CS1010 Tutorial 6
Group BC1A

14 October 2020

Topics for today

Objectives

- Recap on Topics (C Preprocessing, The assert macro, Efficiency)
- Going through problem set 20, 21, 22
- Feedback for Assignment 4
- Going through Assignment 5: Social
- Going through Midterms
- Summary

C Preprocessing

- #include <filename> → system
  - Copies the files from system directory
  - Used for system headers
- #include "filename" → local
  - Copies the files from local directory
  - Used for own headers → #include "cslocco.h"
- #define CONSTANT   #define x 4
  - Not the same as global variable
  - CONSTANT is read-only
- #define MACRO   # define square (x)*(x)
  - MACRO is a function expression

*Take note that #define copies the variable and replaces it within code
- Always add parenthesis to the macro expression

Problem 20.1 Question

a) Consider the macro below:

```
#define MIN(a,b) (a)>(b)?(a): b)

long i = MIN(10, 20)
long j = MIN(10, 20) + 1;
```

10 ✓

10 11?

long i = 10 < 20 ? 10 : 20

long j = 10 < 20 ? 10 : 20 + 1)

(10 < 20 ? 10 : 20) + 1

What are the values for `i` and `j` after executing the above?

b)

CS1010-trained students should know better than to use the `++` operator, which combines two steps into one and has the side effects on value `i`. Let's say someone who is not trained this way wrote the following code:

```
#define MIN(a,b)  a < b ? a : b

long i = 10
long j = 20
long k = MIN(j, i++)
```

k = 11 i = 12.

long k = j < i++ ? j : i++

i = 11

What are the values of `i` and `k` after executing the above? Explain what happen.

Problem 20.1(a) Answer

The code gets expanded into

```
long i = 10 < 20 ? 10 : 20;
long j = 10 < 20 ? 10 : 20 + 1;
```

So `i` will become 10, `j` will also be 10.

A common mistake is to think that j will be 11.

To fix this, add parenthesis around `a` and `b`

```
#define MIN(a, b) (a) < (b) ? (a) : (b)
```

Problem 20.1(b) Answer

The code gets expanded into

```
long i = 10;
long j = 20;
long k = j < i++ ? j : i++;
```

`k` becomes 11, but `i` becomes 12!

Note that `i++` has side effects, and with a macro, the expression `i++` gets substituted and evaluated twice. So the value of `i` increases.

This **CANNOT** be fixed with putting parenthesis around a and b in the macro. The only way to avoid the problem is to be very careful when we mix macro with statements with side effects or not use `i++` all together.

Problem 20.2 Question

Suppose we write our `SWAP` macro without the opening and closing brackets:

```
#define SWAP(T, x, y) T temp = x;\
  x = y;\
  y = temp;
```

CS1010-trained students should know better than to write `if`-`else` block without `{` and `}`. Suppose someone writes an `if-else` block without `{` and `}`. What could go wrong if they use the macro above, also without the opening and closing brackets?

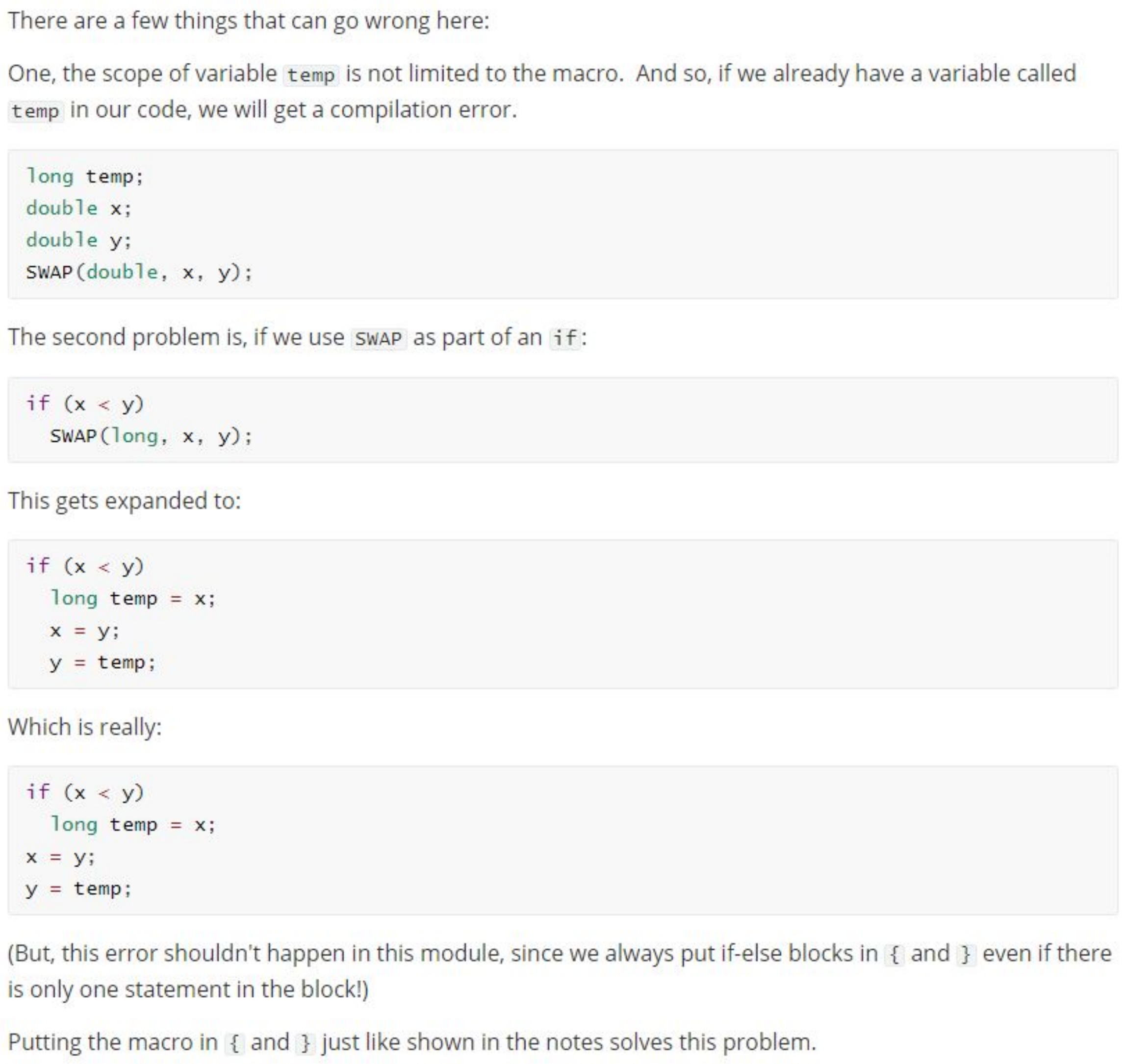
```
② if (sth) {
     swap(T, x, y);
  }

  if (sth) {
     T temp = x;
  x = y;
  y = temp;
  }
```

```
①  int main() {
       long temp = 4
       long x = 1
       long y = 2
     swap (temp, x, y);

    }
```

```
int main() {
   long temp = 4
   long x = 1
   long y = 2
   long temp = x;
   x = y
   y = temp!

}
```

