CSE 3100: Systems Programming

Exam 1 Review

Exam Format (1)

- You can ONLY take the exam in your own lab section.
- This is a 110-minute exam.
- During the exam, you cannot communicate with and/or obtain help from people other than TAs and instructors on exam questions. Particularly, you cannot use any messaging applications.
- During the exam, you can only access data/files on HuskyCT, your Virtual Machine and handwritten or typed notes (non-digital).
- You cannot access data/files on other computers/servers, you cannot use ChatGPT, you cannot use your phone.

Exam Format (2)

- Exam is two parts: multiple choice and coding. The multiple choice will be given on HuskyCT.
- You must submit your code on Gradescope. We only grade submissions we have received on Gradescope.
- You cannot email the TA/professor exam submissions.
- **Note if your code does not compile, you will lose at least half of the points.**
- After the exam, you cannot discuss/disclose exam problems and/or share your code with students who have not taken the exam.

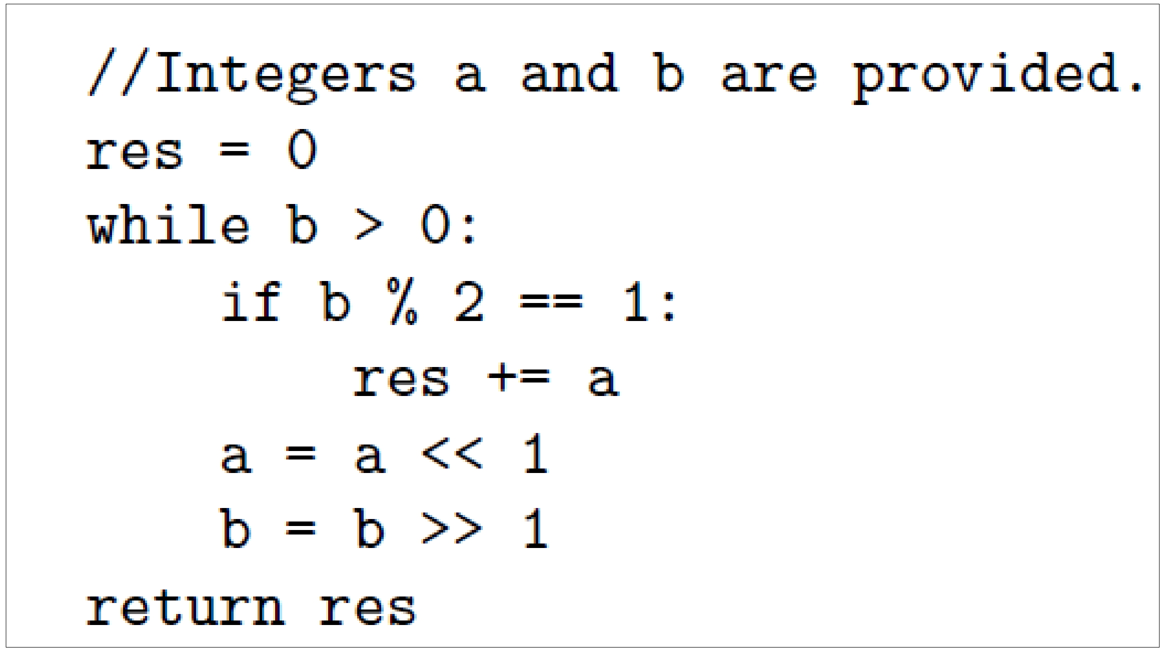
Answers to the Practice Exam

Short Answer Question

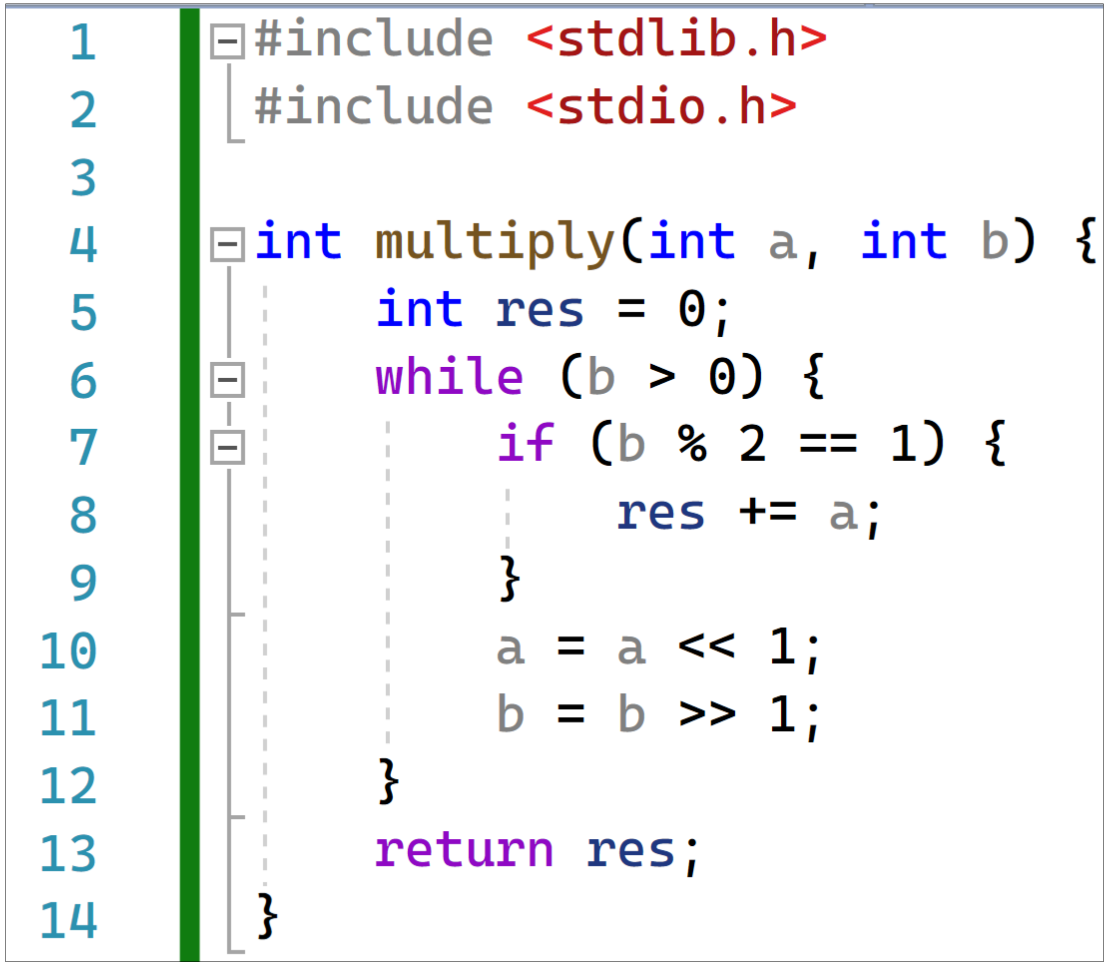
The "Russian Peasant's Algorithm" uses division and multiplication by 2 (which can be done using bit operations) repeatedly to calculate the product of two integers. This is the pseudo-code to the algorithm; please translate it to C, and put it in the function `multiply` in `multiply.c`:

```
//Integers a and b are provided.
res = 0
while b > 0:
    if b % 2 == 1:
        res += a
    a = a << 1
    b = b >> 1
return res
```

Short Answer Question



```
//Integers a and b are provided.
res = 0
while b > 0:
    if b % 2 == 1:
        res += a
    a = a << 1
    b = b >> 1
return res
```

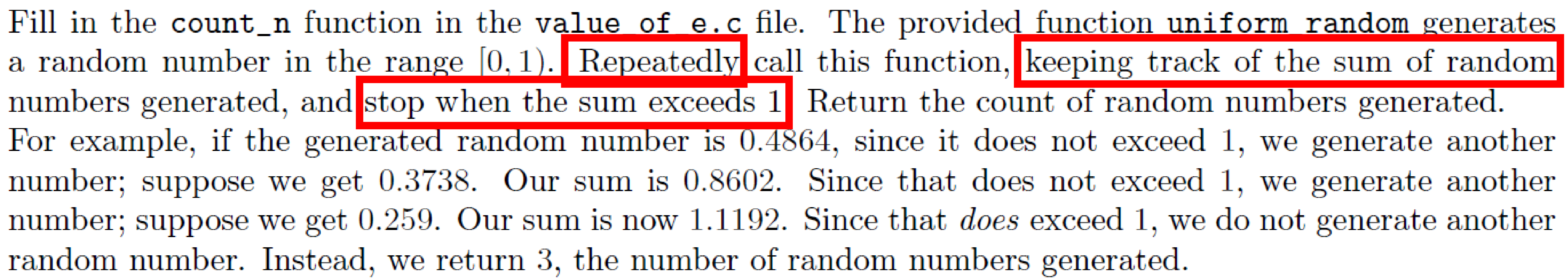


```c
#include <stdlib.h>
#include <stdio.h>

int multiply(int a, int b) {
    int res = 0;
    while (b > 0) {
        if (b % 2 == 1) {
            res += a;
        }
        a = a << 1;
        b = b >> 1;
    }
    return res;
}
```

Code Problem 1

Fill in the `count_n` function in the `value_of_e.c` file. The provided function `uniform_random` generates a random number in the range $[0, 1)$. Repeatedly call this function, keeping track of the sum of random numbers generated, and stop when the sum exceeds 1. Return the count of random numbers generated.
For example, if the generated random number is 0.4864, since it does not exceed 1, we generate another number; suppose we get 0.3738. Our sum is 0.8602. Since that does not exceed 1, we generate another number; suppose we get 0.259. Our sum is now 1.1192. Since that *does* exceed 1, we do not generate another random number. Instead, we return 3, the number of random numbers generated.

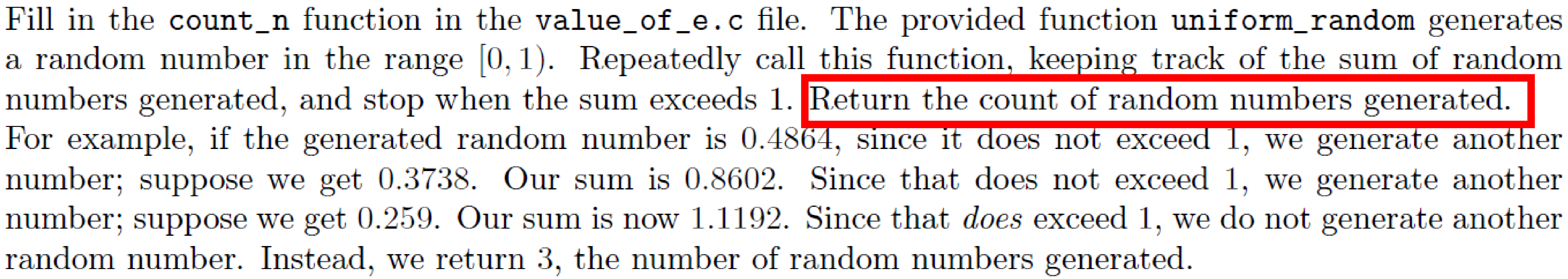
Code Problem 1



Fill in the `count_n` function in the `value_of_e.c` file. The provided function `uniform_random` generates a random number in the range $[0, 1)$. Repeatedly call this function, keeping track of the sum of random numbers generated, and stop when the sum exceeds 1. Return the count of random numbers generated. For example, if the generated random number is 0.4864, since it does not exceed 1, we generate another number; suppose we get 0.3738. Our sum is 0.8602. Since that does not exceed 1, we generate another number; suppose we get 0.259. Our sum is now 1.1192. Since that *does* exceed 1, we do not generate another random number. Instead, we return 3, the number of random numbers generated.

First part of the problem: Keep track of the sum. Stop when the sum is bigger than 1.

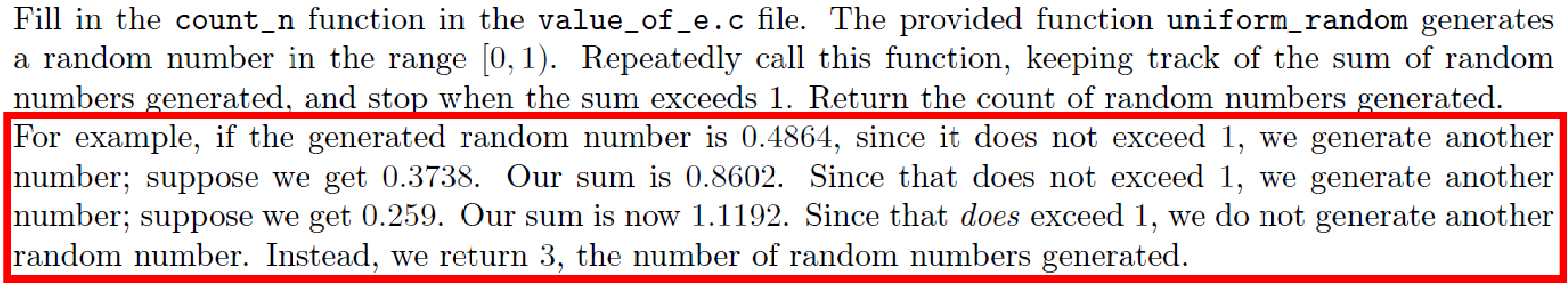
Code Problem 1



Fill in the `count_n` function in the `value_of_e.c` file. The provided function `uniform_random` generates a random number in the range $[0, 1)$. Repeatedly call this function, keeping track of the sum of random numbers generated, and stop when the sum exceeds 1. Return the count of random numbers generated. For example, if the generated random number is 0.4864, since it does not exceed 1, we generate another number; suppose we get 0.3738. Our sum is 0.8602. Since that does not exceed 1, we generate another number; suppose we get 0.259. Our sum is now 1.1192. Since that *does* exceed 1, we do not generate another random number. Instead, we return 3, the number of random numbers generated.

Second Pass: I need to return how many times we add to sum.

Code Problem 1



Fill in the `count_n` function in the `value_of_e.c` file. The provided function `uniform_random` generates a random number in the range $[0, 1)$. Repeatedly call this function, keeping track of the sum of random numbers generated, and stop when the sum exceeds 1. Return the count of random numbers generated. For example, if the generated random number is 0.4864, since it does not exceed 1, we generate another number; suppose we get 0.3738. Our sum is 0.8602. Since that does not exceed 1, we generate another number; suppose we get 0.259. Our sum is now 1.1192. Since that *does* exceed 1, we do not generate another random number. Instead, we return 3, the number of random numbers generated.

