.
- The final sum will be the number in base 10.

Selected Problems from Exercise 2

binary.c

```c
long sum = 0;
while (...) {
    long ith_digit = b % 10;
    // Continue summing the digits up
    // utill there's no digit left.
    long ith_digit_in_base_10 = ...;
    // The i-th digit b_i will be converted
    // to 2^i * b_i. (Do we re-compute 2^i every time?)
    sum += ith_digit_in_base_10;
    b /= 10; // Now discard the last digit.
}
return sum;
```

Selected Problems from Exercise 2

`fibonacci.c` (A never-dying classic)

- The *Fibonacci Sequence* is defined by the following recurrence relation:

$$f_1 = f_2 = 1,$$
$$f_n = f_{n-1} + f_{n-2}, \quad \text{for } n \geq 3.$$

Selected Problems from Exercise 2

`fibonacci.c` (A never-dying classic)

- The *Fibonacci Sequence* is defined by the following recurrence relation:

$$
\begin{aligned}
f_1 &= f_2 = 1, \\
f_n &= f_{n-1} + f_{n-2}, \quad \text{for } n \geq 3.
\end{aligned}
$$

- This is already an established recursive algorithm!

Selected Problems from Exercise 2

`fibonacci.c` (A never-dying classic)

- The *Fibonacci Sequence* is defined by the following recurrence relation:

$$
\begin{aligned}
f_1 &= f_2 = 1, \\
f_n &= f_{n-1} + f_{n-2}, \quad \text{for } n \geq 3.
\end{aligned}
$$

- This is already an established recursive algorithm!
- Any issues with using just this recurrence relation to implement our program?

Selected Problems from Exercise 2

`fibonacci.c` (A never-dying classic)

- The *Fibonacci Sequence* is defined by the following recurrence relation:

$$
\begin{aligned}
f_1 &= f_2 = 1, \\
f_n &= f_{n-1} + f_{n-2}, \quad \text{for } n \geq 3.
\end{aligned}
$$

- This is already an established recursive algorithm!
- Any issues with using just this recurrence relation to implement our program?
- **There are duplicated computations!** (But we can tackle those when you learn about arrays).

Selected Problems from Exercise 2

`fibonacci.c` (A never-dying classic)

- An iterative approach is more efficient.

Selected Problems from Exercise 2

`fibonacci.c` (A never-dying classic)

- An iterative approach is more efficient.
- Consider the following procedures:

$$\underset{(prev)}{f_{n-1}} + \underset{(curr)}{f_n} = \underset{(next)}{f_{n+1}}$$
$$\underset{(prev)}{f_n} + \underset{(curr)}{f_{n+1}} = \underset{(next)}{f_{n+2}}.$$

Any inspirations?

Selected Problems from Exercise 2

`fibonacci.c` (A never-dying classic)

- An iterative approach is more efficient.
- Consider the following procedures:

$$\underset{(prev)}{f_{n-1}} + \underset{(curr)}{f_n} = \underset{(next)}{f_{n+1}}$$
$$\underset{(prev)}{f_n} + \underset{(curr)}{f_{n+1}} = \underset{(next)}{f_{n+2}}\,.$$

  Any inspirations?

- It turns out to compute the **next** Fibonacci number, we only need to add the **previous** one to the **current** one, so we only need to keep track and update the values inside these three "containers", until we reach the target index.

Selected Problems from Exercise 2

`fibonacci.c` (A never-dying classic)

- An iterative approach is more efficient.
- Consider the following procedures:

$$\underset{(prev)}{f_{n-1}} + \underset{(curr)}{f_n} = \underset{(next)}{f_{n+1}}$$

$$\underset{(prev)}{f_n} + \underset{(curr)}{f_{n+1}} = \underset{(next)}{f_{n+2}}.$$

  Any inspirations?
- It turns out to compute the **next** Fibonacci number, we only need to add the **previous** one to the **current** one, so we only need to keep track and update the values inside these three "containers", until we reach the target index.
- This technique is known as **sliding window**.

Selected Problems from Exercise 2

binary.c

```
long prev = ...;
long curr = ...;
// How to properly initialise prev and curr?
long next;
for (i = 3; i <= n; i += 1) {
    // We will keep generating the next
    // Fibonacci number,
    // until the nth one is computed.
    long next = curr + prev;
    // Then, we update prev and curr.
    prev = ...;
    curr = ...;
}
return next;
```

Fibonacci Sequence: for Your Info

The general formula for f_n is

$$f_n = \frac{\sqrt{5}}{5}\left[\left(\frac{1+\sqrt{5}}{2}\right)^n - \left(\frac{1-\sqrt{5}}{2}\right)^n\right],$$

where

$$\frac{1+\sqrt{5}}{2} \approx 1.618$$

is known as the **Golden Ratio**. And surprisingly (or unsurprisingly),

$$\lim_{n\to\infty} \frac{f_{n+1}}{f_n} = \frac{1+\sqrt{5}}{2}.$$