compilation error!

Problem 20.2 Answer



There are a few things that can go wrong here:

One, the scope of variable `temp` is not limited to the macro. And so, if we already have a variable called `temp` in our code, we will get a compilation error.

```
long temp;
double x;
double y;
SWAP(double, x, y);
```

The second problem is, if we use `SWAP` as part of an `if`:

```
if (x < y)
  SWAP(long, x, y);
```

This gets expanded to:

```
if (x < y)
  long temp = x;
  x = y;
  y = temp;
```

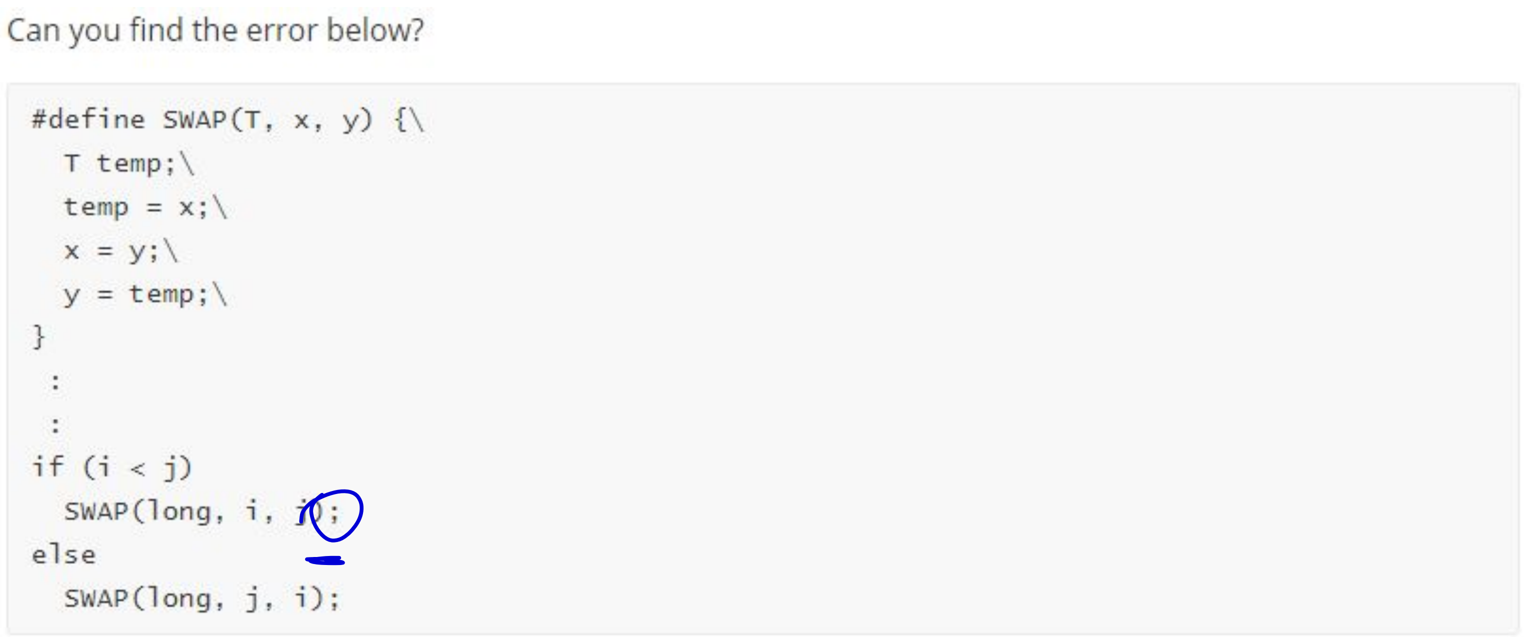
Which is really:

```
if (x < y)
  long temp = x;
x = y;
y = temp;
```

(But, this error shouldn't happen in this module, since we always put if-else blocks in `{` and `}` even if there is only one statement in the block!)