Third part: keep looping until sum is bigger than 1.

Code Problem 1: Put all the parts together

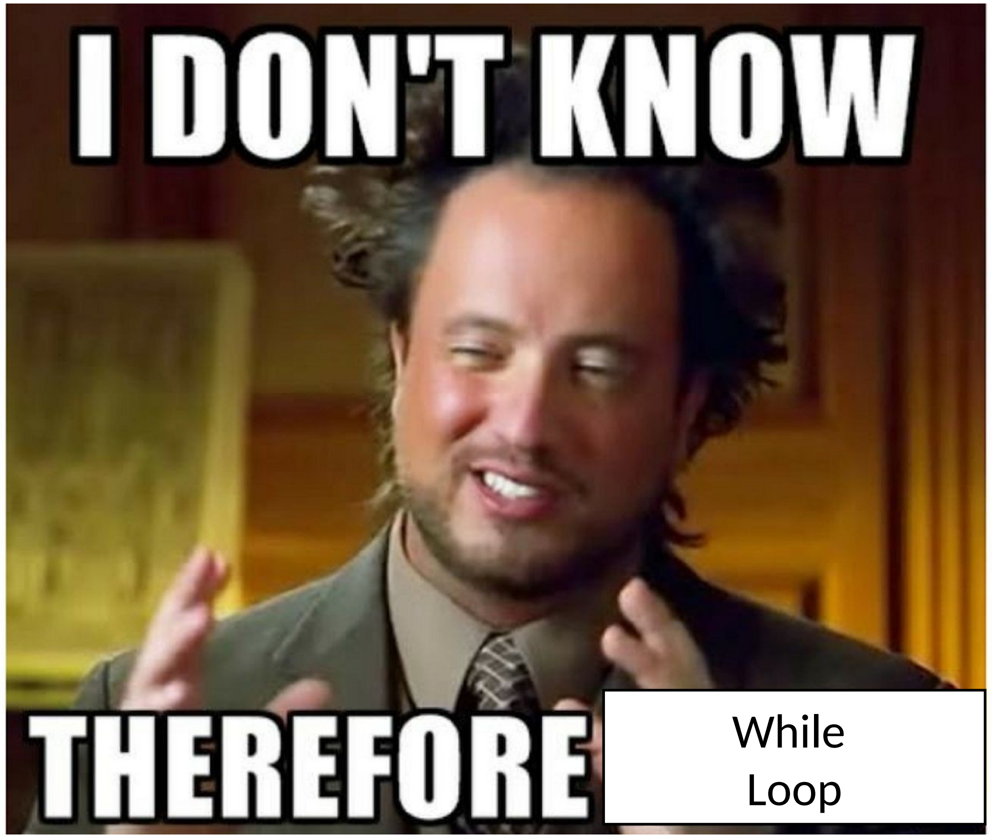
First part of the problem: Keep track of the sum. Stop when the sum is bigger than 1.

Second Pass: I need to return how many times we add to sum.

Third part: keep looping until sum is bigger than 1.

Question: Do I know exactly how many times I am going to be looping?

Question: *Do I know exactly how many times I am going to be looping?*



Third part: keep looping until sum is bigger than 1.

```
While (sum <= 1){

                                        //do something
}
```

Third part: keep looping until sum is bigger than 1.

Second Pass: I need to return how many times we add to sum.

```
int count = 0;
While (sum <= 1){

                    //do something
                    count++;
}
```

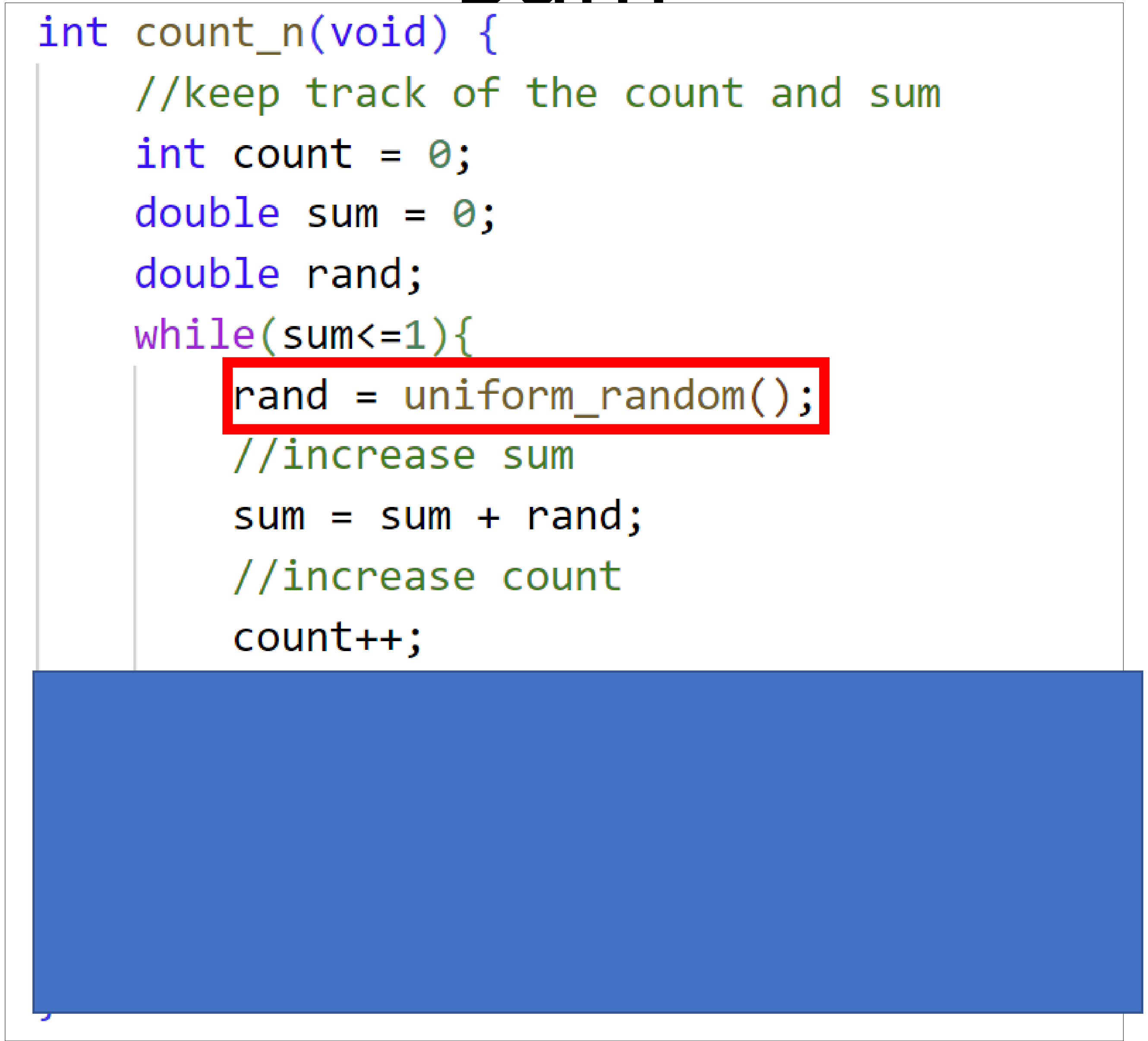
Third part: keep looping until sum is bigger than 1.

Second Pass: I need to return how many times we add to sum.

First part of the problem: Keep track of the sum. Stop when the sum is bigger than 1.

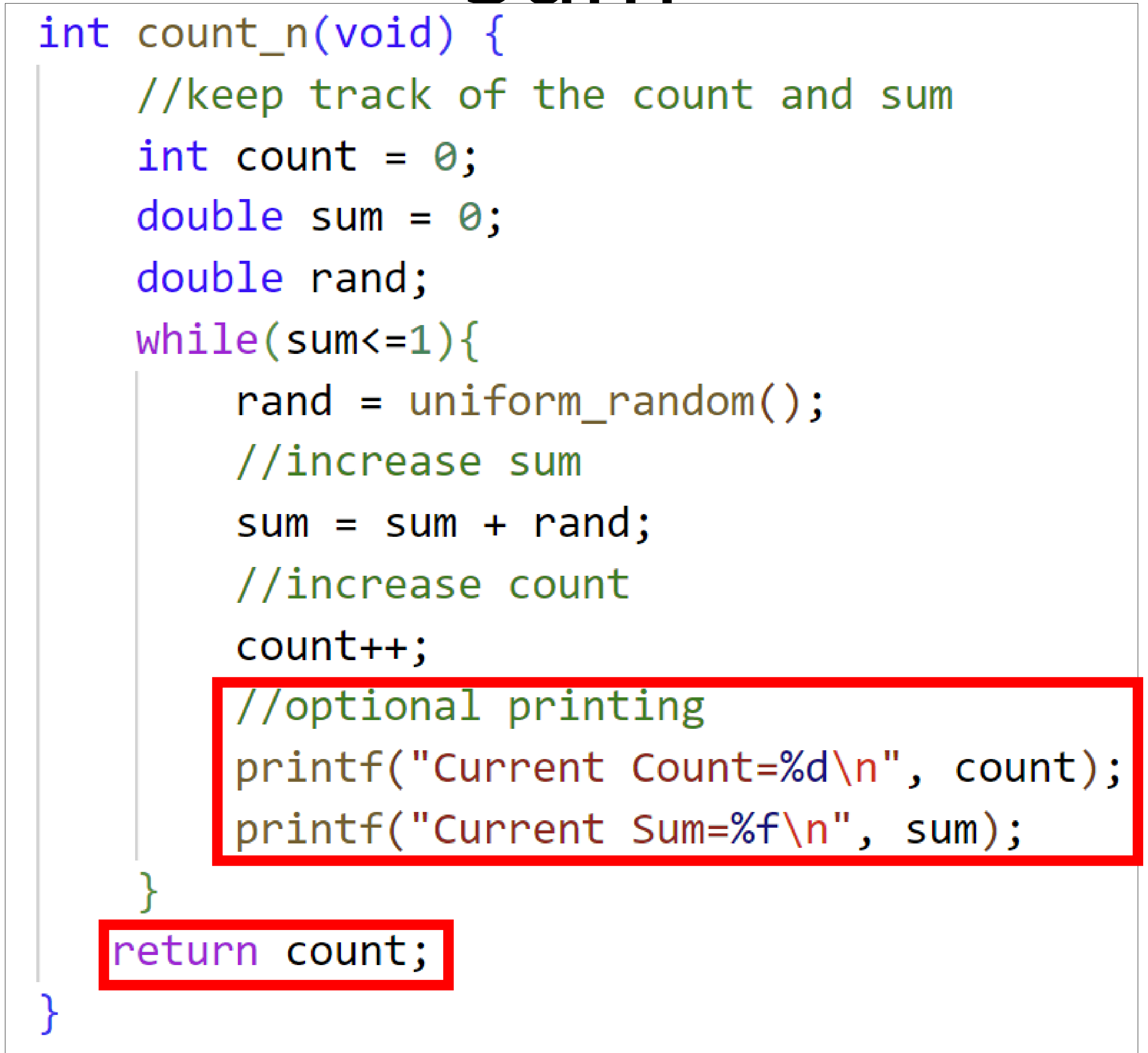
```
int count = 0;
int sum = 0;
double rand;
While (sum <= 1){

                                        sum = sum + rand;
                                        count++;

}
```

Lastly: add in the incrementing for sum



```
int count_n(void) {
    //keep track of the count and sum
    int count = 0;
    double sum = 0;
    double rand;
    while(sum<=1){
        rand = uniform_random();
        //increase sum
        sum = sum + rand;
        //increase count
        count++;
[redacted]
```

Lastly: add in the incrementing for sum



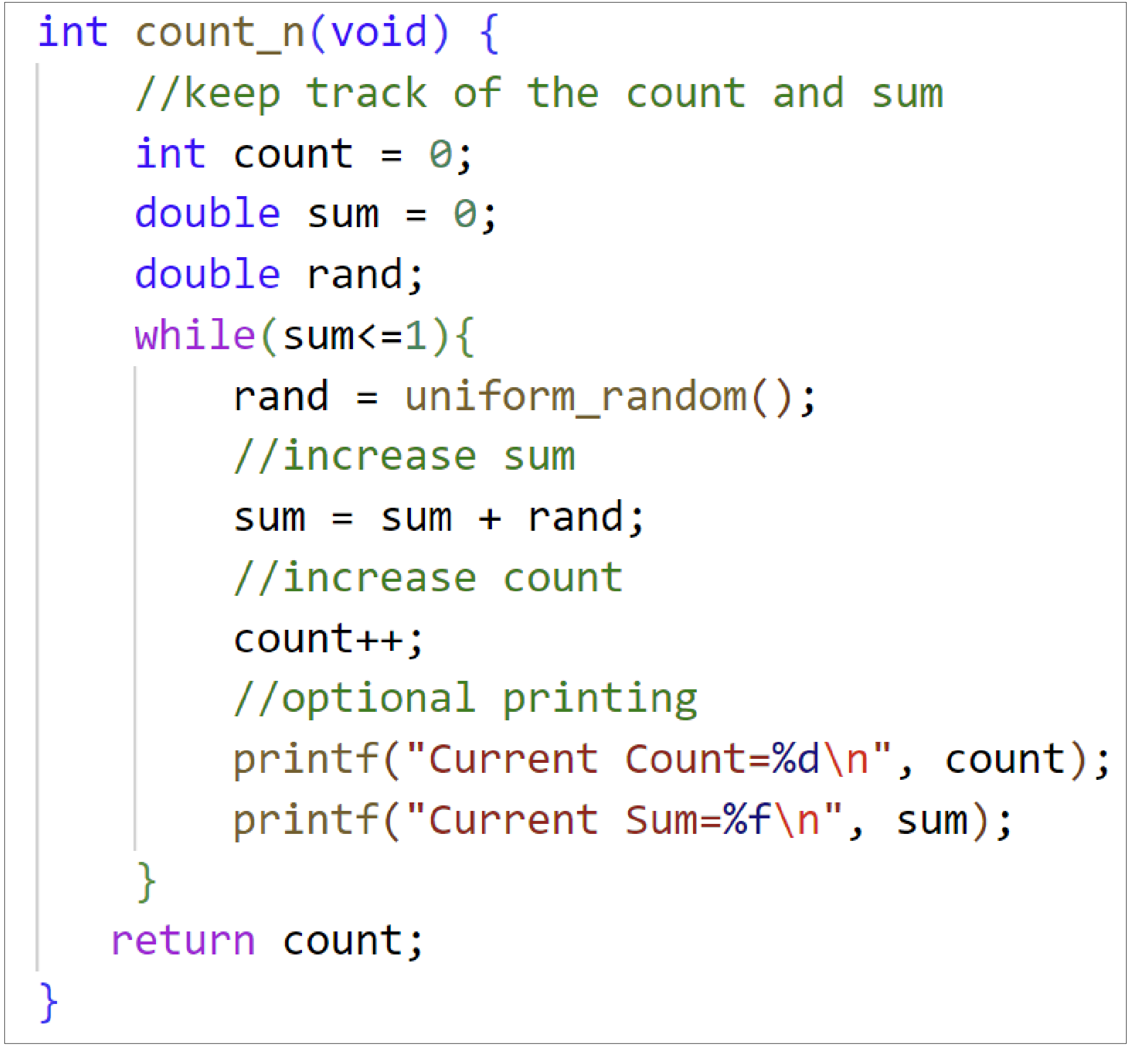
```
int count_n(void) {
    //keep track of the count and sum
    int count = 0;
    double sum = 0;
    double rand;
    while(sum<=1){
        rand = uniform_random();
        //increase sum
        sum = sum + rand;
        //increase count
        count++;
        //optional printing
        printf("Current Count=%d\n", count);
        printf("Current Sum=%f\n", sum);
    }
    return count;
}
```

