1. head
   ┌───────┐
   │ 5     │
   │       │
   └───────┘
   tail

where
head = front
tail = rear

2. head
   ┌───────┐
   │ 1     │
   │       │
   └───────┘
   tail

3. head
   ┌───────┐
   │ 1     │
   │       │
   └───────┘
   tail

4. head
   ┌───────┐
   │ 1     │
   │       │
   └───────┘
   tail

5. head
   ┌───────┐
   │ 1     │
   │       │
   └───────┘
   tail

6. head
   ┌───────┐
   │ 1     │
   │       │
   └───────┘
   tail

7. head
   ┌───────┐
   │ 1     │
   │       │
   └───────┘
   tail

8. head
   ┌───────┐
   │ 1     │
   │       │
   └───────┘
   tail

9. head
   ┌───────┐
   │ 1     │
   │       │
   └───────┘
   tail

10. head
    ┌───────┐
    │ 1     │
    │       │
    └───────┘
    tail
4. (a) Store all data on blue, while using green for temp operations on the blue stack, the layout is like:

```
stack 1
stack 2
stack 3
```

We use three additional counters \( C[1], C[2], C[3] \) to store the number of elements in each stack.

```python
function operate_on_stack(num):
    for i in range(1, num-1):
        for t in range(C[i]):
            push 'pop from stack blue' to stack green
```

```python
function push_to_stack(num, element):
    operate_on_stack(num)
    push element to stack blue
    C[num] = C[num] + 1
    clear_green()
```

```python
function pop_of_stack(num):
    operate_on_stack(num)
    pop element from stack blue
    C[num] = C[num] - 1
    clear_green()
    return the popped element
```

```python
function clear_green():
    while stack green not empty:
        push 'pop from stack green' to stack blue
```

...
We use a similar scheme, but out \( C[i] \) on the top of each stack segment instead.

The operation on stack can be changed such that the elements to be popped is determined by popping \( C[i] \) first.

Push to stack can be changed to pop \( C[nun] \) first, do the pushing, and then push the updated \( C[nun] \) to the stack before clear.

Pop of stack can be done similarly to push to stack.

While \( up \neq NULL \) and \( up \neq NULL \)

\[
\text{if } up->index = up->index
\]

\[
res \leftarrow res + (up->value - up->value)^2
\]

\[
up \leftarrow (up->next), up \leftarrow (up->next)
\]

\[
\text{else if } up->index < up->index
\]

\[
res \leftarrow res + (up->value)^2
\]

\[
up \leftarrow (up->next)
\]

\[
\text{else}
\]

\[
res \leftarrow res + (up->value)^2
\]

\[
up \leftarrow (up->next)
\]

While \( up \neq NULL \)

\[
res \leftarrow res + (up->value)^2, up \leftarrow (up->next)
\]

While \( up \neq NULL \)

\[
res \leftarrow res + (up->value)^2, up \leftarrow (up->next)
\]

Return res
define \( \text{unvisited} \) being the \# of non-zero terms in \( u \) and \( v \) that have not been calculated for \( v \), we see the \( \text{unvisited} \) is either discounted by 1 or 2 in each iteration of (any of the) three "while"s.

Each iteration of while only has constant \# of primitive operations.

So the time complexity is

\[
O(\text{initial unvisited}) = O(\#nz in u and v)
\]

6.(a) Because \( 1 + |a_kn^k + a_{k-1}n^{k-1} + \ldots + a_0| = O(n^k) \)

\[\exists n_1, c_1 \text{ s.t.} \]

\[1 \leq |a_kn^k + \ldots + a_0| \leq c_1 \cdot n^k \text{ for } n \geq n_1,\]

that is

\[f(n) \leq \log c_1 + g_k(n) \text{ for } n \geq n_1,\]

because \( g_k(n) = \log n^k \geq 1 \) for \( n \geq 2 \)

\[\max(\log c_1, 1) g_k(n) \geq \max(\log c_1, 1) \text{ for } n \geq 2\]

Let \( n_0 = \max(n_1, 2) \)

\[f(n) \leq \log c_1 + g_k(n)\]

\[\leq \max(\log c_1, 1) + g_k(n)\]

\[\leq \max(\log c_1, 1) g_k(n) + g_k(n)\]

Let \( c = \max(\log c_1, 1) + 1 \geq 2 > 0 \)

we see that

\[f(n) \leq c g_k(n) \text{ for } n \geq n_0\]

that is, \( f(n) = O(g_k(n)) \)
(b) By $f(n) = O(g_k(n))$

$\exists n_0, c \ s.t.$

$f(n) \leq c \cdot (\log_2 n)^k$ for $n \geq n_0$

that is,

$f(n) \leq c \cdot k \cdot \log_2 n$ for $n \geq n_0$

Let $c' = c \cdot k$, $n_0' = n_0$

we see that

$f(n) \leq c' \cdot g(n)$ for $n \geq n_0'$

that is, $f(n) = O(g(n))$

7. disprove:

Let $f(n) = n$

g(n) = \frac{1}{2}n$

so $f(n) = O(g(n))$ by choosing $n_0 = 1$, $c = 2$

Let $h(n) = 4^n$ which is non-decreasing

then $h(f(n)) = 4^n$

$h(g(n)) = 2^n$

if $h(f(n)) = O(h(g(n)))$

$\exists n_0, c > 0$ s.t.

$4^n \leq c \cdot 2^n$ for all $n \geq n_0$

take log on both sides

$2n \leq \log_2 c + n$

$\Rightarrow n \leq \log_2 c$

that is, take $n' = \max(n_0, \lceil \log_2 (c + 1) \rceil)$

because $h' > \log_2 c$

$4^n > c \cdot 2^n$

because $n' \geq n_0$

$4^n \leq c \cdot 2^n$

so $h(f(n)) \neq O(h(g(n)))$. . . . . . . .
8. perform binary search, but changing the condition to be

\[
\text{if } \text{arr[mid]} = \text{NULL}
\]

search for k in \([\text{left}, \text{mid})\)

else

search for k in \([\text{mid}, \text{right})\)

binary search maintains the invariance

"to-be-searched within \([\text{left}, \text{right})\)"

we maintain the invariance

"k within \([\text{left}, \text{right})\)"

so correctness & time complexity
are both \(\mathcal{O}\) binary search

One small detail is to check whether

\[
\text{arr[0]} \text{ is NULL first.}
\]

If so, simply return 0

without going into the main algorithm

to maintain the invariance.