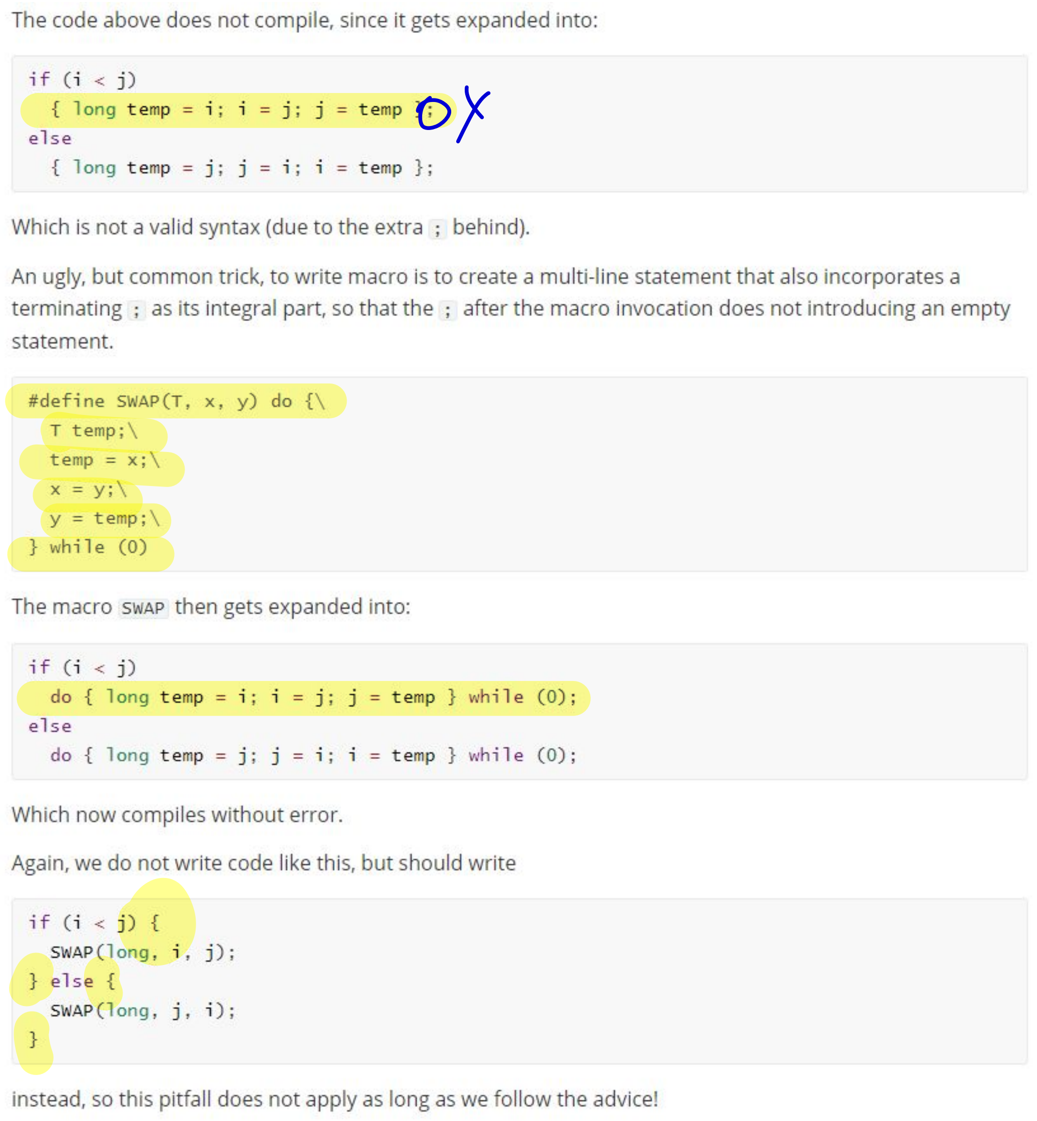
Putting the macro in `{` and `}` just like shown in the notes solves this problem.

Problem 20.2 Question (Extended)


Can you find the error below?

```
#define SWAP(T, x, y) {\
  T temp;\
  temp = x;\
  x = y;\
  y = temp;\
}
 :
 :
if (i < j)
  SWAP(long, i, j);
else
  SWAP(long, j, i);
```

Problem 20.2 Answer (Extended)



The code above does not compile, since it gets expanded into:

```
if (i < j)
  { long temp = i; i = j; j = temp }; X
else
  { long temp = j; j = i; i = temp };
```

Which is not a valid syntax (due to the extra `;` behind).

An ugly, but common trick, to write macro is to create a multi-line statement that also incorporates a terminating `;` as its integral part, so that the `;` after the macro invocation does not introducing an empty statement.

```
#define SWAP(T, x, y) do {\
  T temp;\
  temp = x;\
  x = y;\
  y = temp;\
} while (0)
```

The macro `SWAP` then gets expanded into:

```
if (i < j)
  do { long temp = i; i = j; j = temp } while (0);
else
  do { long temp = j; j = i; i = temp } while (0);
```

Which now compiles without error.

Again, we do not write code like this, but should write

```
if (i < j) {
  SWAP(long, i, j);
} else {
  SWAP(long, j, i);
}
```

instead, so this pitfall does not apply as long as we follow the advice!