An Example Run

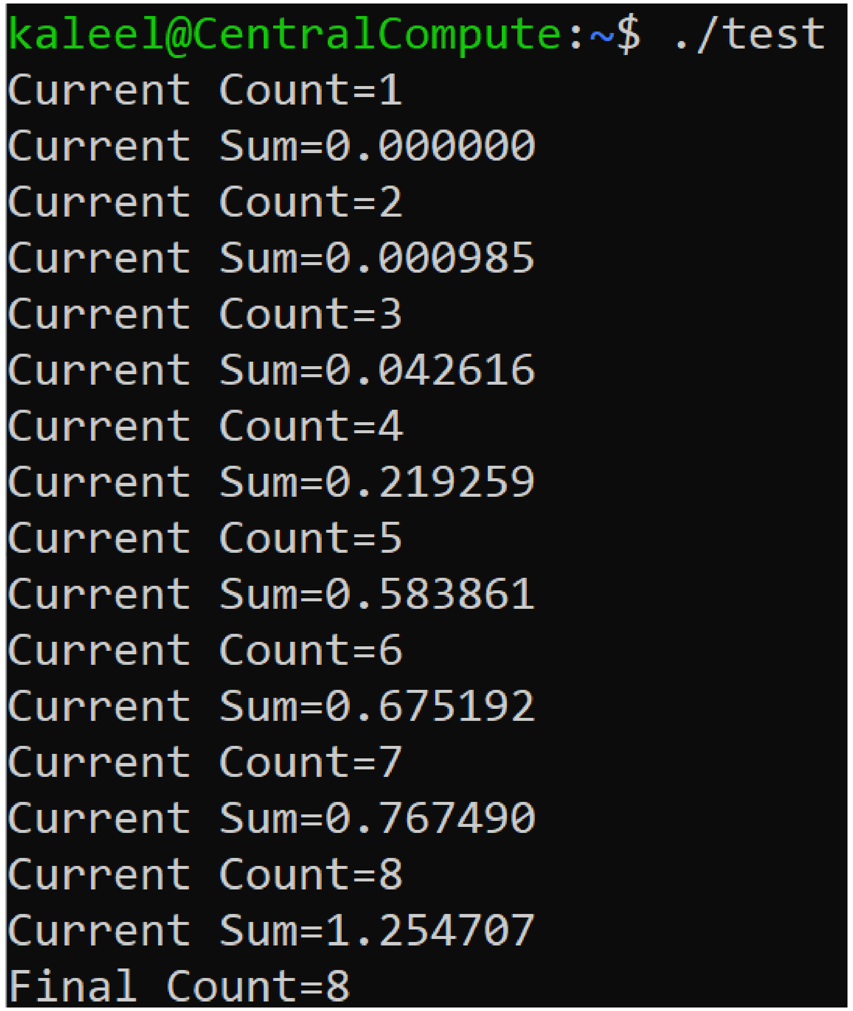
```c
int main() {
    int count = count_n();
    printf("Final Count=%d\n", count);
    return 0;
}
```

```c
int count_n(void) {
    //keep track of the count and sum
    int count = 0;
    double sum = 0;
    double rand;
    while(sum<=1){
        rand = uniform_random();
        //increase sum
        sum = sum + rand;
        //increase count
        count++;
        //optional printing
        printf("Current Count=%d\n", count);
        printf("Current Sum=%f\n", sum);
    }
   return count;
}
```

An Example Run



```c
int count_n(void) {
    //keep track of the count and sum
    int count = 0;
    double sum = 0;
    double rand;
    while(sum<=1){
        rand = uniform_random();
        //increase sum
        sum = sum + rand;
        //increase count
        count++;
        //optional printing
        printf("Current Count=%d\n", count);
        printf("Current Sum=%f\n", sum);
    }
   return count;
}
```



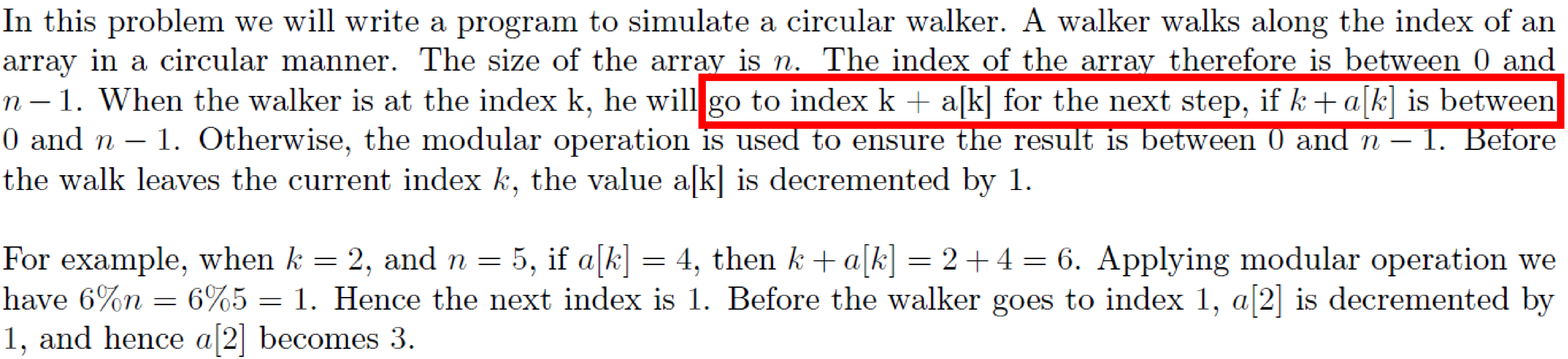
```
kaleel@CentralCompute:~$ ./test
Current Count=1
Current Sum=0.000000
Current Count=2
Current Sum=0.000985
Current Count=3
Current Sum=0.042616
Current Count=4
Current Sum=0.219259
Current Count=5
Current Sum=0.583861
Current Count=6
Current Sum=0.675192
Current Count=7
Current Sum=0.767490
Current Count=8
Current Sum=1.254707
Final Count=8
```

Code Problem 2

In this problem we will write a program to simulate a circular walker. A walker walks along the index of an array in a circular manner. The size of the array is n. The index of the array therefore is between 0 and $n-1$. When the walker is at the index k, he will go to index k + a[k] for the next step, if $k+a[k]$ is between 0 and $n-1$. Otherwise, the modular operation is used to ensure the result is between 0 and $n-1$. Before the walk leaves the current index k, the value a[k] is decremented by 1.

For example, when $k=2$, and $n=5$, if $a[k]=4$, then $k+a[k]=2+4=6$. Applying modular operation we have $6\%n=6\%5=1$. Hence the next index is 1. Before the walker goes to index 1, $a[2]$ is decremented by 1, and hence $a[2]$ becomes 3.

Code Problem 2

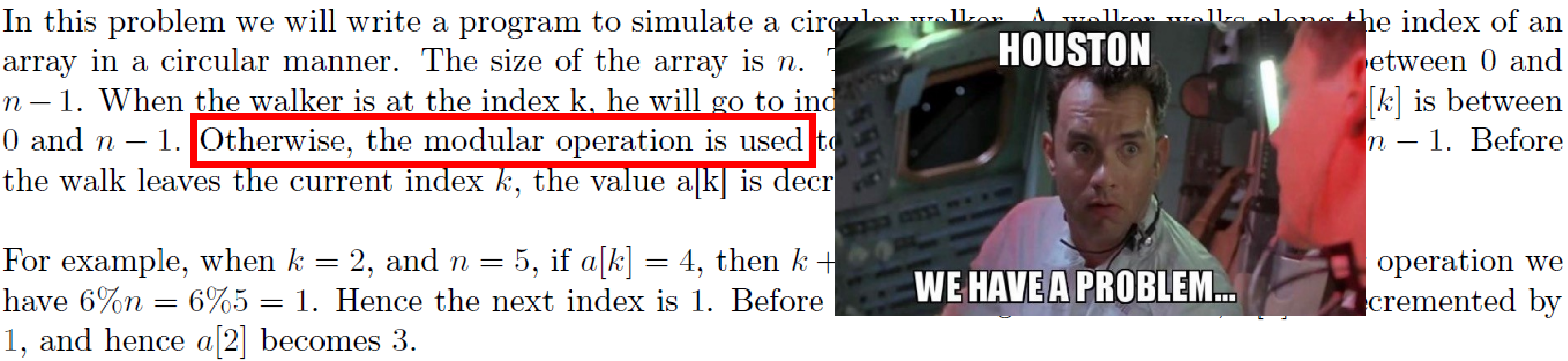


In this problem we will write a program to simulate a circular walker. A walker walks along the index of an array in a circular manner. The size of the array is n. The index of the array therefore is between 0 and $n-1$. When the walker is at the index k, he will go to index k + a[k] for the next step, if $k + a[k]$ is between 0 and $n-1$. Otherwise, the modular operation is used to ensure the result is between 0 and $n-1$. Before the walk leaves the current index k, the value a[k] is decremented by 1.

For example, when $k = 2$, and $n = 5$, if $a[k] = 4$, then $k + a[k] = 2 + 4 = 6$. Applying modular operation we have $6\%n = 6\%5 = 1$. Hence the next index is 1. Before the walker goes to index 1, $a[2]$ is decremented by 1, and hence $a[2]$ becomes 3.

First part of the problem: Compute k + a[k] IF it is bigger than n, we have a problem.

Code Problem 2

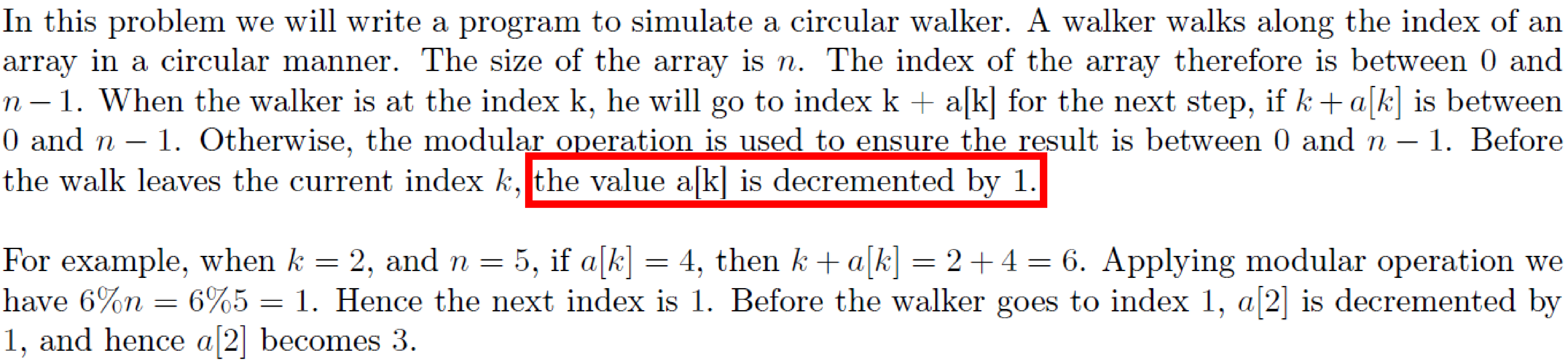


In this problem we will write a program to simulate a circular walker. A walker walks along the index of an array in a circular manner. The size of the array is n. T etween 0 and $n-1$. When the walker is at the index k, he will go to ind $[k]$ is between 0 and $n-1$. Otherwise, the modular operation is used to $n-1$. Before the walk leaves the current index k, the value a[k] is decr

For example, when $k=2$, and $n=5$, if $a[k]=4$, then $k+$ operation we have $6\%n = 6\%5 = 1$. Hence the next index is 1. Before cremented by 1, and hence $a[2]$ becomes 3.