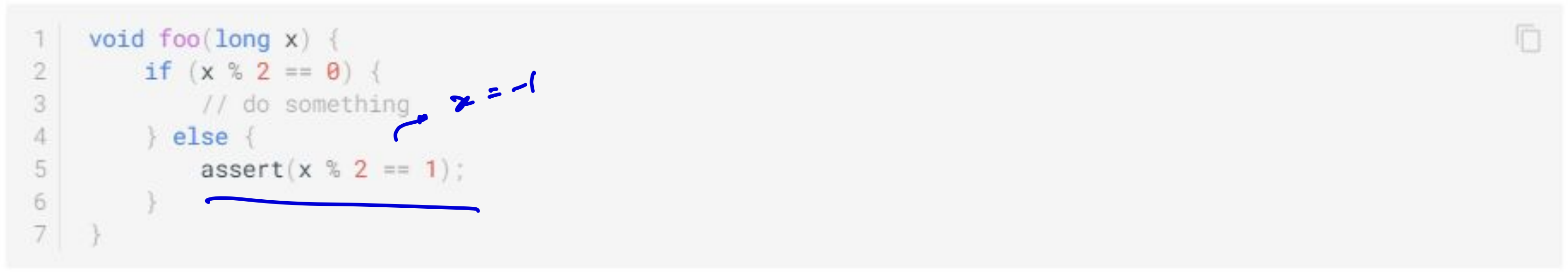
Assert

- Use assert to help you debug your code
- Assert will print the assertion and line that caused program termination

```
int main() {



10    assert ( )        logic expression

      assert
}
```

Problem 21.1 Question & Answer

Consider the code:



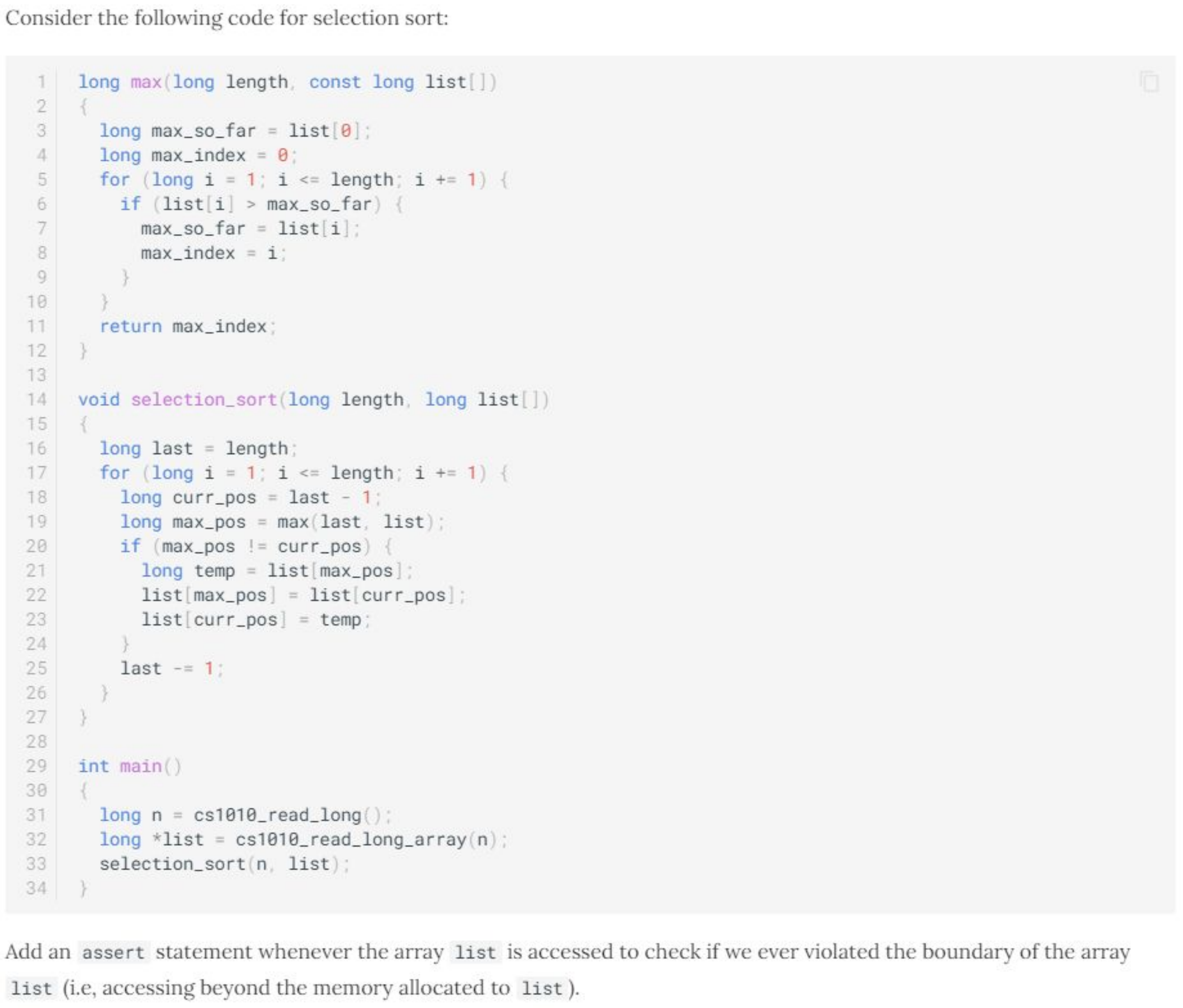
```
1  void foo(long x) {
2      if (x % 2 == 0) {
3          // do something
4      } else {
5          assert(x % 2 == 1);
6      }
7  }
```

$x = -1$

Would the assert in Line 5 above ever fail?

Ans: The assert will fail if x is negative

Problem 21.2 Question & Answer



Consider the following code for selection sort:

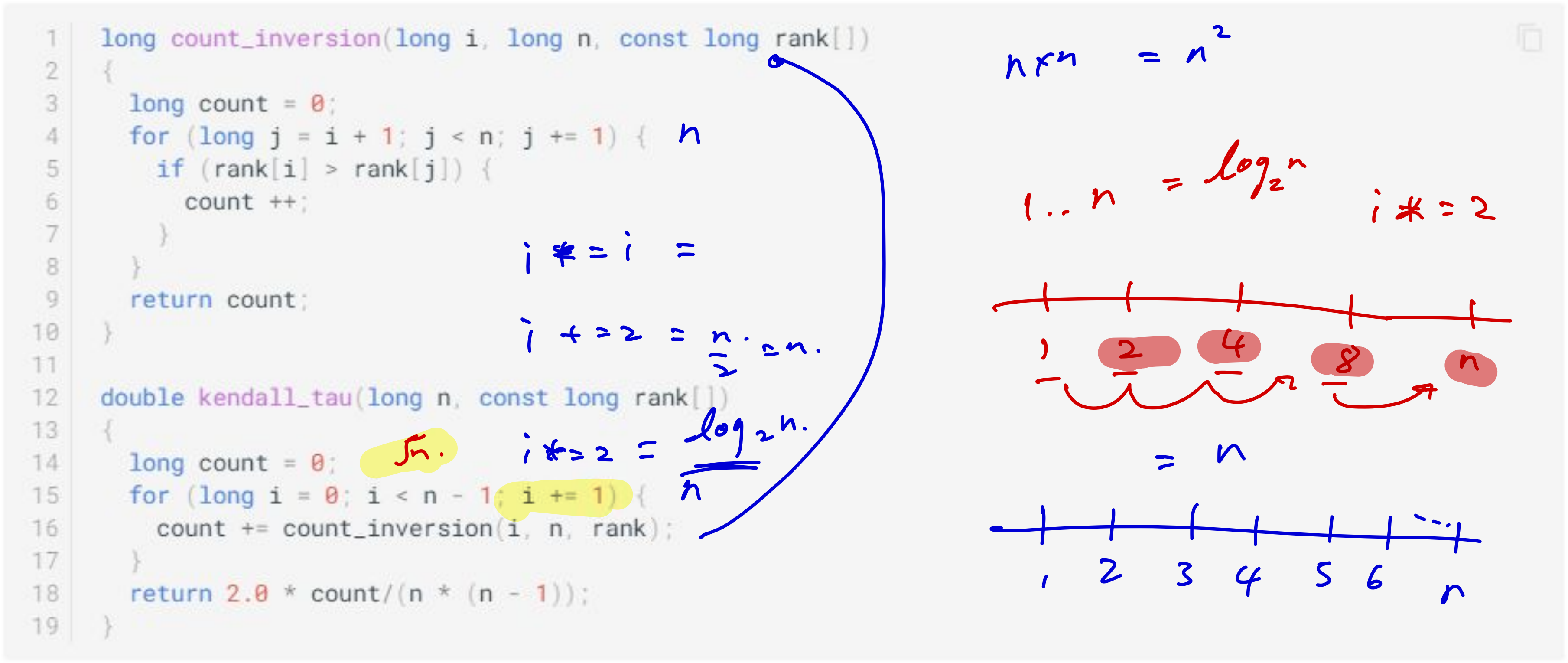
```c
long max(long length, const long list[])
{
  long max_so_far = list[0];
  long max_index = 0;
  for (long i = 1; i <= length; i += 1) {
    if (list[i] > max_so_far) {
      max_so_far = list[i];
      max_index = i;
    }
  }
  return max_index;
}

void selection_sort(long length, long list[])
{
  long last = length;
  for (long i = 1; i <= length; i += 1) {
    long curr_pos = last - 1;
    long max_pos = max(last, list);
    if (max_pos != curr_pos) {
      long temp = list[max_pos];
      list[max_pos] = list[curr_pos];
      list[curr_pos] = temp;
    }
    last -= 1;
  }
}

int main()
{
  long n = cs1010_read_long();
  long *list = cs1010_read_long_array(n);
  selection_sort(n, list);
}
```

Add an `assert` statement whenever the array `list` is accessed to check if we ever violated the boundary of the array `list` (i.e, accessing beyond the memory allocated to `list`).

Ans: add assert just before every line the array is accessed

Big O Notation (Efficiency)

- Big O notation looks at the worse case efficiency
    - We ignore coefficient and only look at how a function grows
    - Sometimes Big O Notation is also known as the "rate of growth" of a function



```
long count_inversion(long i, long n, const long rank[])
{
  long count = 0;
  for (long j = i + 1; j < n; j += 1) {
    if (rank[i] > rank[j]) {
      count ++;
    }
  }
  return count;
}

double kendall_tau(long n, const long rank[])
{
  long count = 0;
  for (long i = 0; i < n - 1; i += 1) {
    count += count_inversion(i, n, rank);
  }
  return 2.0 * count/(n * (n - 1));
}
```