Second part of the problem: Assume k + a[k] is bigger than n: use modulo to reduce

Code Problem 2



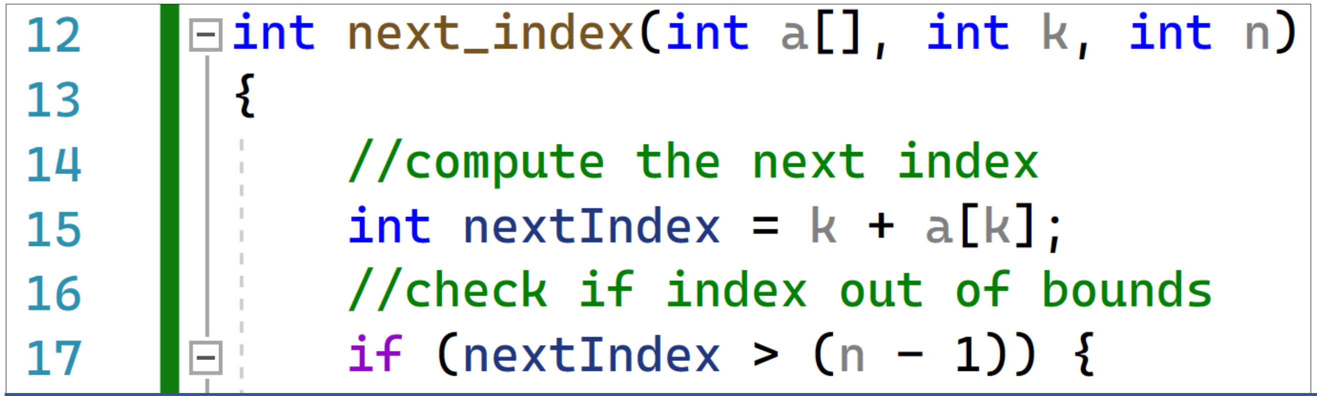
In this problem we will write a program to simulate a circular walker. A walker walks along the index of an array in a circular manner. The size of the array is n. The index of the array therefore is between 0 and $n-1$. When the walker is at the index k, he will go to index k + a[k] for the next step, if $k+a[k]$ is between 0 and $n-1$. Otherwise, the modular operation is used to ensure the result is between 0 and $n-1$. Before the walk leaves the current index k, the value a[k] is decremented by 1.

For example, when $k=2$, and $n=5$, if $a[k]=4$, then $k+a[k]=2+4=6$. Applying modular operation we have $6\%n=6\%5=1$. Hence the next index is 1. Before the walker goes to index 1, $a[2]$ is decremented by 1, and hence $a[2]$ becomes 3.

Third part of the problem: reduce a[k] by one regardless.

Code Problem 2

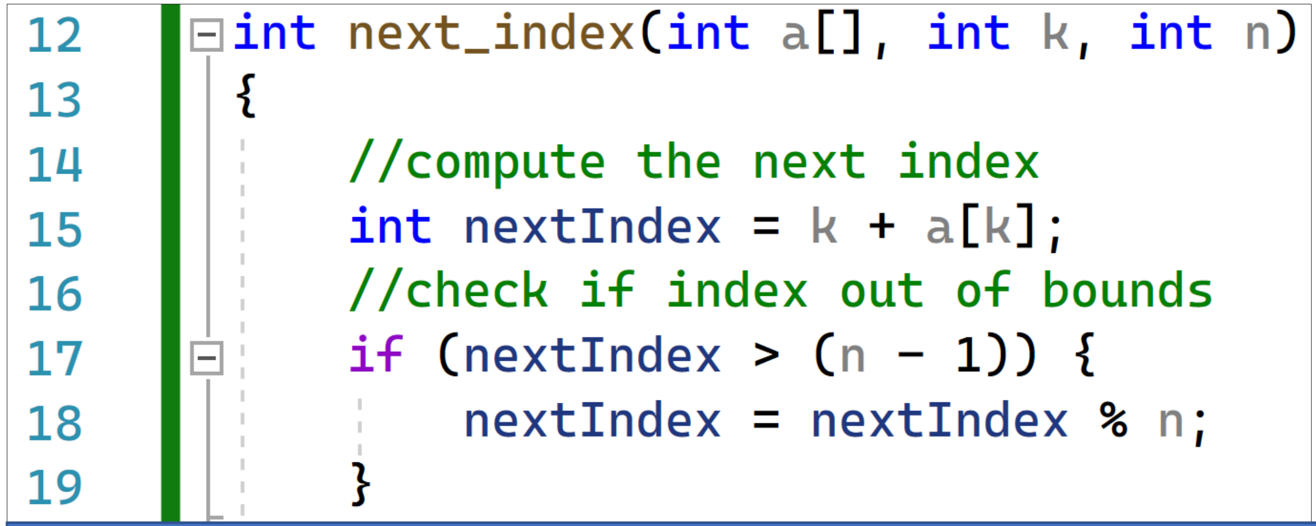
First part of the problem: Compute k + a[k] IF it is bigger than n, we have a problem.



```
int next_index(int a[], int k, int n)
{
    //compute the next index
    int nextIndex = k + a[k];
    //check if index out of bounds
    if (nextIndex > (n - 1)) {
```

Code Problem 2

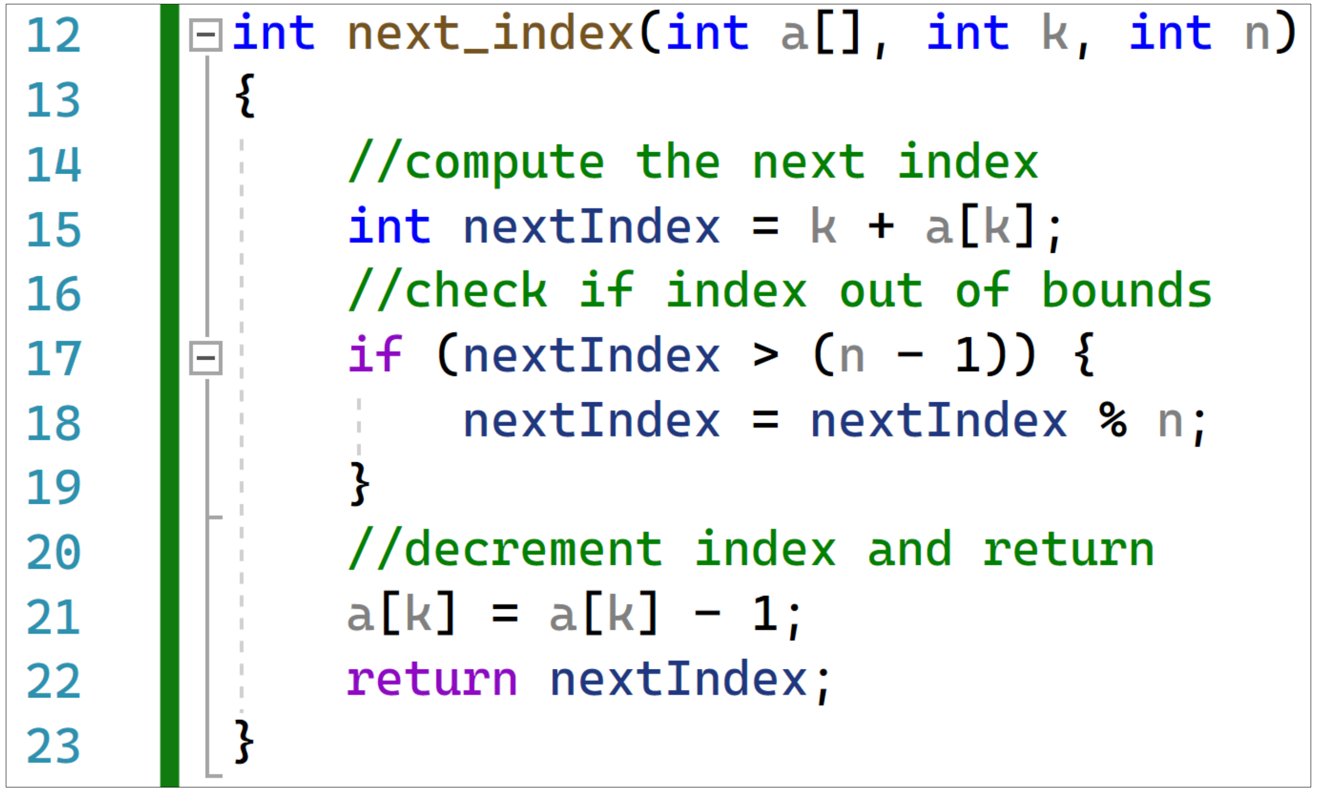
Second part of the problem: Assume k + a[k] is bigger than n: use modulo to reduce



```
int next_index(int a[], int k, int n)
{
    //compute the next index
    int nextIndex = k + a[k];
    //check if index out of bounds
    if (nextIndex > (n - 1)) {
        nextIndex = nextIndex % n;
    }
```

Code Problem 3

Third part of the problem: reduce a[k] by one regardless.



```
int next_index(int a[], int k, int n)
{
    //compute the next index
    int nextIndex = k + a[k];
    //check if index out of bounds
    if (nextIndex > (n - 1)) {
        nextIndex = nextIndex % n;
    }
    //decrement index and return
    a[k] = a[k] - 1;
    return nextIndex;
}
```

Side Note: Do we actually need the if statement?

- What happens if we are in 0 to n-1? Well we don't need to reduce.
- But what if we still apply module operation? E.g.

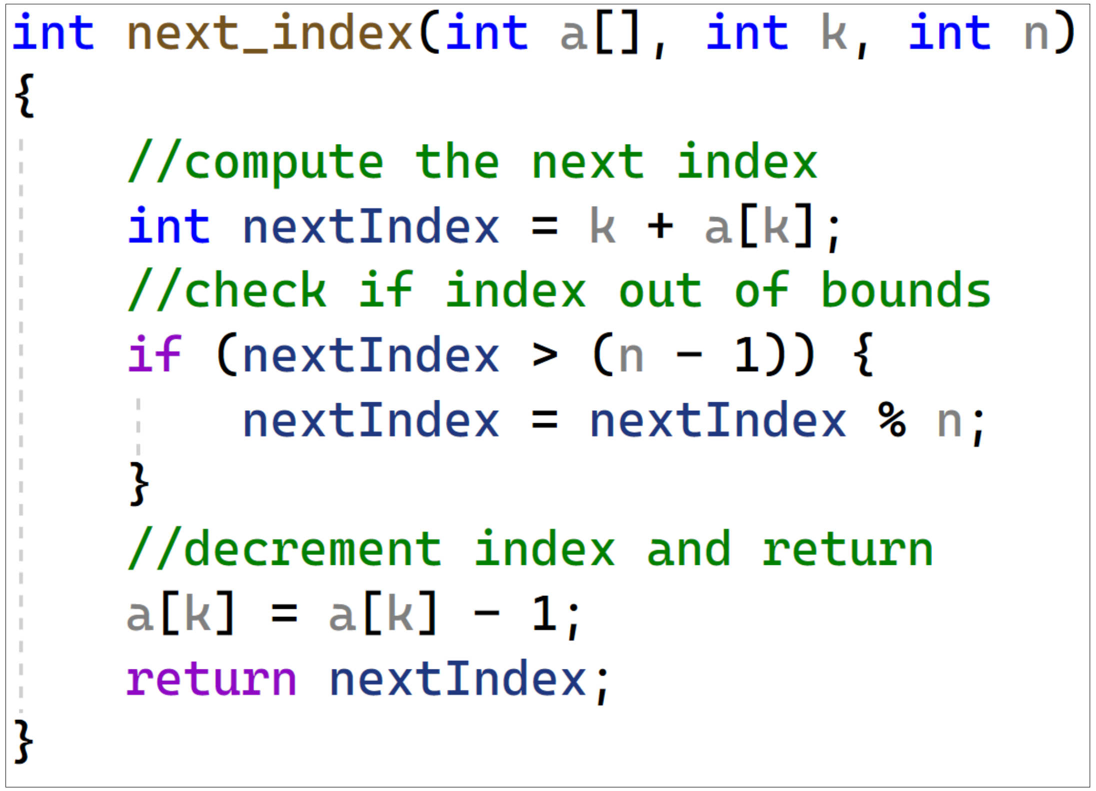
nextIndex = 3

n = 5

3 % 5 = 3

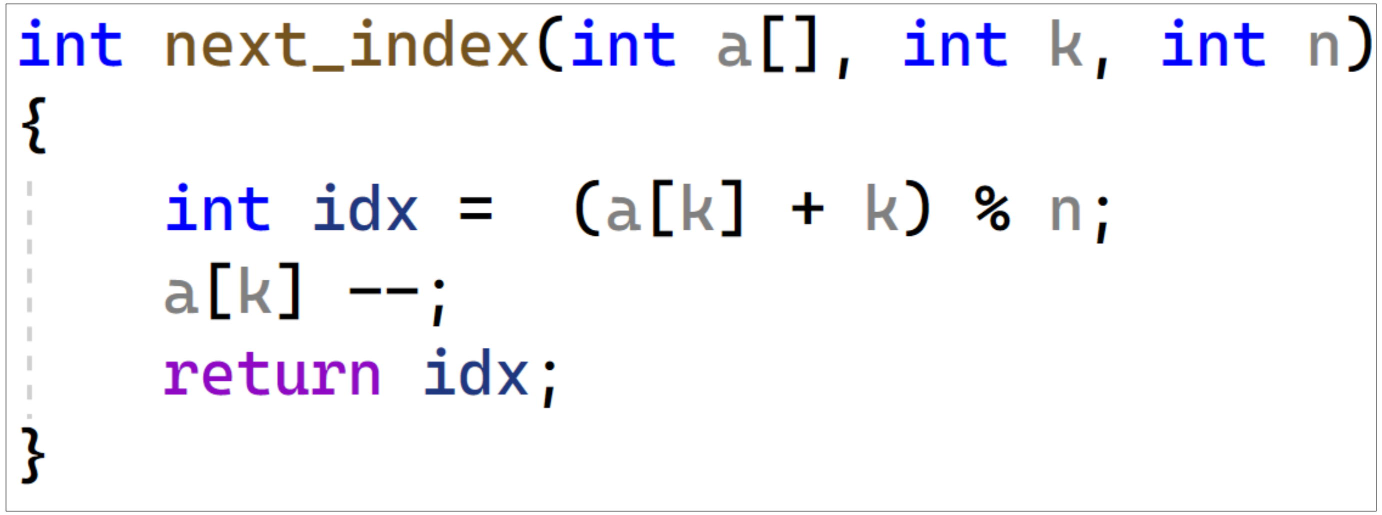
An even simpler implementation:

Original Implementation



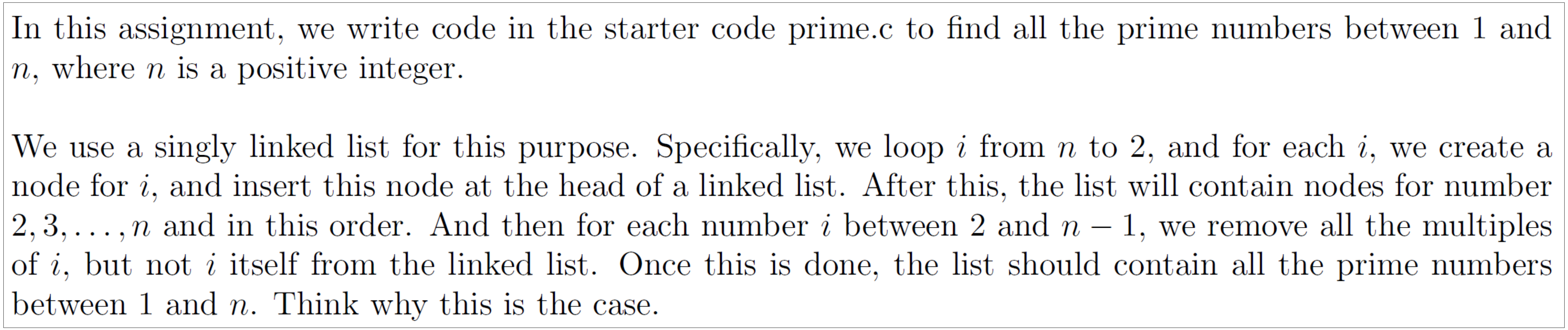
```
int next_index(int a[], int k, int n)
{
    //compute the next index
    int nextIndex = k + a[k];
    //check if index out of bounds
    if (nextIndex > (n - 1)) {
        nextIndex = nextIndex % n;
    }
    //decrement index and return
    a[k] = a[k] - 1;
    return nextIndex;
}
```

Simple Implementation



```
int next_index(int a[], int k, int n)
{
    int idx =  (a[k] + k) % n;
    a[k] --;
    return idx;
}
```

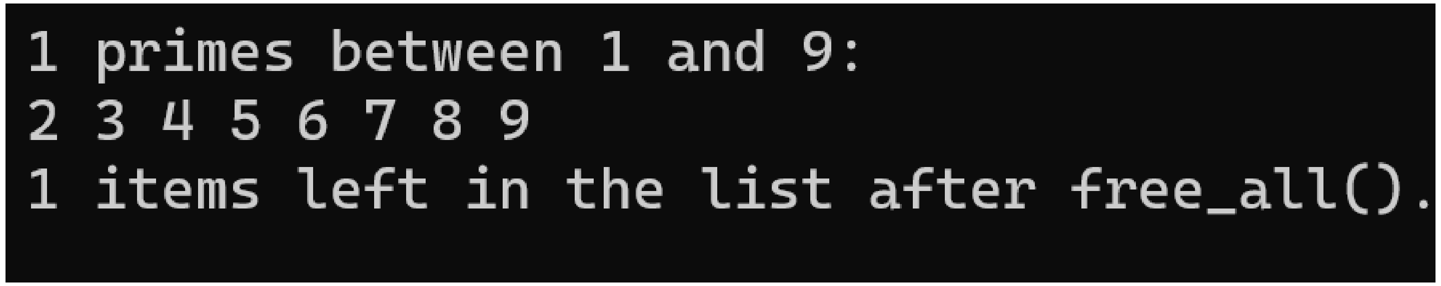
Code Problem 4



In this assignment, we write code in the starter code prime.c to find all the prime numbers between 1 and n, where n is a positive integer.

We use a singly linked list for this purpose. Specifically, we loop i from n to 2, and for each i, we create a node for i, and insert this node at the head of a linked list. After this, the list will contain nodes for number $2, 3, \ldots, n$ and in this order. And then for each number i between 2 and $n - 1$, we remove all the multiples of i, but not i itself from the linked list. Once this is done, the list should contain all the prime numbers between 1 and n. Think why this is the case.

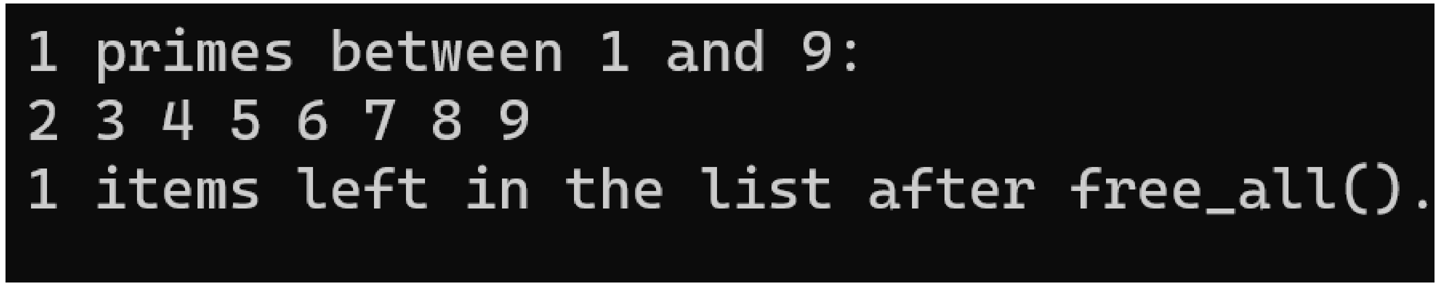
Starter Code Output:



```
1 primes between 1 and 9:
2 3 4 5 6 7 8 9
1 items left in the list after free_all().
```

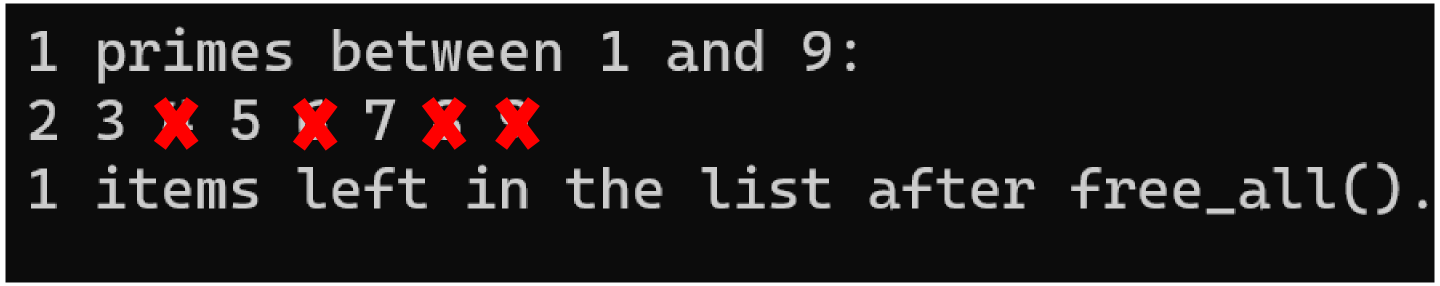
First step to solve: Think about ideal output

Starter code output:



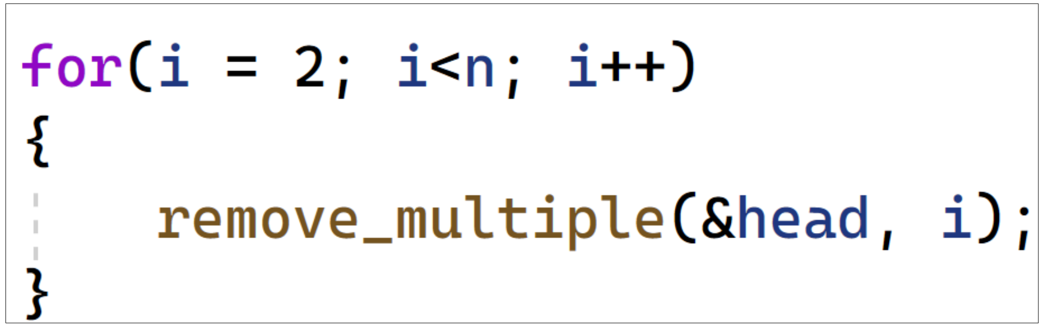
```
1 primes between 1 and 9:
2 3 4 5 6 7 8 9
1 items left in the list after free_all().
```

What output we want to see:

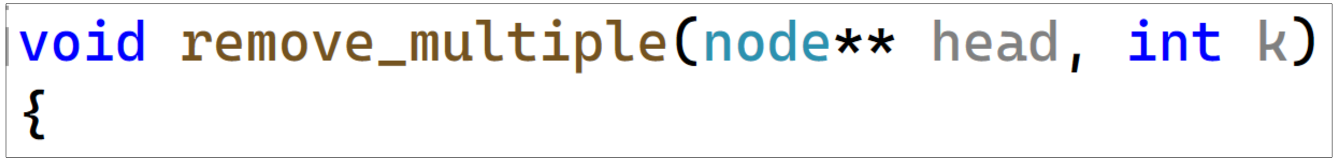


```
1 primes between 1 and 9:
2 3 ~~4~~ 5 ~~6~~ 7 ~~8~~ ~~9~~
1 items left in the list after free_all().
```

Second Step To Solve:
See what is missing in the starter code



```
for(i = 2; i<n; i++)
{
    remove_multiple(&head, i);
}
```



```
void remove_multiple(node** head, int k)
{
```

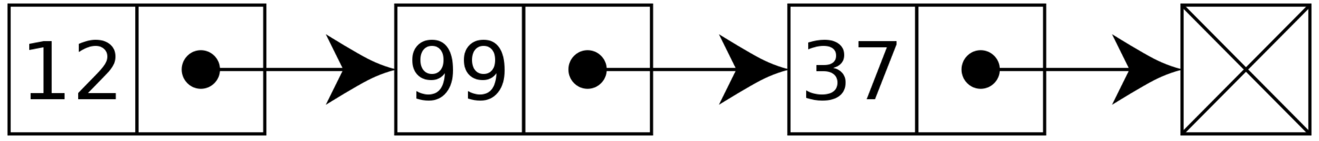
- Consider the steps for the solution: remove all prime numbers from a linked list:

1. Iterate through the linked list.
2. IF the node is divisible by the input k AND it is not equal to k we should remove it.

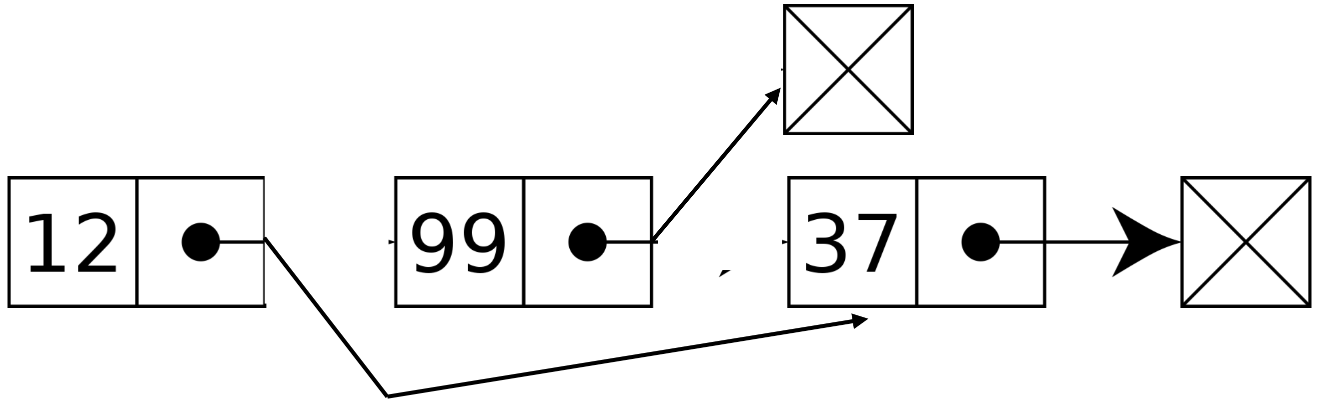
How can we remove it?

Removing an arbitrary element from a linked list

Let’s say we want to remove 99:



Two things have to happen: 12 needs to point 37. And have 99 point to null:

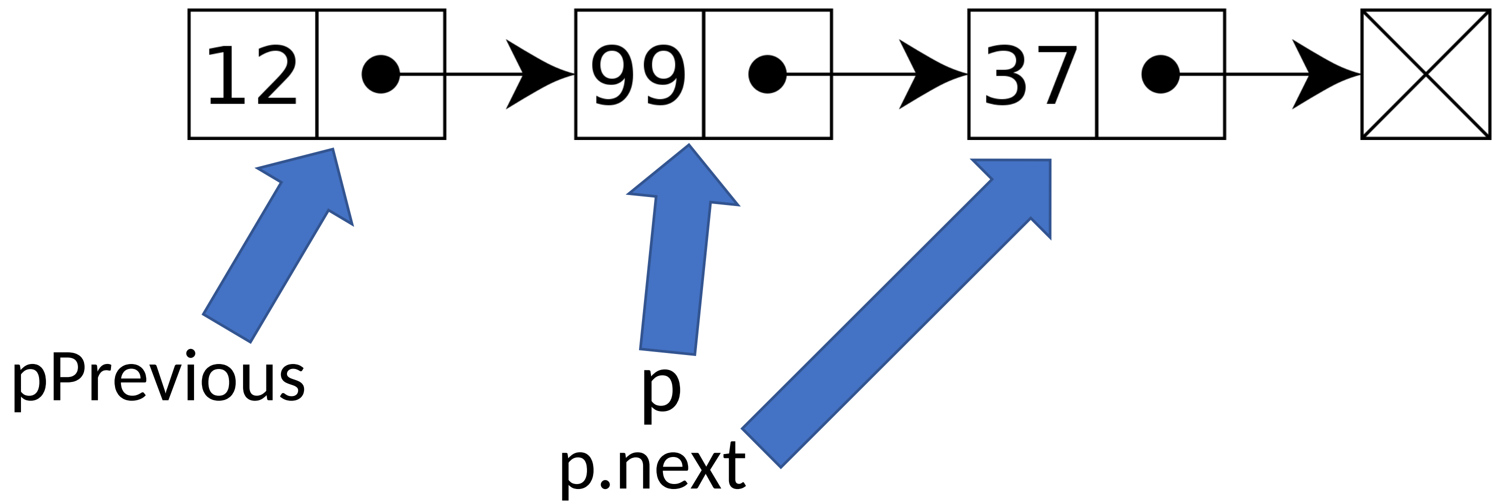


Let’s assume we are working with node 99. We can access 37 by using node.next
But how to go back and change 12?