n

$\sqrt{n}.$

$i *= i =$

$i += 2 = \frac{n}{2}. = n.$

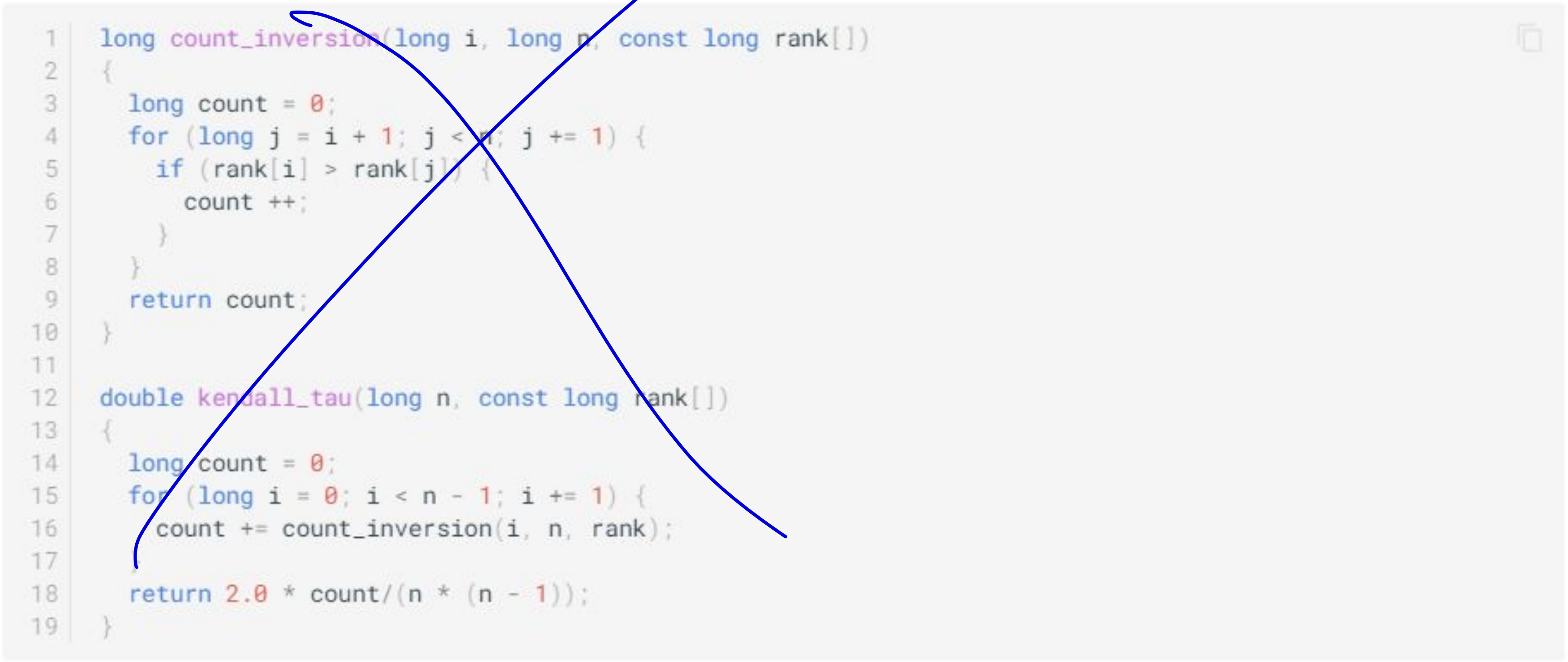
$i *= 2 = \frac{\log_2 n.}{n}$

$n \times n = n^2$

$1 .. n = \log_2 n$

$i *= 2$

Big O Notation (Efficiency)

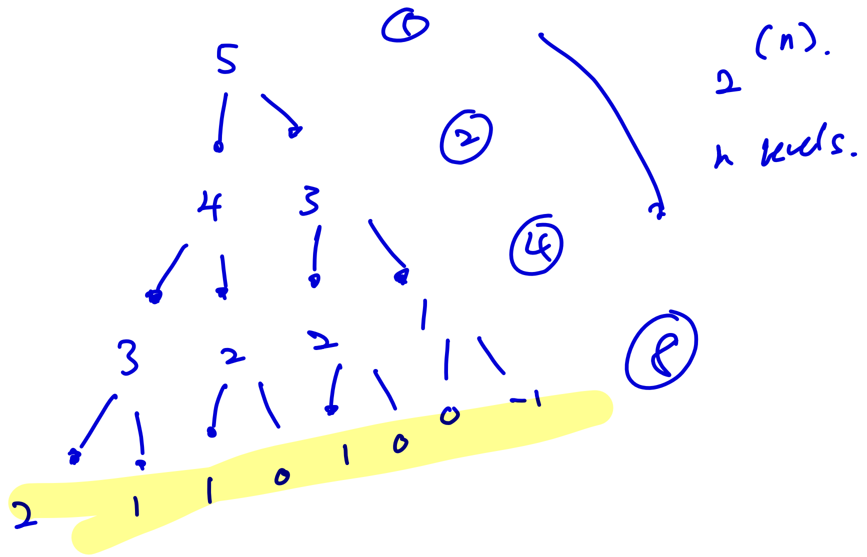


```
long count_inversion(long i, long n, const long rank[])
{
  long count = 0;
  for (long j = i + 1; j < n; j += 1) {
    if (rank[i] > rank[j]) {
      count ++;
    }
  }
  return count;
}

double kendall_tau(long n, const long rank[])
{
  long count = 0;
  for (long i = 0; i < n - 1; i += 1) {
    count += count_inversion(i, n, rank);
  }
  return 2.0 * count/(n * (n - 1));
}
```

Big O Notation (Efficiency)

5.

```
long fib(long n)
{
  if (n == 1 || n == 2) {
    return 1;
  }
  return fib(n-1) + fib(n-2); + fib(n)
}
```



Problem 22.1 Question

Order the following functions in the increasing order of rate of growth:

- $n!$,
- 2^n,
- $\log_{10} n$,
- $\ln n$,
- n^4,
- $n \ln n$,
- n,
- n^2,
- e^n,
- $\sqrt{n}$

Problem 22.1 Answer

search $\quad n^{\frac{1}{2}} \quad n^1 \quad$ best sorting. $\quad 1 \times 2 \times 3 \times 4 \times \dots \times n.$

$$\log_{10}(n) = ln(n) < \sqrt{n} < n < nlog(n) < n^2 < n^4 < 2^n < e^n < n! < n^n$$

$ln = log_{10}$

```
for (n) {
    for (n) {}
}
```

```
for ( n) {
for ( i*=2) { }
}
```

Problem 22.2 Question

What is the Big-O running time of the following code, in terms of n?

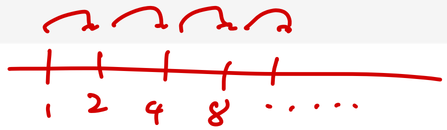
a)

```
for (long i = 0; i < n; i += 1) {
  for (long j = 0; j < n; j += 2) {
    cs1010_println_long(i + j);
  }
}
```

n

$\frac{n}{2}$

$\frac{n^2}{2} = n^2$



b)

```
for (long i = 1; i < n; i *= 2) {
  for (long j = 1; j < n; j *= 2) {
    cs1010_println_long(i + j);
  }
}
```

$\log(n)$

$\log(n)$

$(\log(n))^2.$

$O(\log^2(n))$

$i = 1$	$i = 2.$	$i = 3$
$k = 2.$	$k = 4$	$k = 8$
2^1	3^2	3^3

c)

```
long k = 1;
for (long j = 0; j < n; j += 1) {
  k *= 2;
  for (long i = 0; i < k; i += 1) {
    cs1010_println_long(i + j);
  }
}
```

3 ?

2

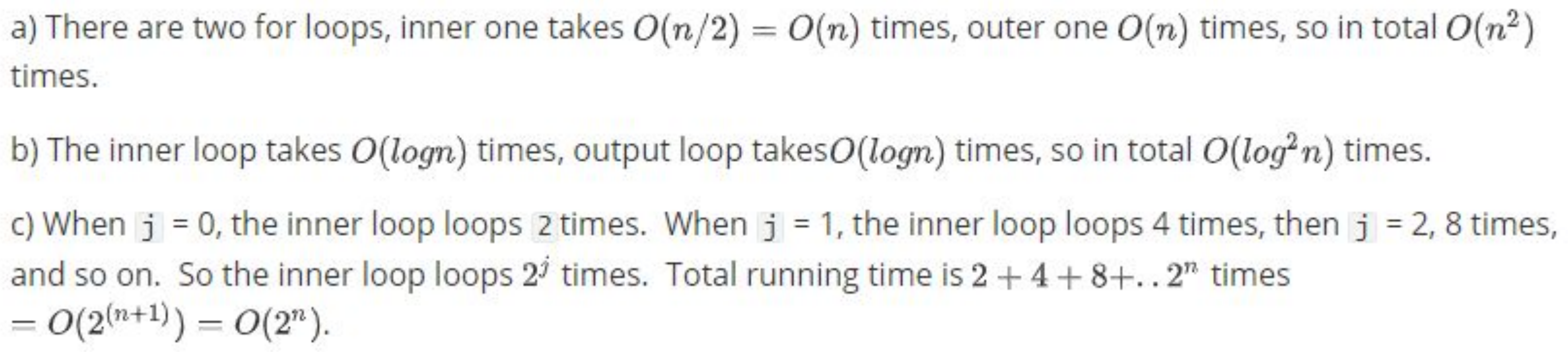
$2(2^n)$

$2 + 4 + 8 + \ldots + 2^n.$

$\underbrace{2 + 2^1 + 2^2 + 2^3 + \ldots + 2^{n-1}}_{2^n} + \boxed{2^n}$

2^n

Problem 22.2 Answer



a) There are two for loops, inner one takes $O(n/2) = O(n)$ times, outer one $O(n)$ times, so in total $O(n^2)$ times.

b) The inner loop takes $O(logn)$ times, output loop takes$O(logn)$ times, so in total $O(log^2 n)$ times.

c) When `j` = 0, the inner loop loops `2` times. When `j` = 1, the inner loop loops 4 times, then `j` = 2, 8 times, and so on. So the inner loop loops 2^j times. Total running time is $2 + 4 + 8 + .. 2^n$ times $= O(2^{(n+1)}) = O(2^n)$.

Assignment 4

- Remember to handle NULL condition when calling malloc and calloc
- Good effort documenting but try to summarise the documentation
  - Normally we do not describe how the code work
  - We mention what it does generally and what to send in

Assignment 5: Social

- Finding an efficient algorithm for Contact first
- Use this algorithm to work on social
- Prepare Question.md for this discussion

Midterms Question 13 - 16

Consider the following C function.

$(i \% 3 == 0) \rightarrow (i-1)/3 + 1 = (i)/3$

```
1    long f(long n) {
2        long i = 1;
3        long res = 0;
4        while (i <= n) {
5            if (i % 3 == 0) {
6                res += 1;
7            }
8            if (i % 7 == 0) {
9                res += 1;
10           }
11           i += 1;
12       }
13       return res;
14   }
```

0

?

0

$i/3$

$res = \boxed{(i-1)/3} + (i-1)/7 \quad \boxed{+1}$

0

$i/7$

$res = (i-1)/3 + (i-1)/7 + 1 + 1$

$res = i/3 + i/7$

$res = (i-1)/3 + (i-1)/7$

0

$$\boxed{res = (i-1)/3 + (i-1)/7}$$

13. (3 points) Specify the loop invariant that relates `res` and `i`.