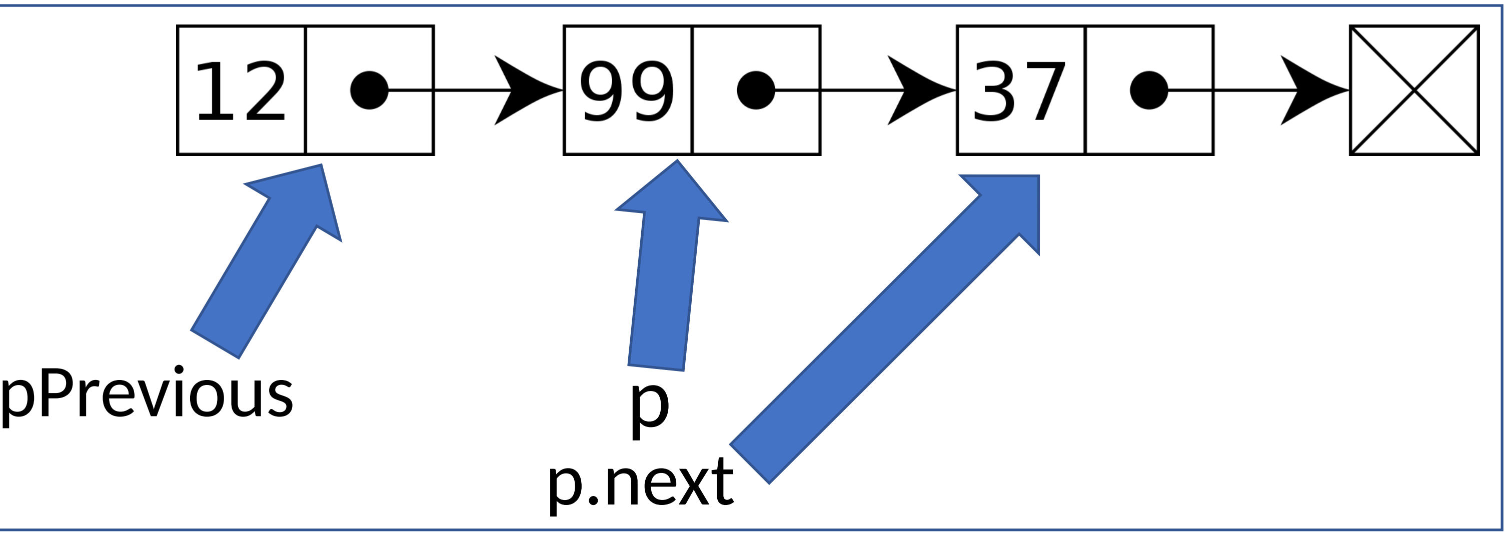
Removing an arbitrary element from a linked list

Let’s assume we are working with node 99. We can access 37 by using node.next
But how to go back and change 12?

Answer: We need to keep track with a pointer to the previous p.

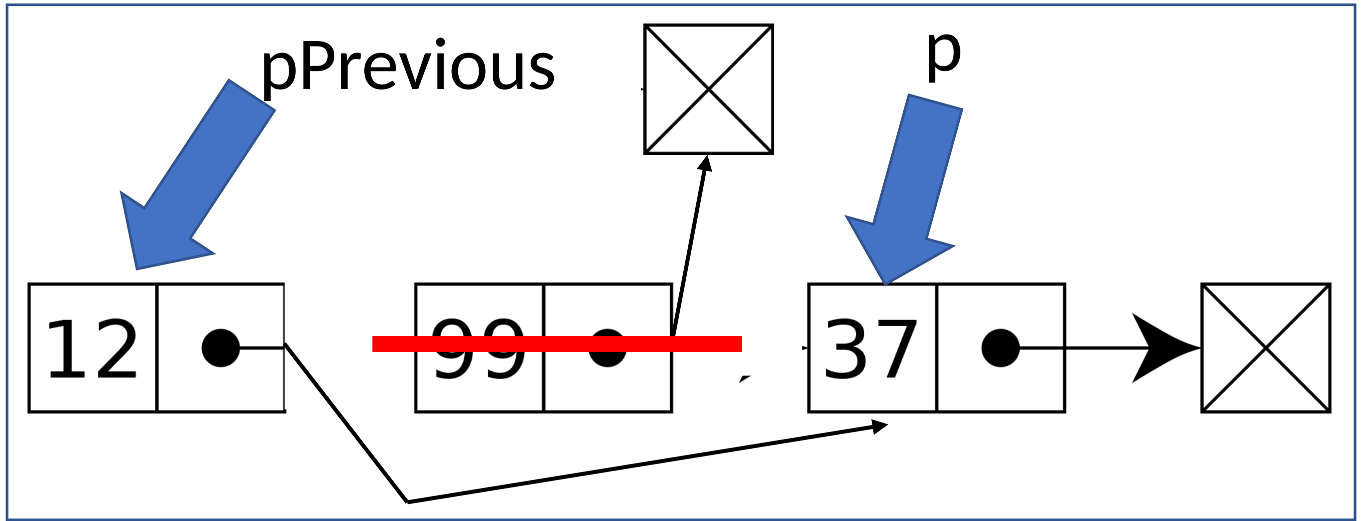


Removing an arbitrary element from a linked list

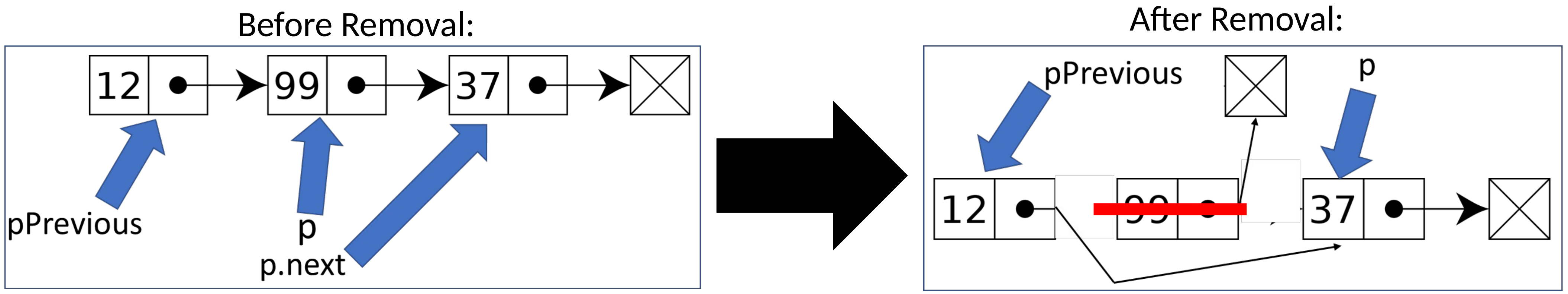


Before Removal.

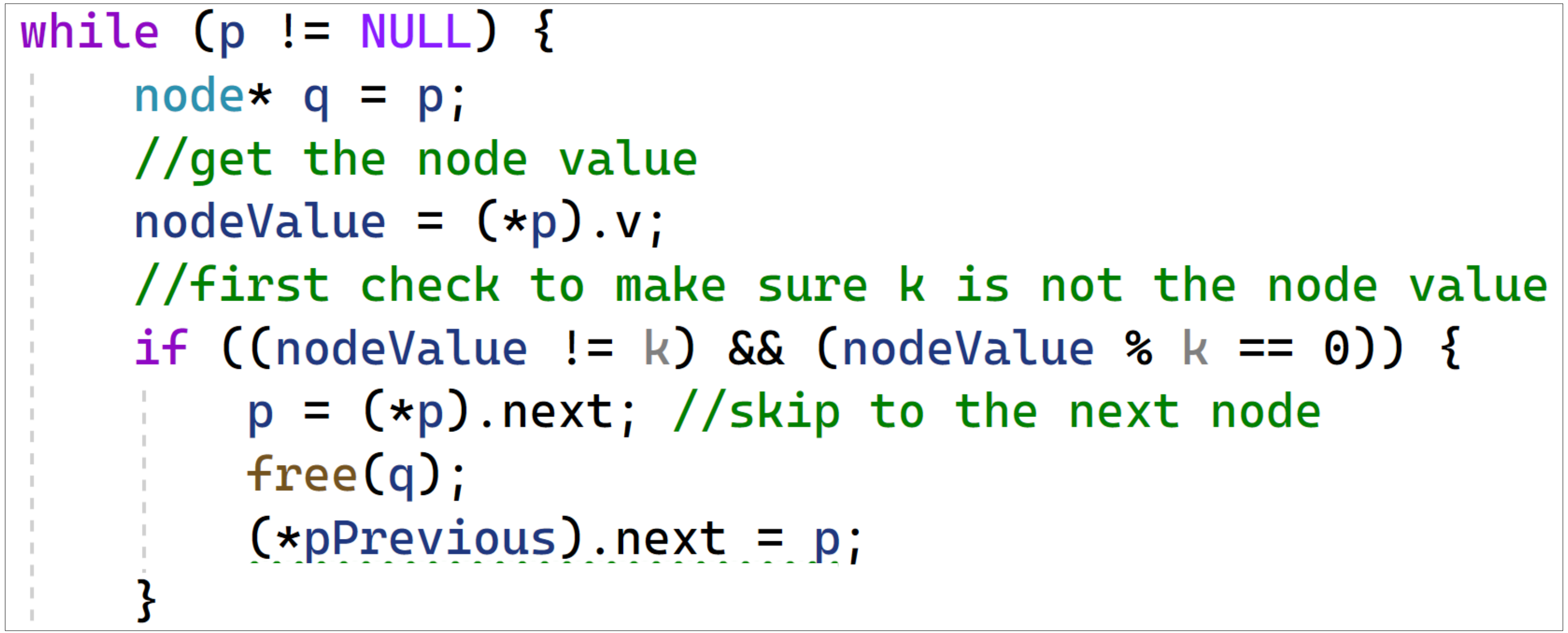
After Removal.



What does this look like in code?



```
while (p != NULL) {
    node* q = p;
    //get the node value
    nodeValue = (*p).v;
    //first check to make sure k is not the node value
    if ((nodeValue != k) && (nodeValue % k == 0)) {
        p = (*p).next; //skip to the next node
        free(q);
        (*pPrevious).next = p;
    }
```



```
while (p != NULL) {
    node* q = p;
    //get the node value
    nodeValue = (*p).v;
    //first check to make sure k is not the node value
    if ((nodeValue != k) && (nodeValue % k == 0)) {
        p = (*p).next; //skip to the next node
        free(q);
        (*pPrevious).next = p;
    }
```

- Consider the steps for the solution: remove all prime numbers from a linked list:

1. Iterate through the linked list.
2. IF the node is divisible by the input k AND it is not equal to k we should remove it.

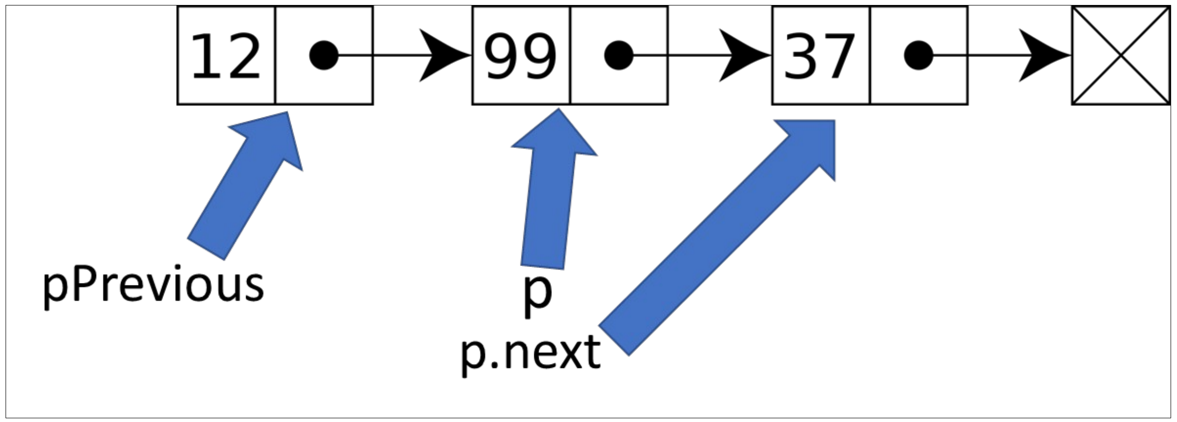
How can we remove it? Answered.

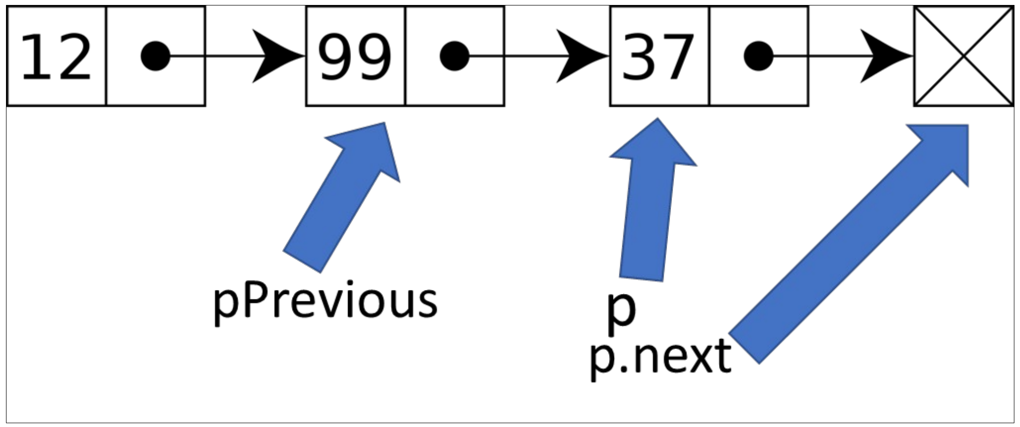
But what if we don’t want to remove it?