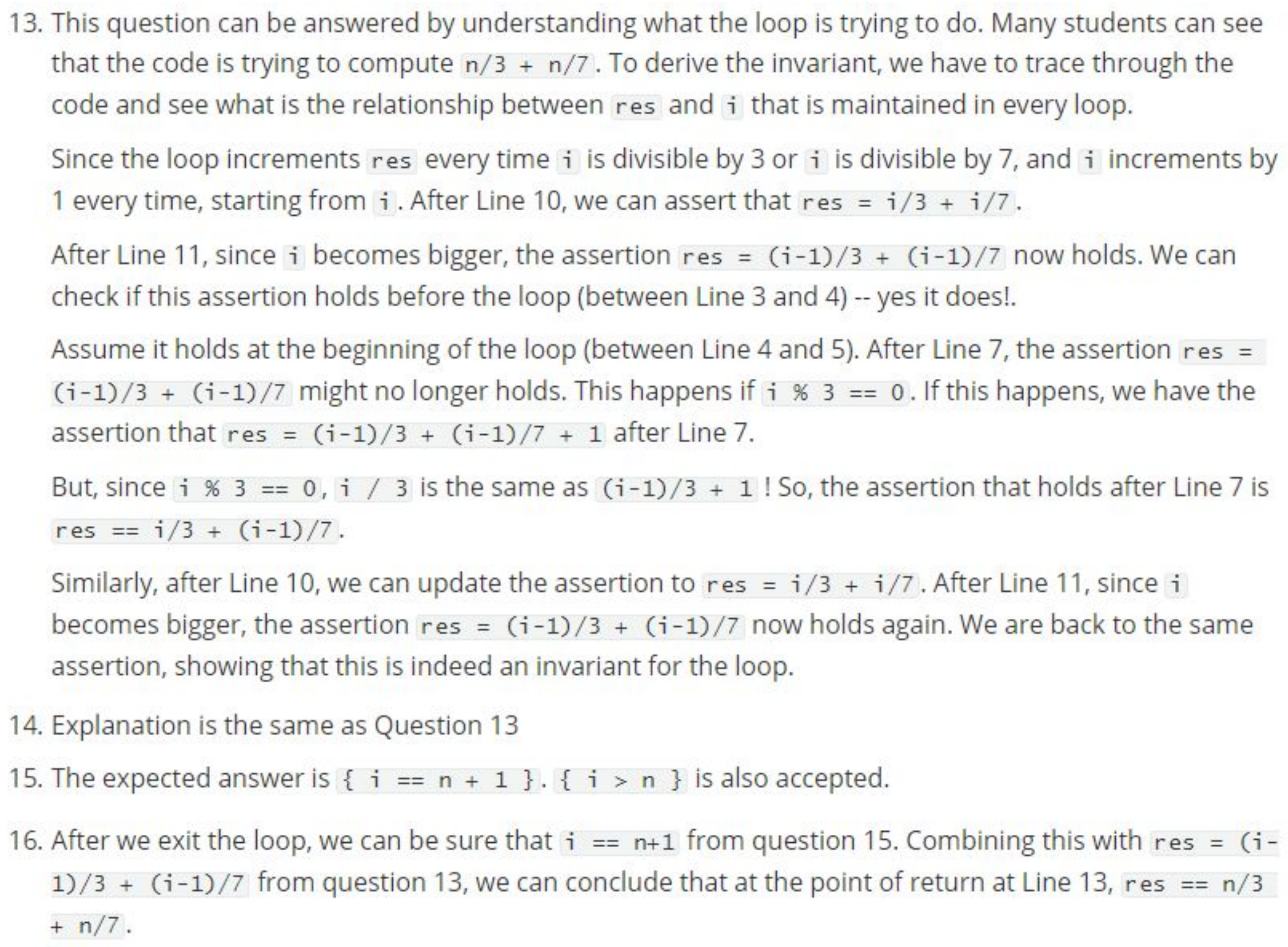
14. (4 points) Argue why the loop invariant holds between Line 11 and Line 12.

15. (2 points) What is an assertion relating `n` and `i` between Line 12 and Line 13? $\{ i == n+1 \}$

16. (2 points) What is the output of the function in terms of `n`? Explain your answer based on the loop invariant (part a) and the assertion (part b) you have given above.

$res = (n)/3 + (n)/7 \quad , \quad n = i-1$

Midterms Answer 13 - 16



13. This question can be answered by understanding what the loop is trying to do. Many students can see that the code is trying to compute `n/3 + n/7`. To derive the invariant, we have to trace through the code and see what is the relationship between `res` and `i` that is maintained in every loop.

    Since the loop increments `res` every time `i` is divisible by 3 or `i` is divisible by 7, and `i` increments by 1 every time, starting from `i`. After Line 10, we can assert that `res = i/3 + i/7`.

    After Line 11, since `i` becomes bigger, the assertion `res = (i-1)/3 + (i-1)/7` now holds. We can check if this assertion holds before the loop (between Line 3 and 4) -- yes it does!.

    Assume it holds at the beginning of the loop (between Line 4 and 5). After Line 7, the assertion `res = (i-1)/3 + (i-1)/7` might no longer holds. This happens if `i % 3 == 0`. If this happens, we have the assertion that `res = (i-1)/3 + (i-1)/7 + 1` after Line 7.

    But, since `i % 3 == 0`, `i / 3` is the same as `(i-1)/3 + 1`! So, the assertion that holds after Line 7 is `res == i/3 + (i-1)/7`.

    Similarly, after Line 10, we can update the assertion to `res = i/3 + i/7`. After Line 11, since `i` becomes bigger, the assertion `res = (i-1)/3 + (i-1)/7` now holds again. We are back to the same assertion, showing that this is indeed an invariant for the loop.

14. Explanation is the same as Question 13
15. The expected answer is `{ i == n + 1 }`. `{ i > n }` is also accepted.
16. After we exit the loop, we can be sure that `i == n+1` from question 15. Combining this with `res = (i-1)/3 + (i-1)/7` from question 13, we can conclude that at the point of return at Line 13, `res == n/3 + n/7`.