Iterating through a linked list (without removing)

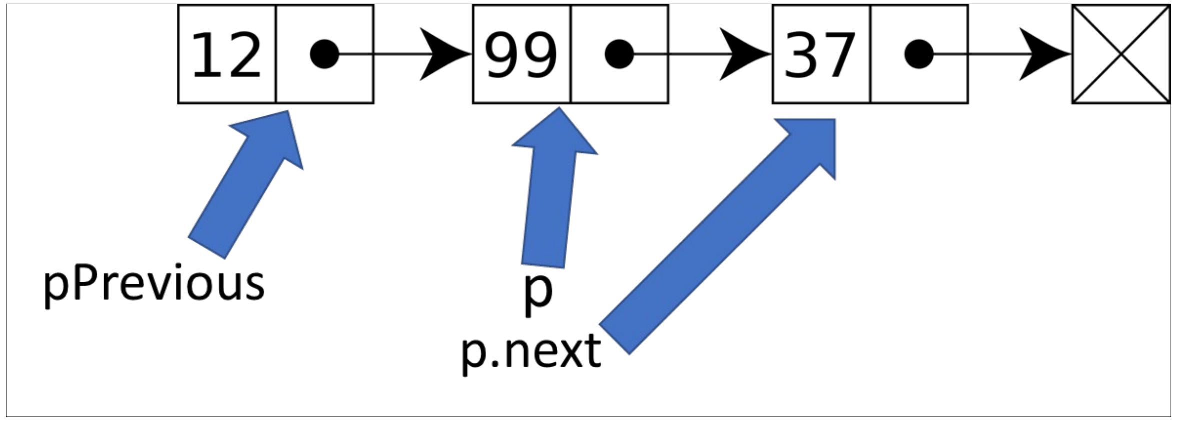
Let's assume we are working with node 99. How can we move forward?

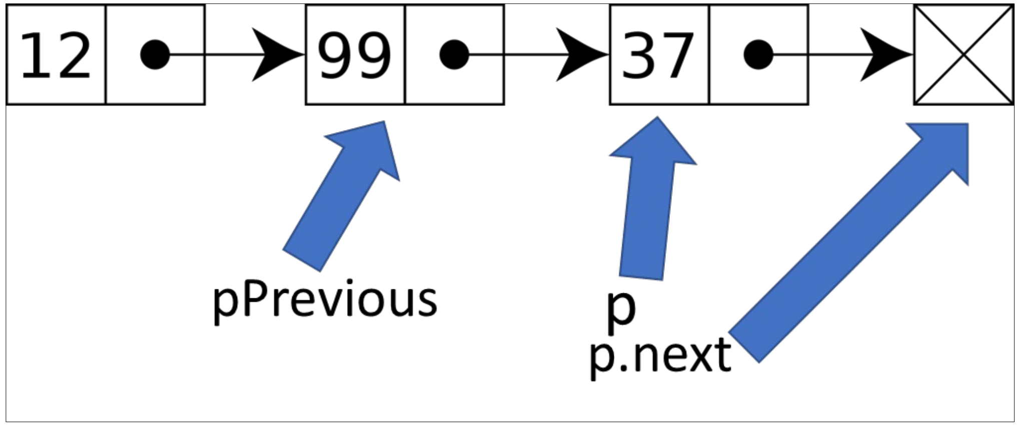
Answer: Just move the pointers!





Iterating through a linked list (without removing)





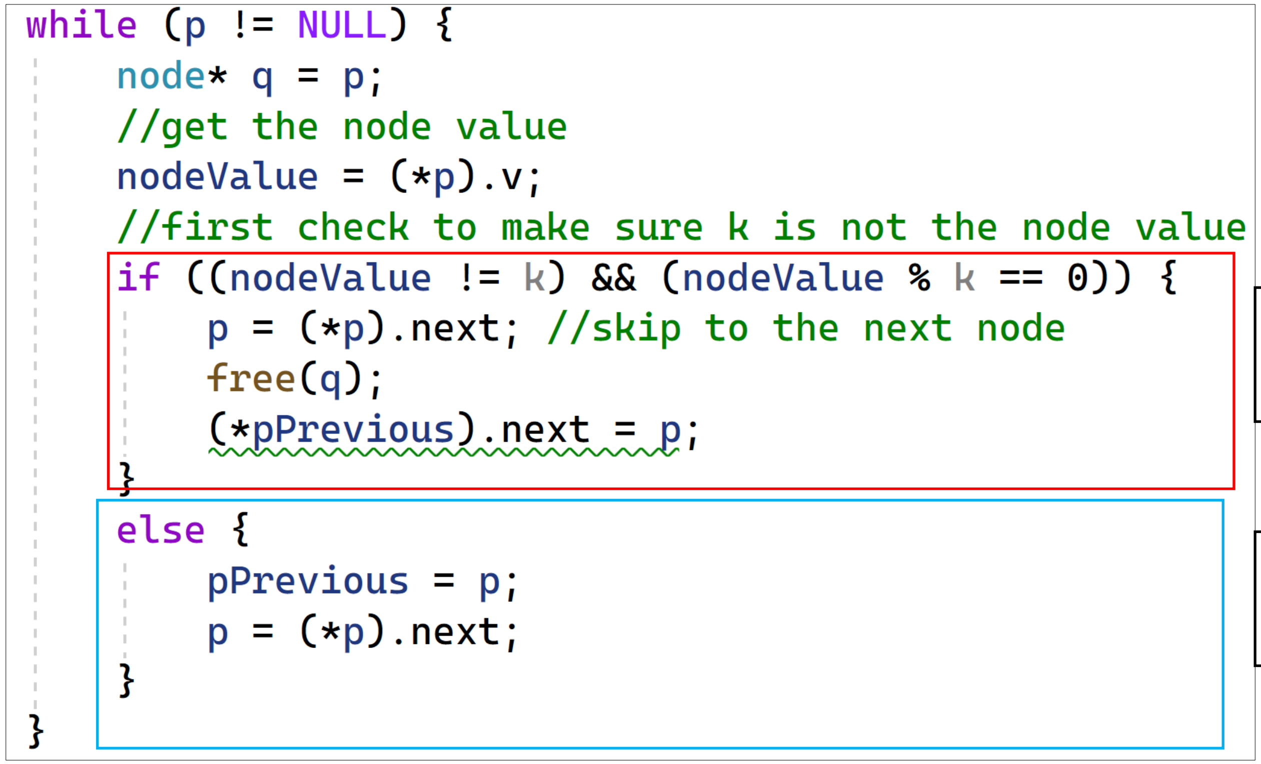
```
else {
    pPrevious = p;
    p = (*p).next;
}
```

- Consider the steps for the solution: remove all prime numbers from a linked list:

1. Iterate through the linked list.
2. IF the node is divisible by the input k AND it is not equal to k we should remove it.

How can we remove it? Answered.

But what if we don’t want to remove it? Answered.

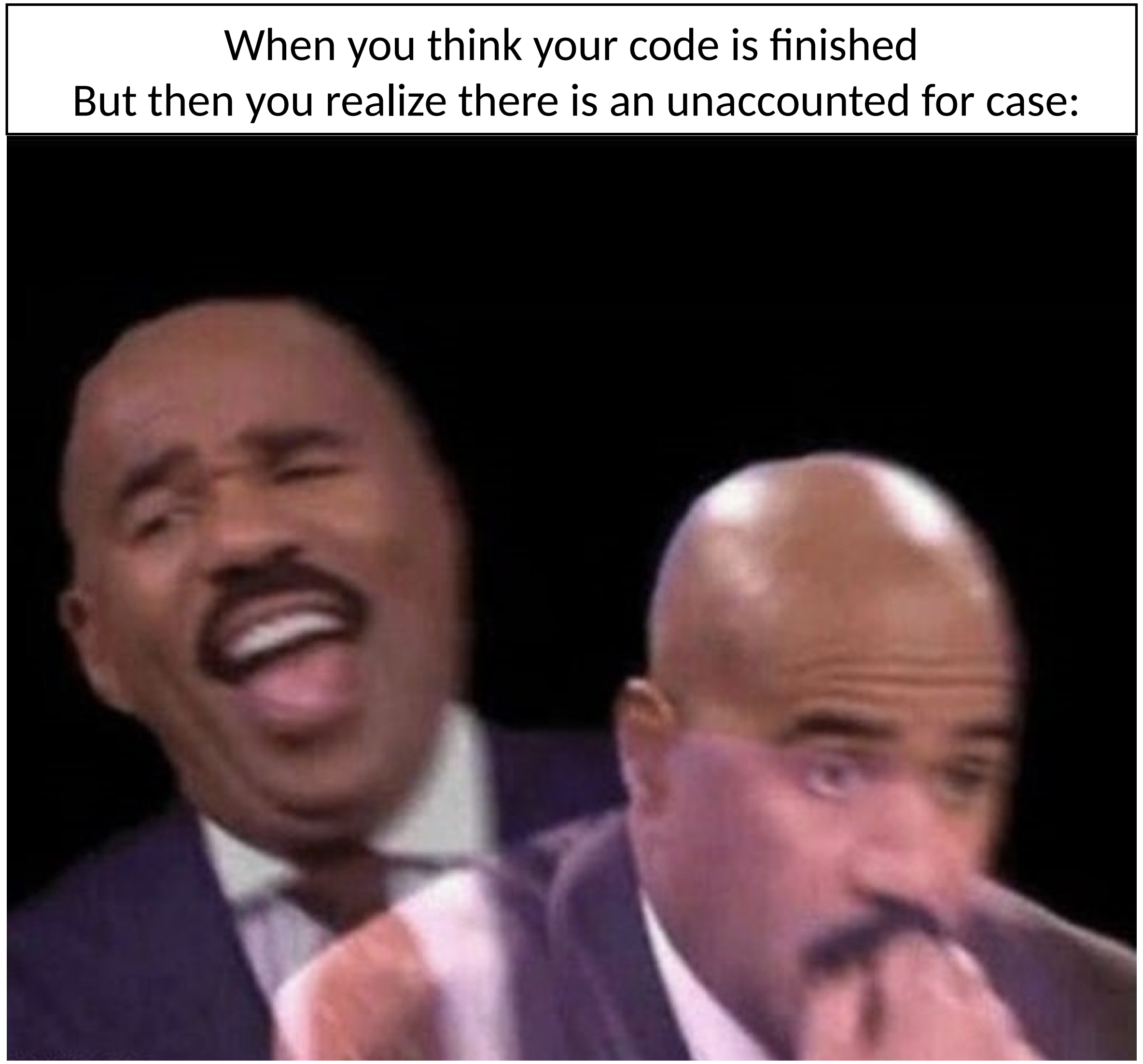


```
while (p != NULL) {
    node* q = p;
    //get the node value
    nodeValue = (*p).v;
    //first check to make sure k is not the node value
    if ((nodeValue != k) && (nodeValue % k == 0)) {
        p = (*p).next; //skip to the next node
        free(q);
        (*pPrevious).next = p;
    }
    else {
        pPrevious = p;
        p = (*p).next;
    }
}
```

Case when we need to remove a node.

Case when we simply move forward.

Have we completed the code?


When you think your code is finished
But then you realize there is an unaccounted for case:

What happens if the first node (the head) needs to be removed?

- Actually the code will fail.

- Case where the code works, when the starting list is:

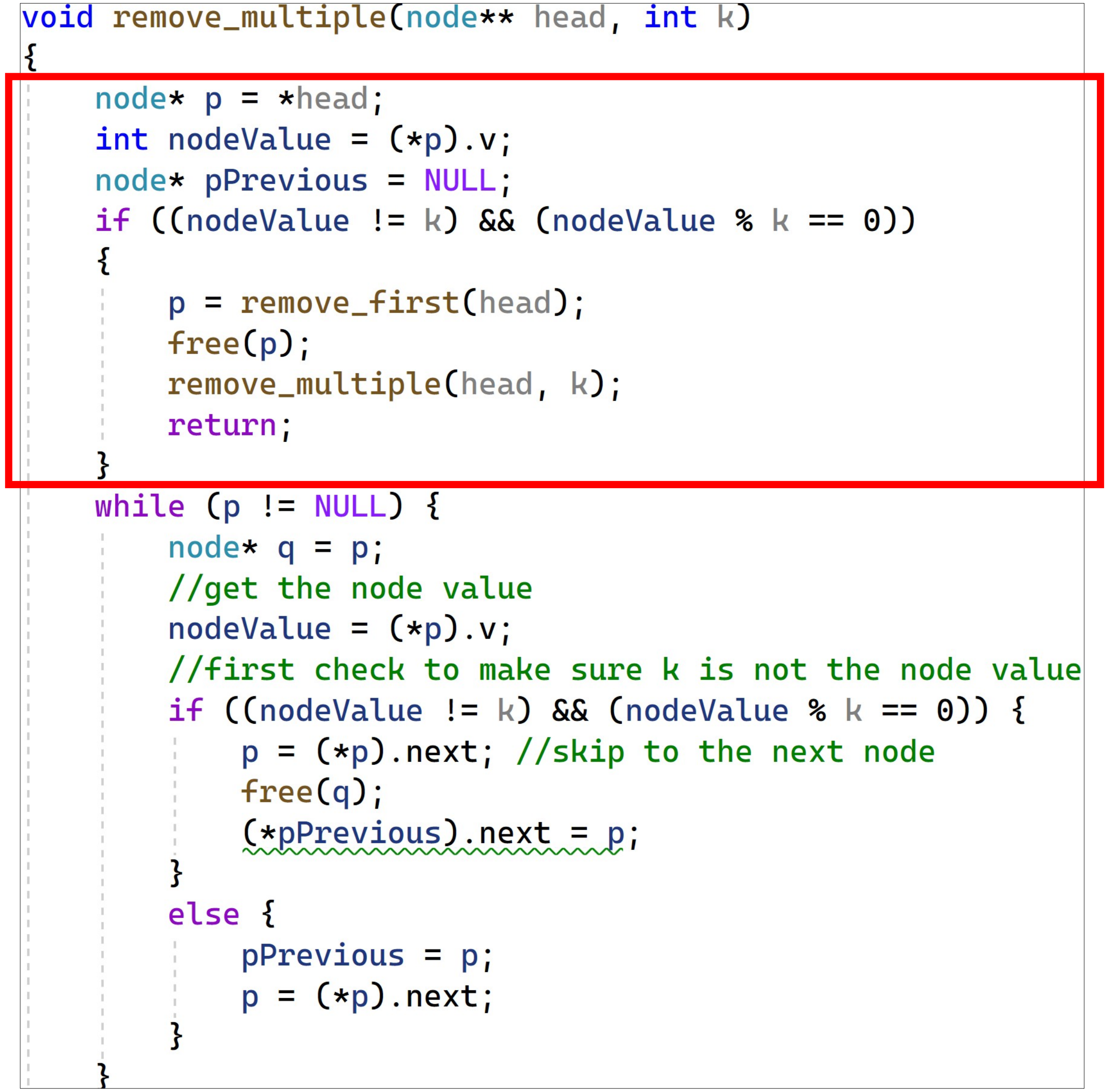
2,3,4,5,6,7,8,9

- Case where the code won’t work, when the starting list is:

9,8,7,6,5,4,3,2

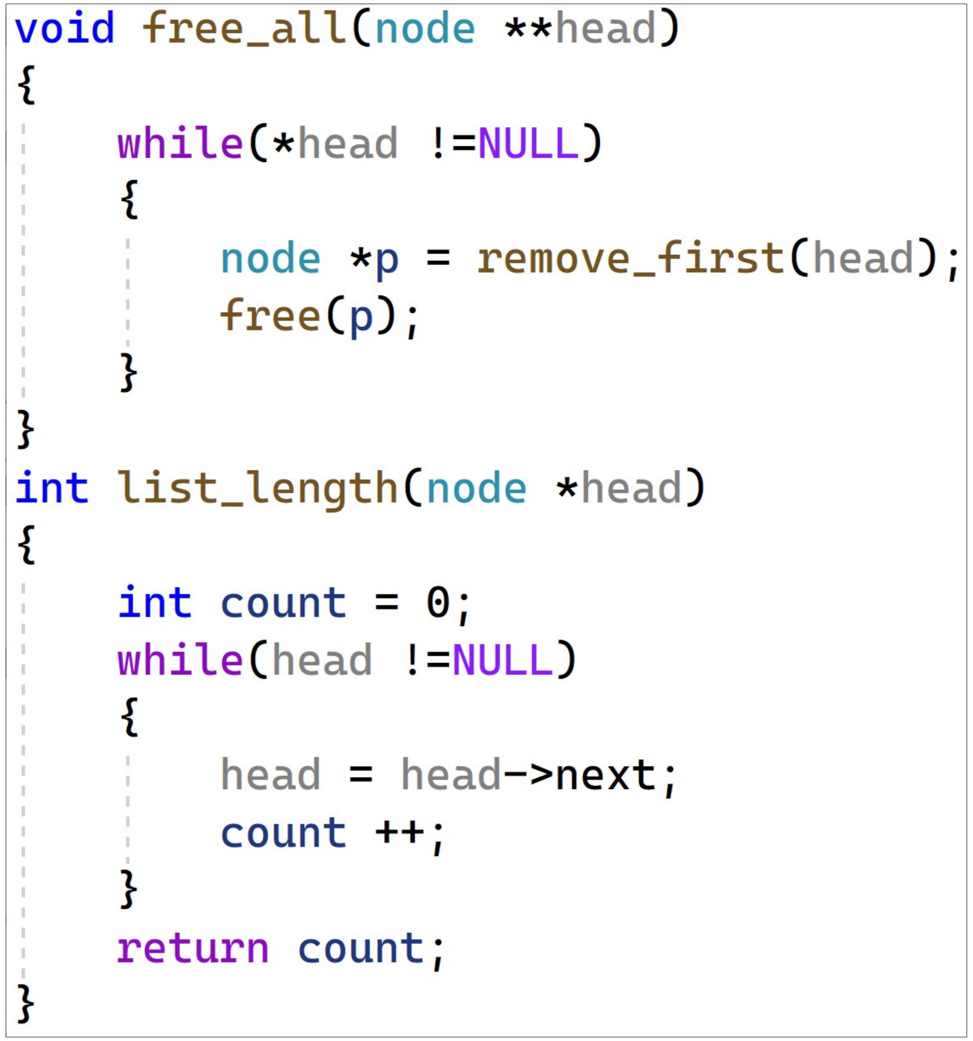
Full code

Check if the head needs to be removed. If so recall the method without the head.



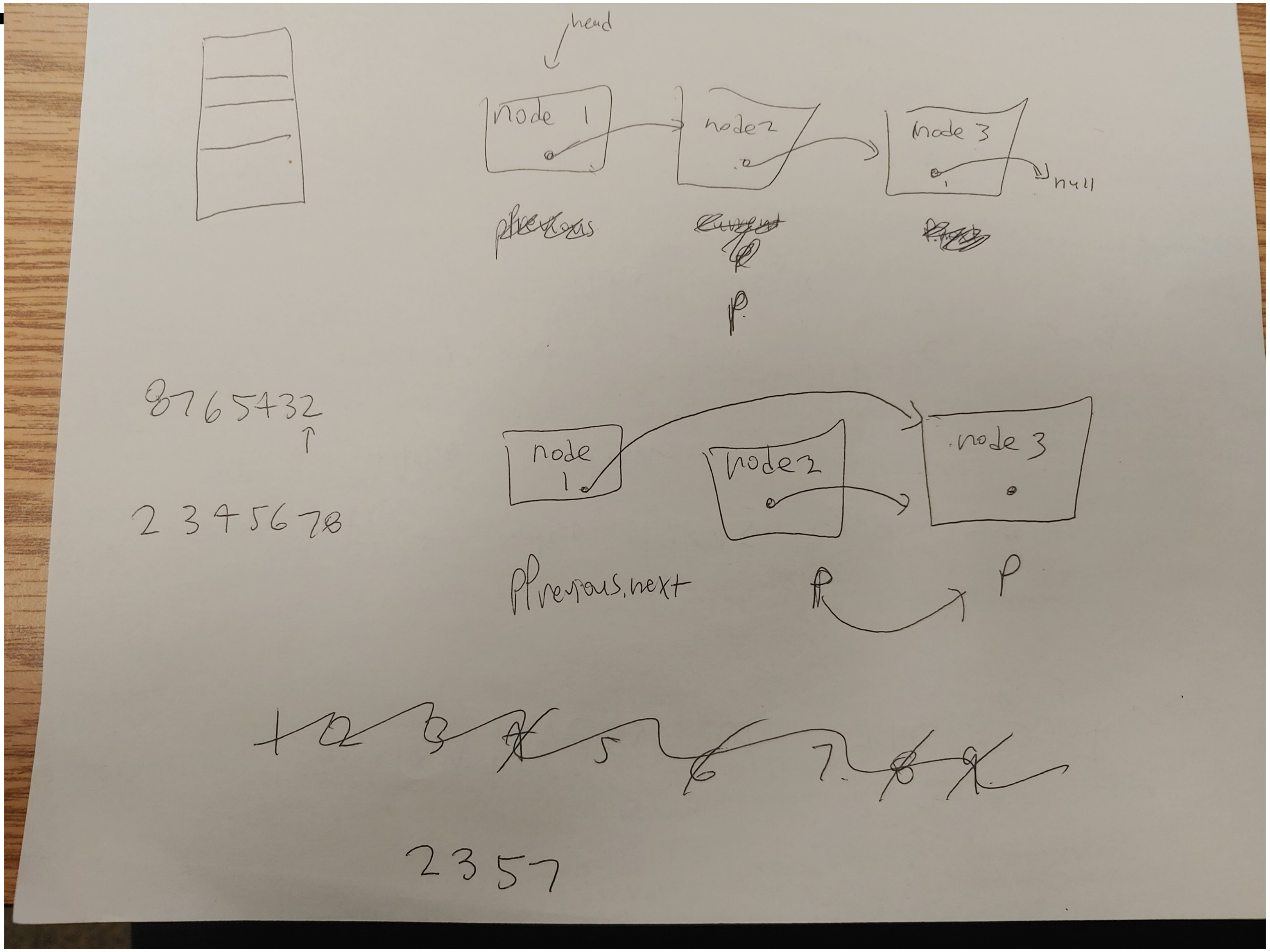
```c
void remove_multiple(node** head, int k)
{
    node* p = *head;
    int nodeValue = (*p).v;
    node* pPrevious = NULL;
    if ((nodeValue != k) && (nodeValue % k == 0))
    {
        p = remove_first(head);
        free(p);
        remove_multiple(head, k);
        return;
    }
    while (p != NULL) {
        node* q = p;
        //get the node value
        nodeValue = (*p).v;
        //first check to make sure k is not the node value
        if ((nodeValue != k) && (nodeValue % k == 0)) {
            p = (*p).next; //skip to the next node
            free(q);
            (*pPrevious).next = p;
        }
        else {
            pPrevious = p;
            p = (*p).next;
        }
    }
```

Other parts of Q3 solution code

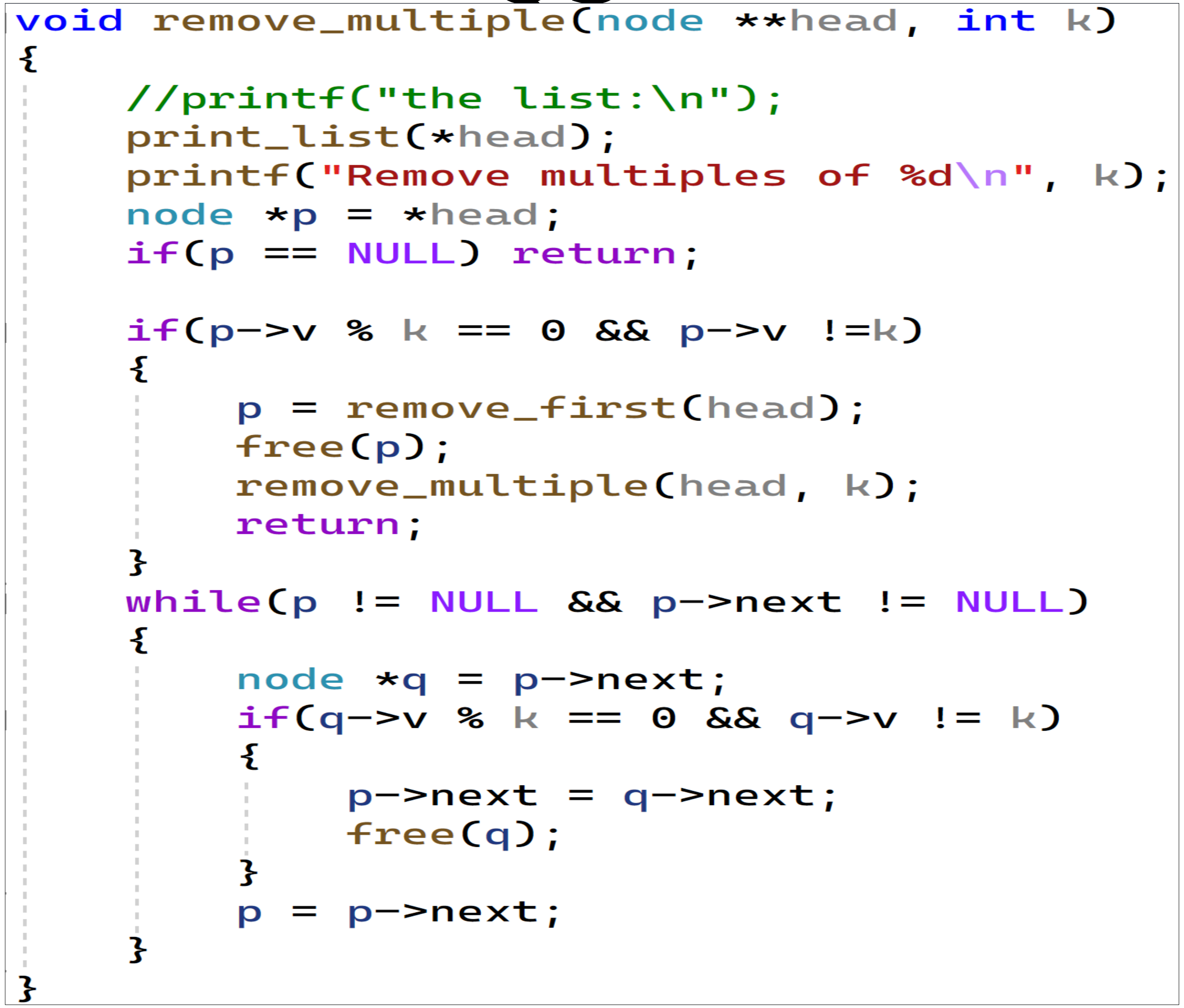


```c
void free_all(node **head)
{
    while(*head !=NULL)
    {
        node *p = remove_first(head);
        free(p);
    }
}
int list_length(node *head)
{
    int count = 0;
    while(head !=NULL)
    {
        head = head->next;
        count ++;
    }
    return count;
}
```

Not all starting solutions are beautiful.



Original Practice Exam Solution to Q3



```c
void remove_multiple(node **head, int k)
{
    //printf("the list:\n");
    print_list(*head);
    printf("Remove multiples of %d\n", k);
    node *p = *head;
    if(p == NULL) return;

    if(p->v % k == 0 && p->v !=k)
    {
        p = remove_first(head);
        free(p);
        remove_multiple(head, k);
        return;
    }
    while(p != NULL && p->next != NULL)
    {
        node *q = p->next;
        if(q->v % k == 0 && q->v != k)
        {
            p->next = q->next;
            free(q);
        }
        p = p->next;
    }
}
```

What should you focus on studying?

- How to write functions.
- How to manipulate structures.
- Be familiar with pointer operations.
- How to handle linked lists.
- How to solve coding problems in a reasonable amount of time.

Figure Sources

1. https://www.liveabout.com/thmb/0G67GhMHwZmedkiyOJ3hYdbP6Ug=/1500x0/filters:no_upscale():max_bytes(150000):strip_icc()/aliens1-5ad5062f3037130037462d1c.jpg
2. https://lh4.googleusercontent.com/_kCnLQGrMHXozZQqjGxWGvzFaNgr6SfWd5iI3qsIURnGuNC_CS1rUBvJrLIuZ4Bz-_l0-1JpWaZ8VSh5thbXY-0CP0d04ya_lWY6Pyzw8f2JxPK6B60S8fyc_fIgAp9TKd7BtREW
3. https://urbanmatter.com/phoenix/wp-content/uploads/2020/03/Laughing-Or-Coughing.jpg
4. https://i.imgflip.com/37ab1w.jpg
5. https://media.tenor.com/images/df614533e7fff38e27827c15853247dd/raw