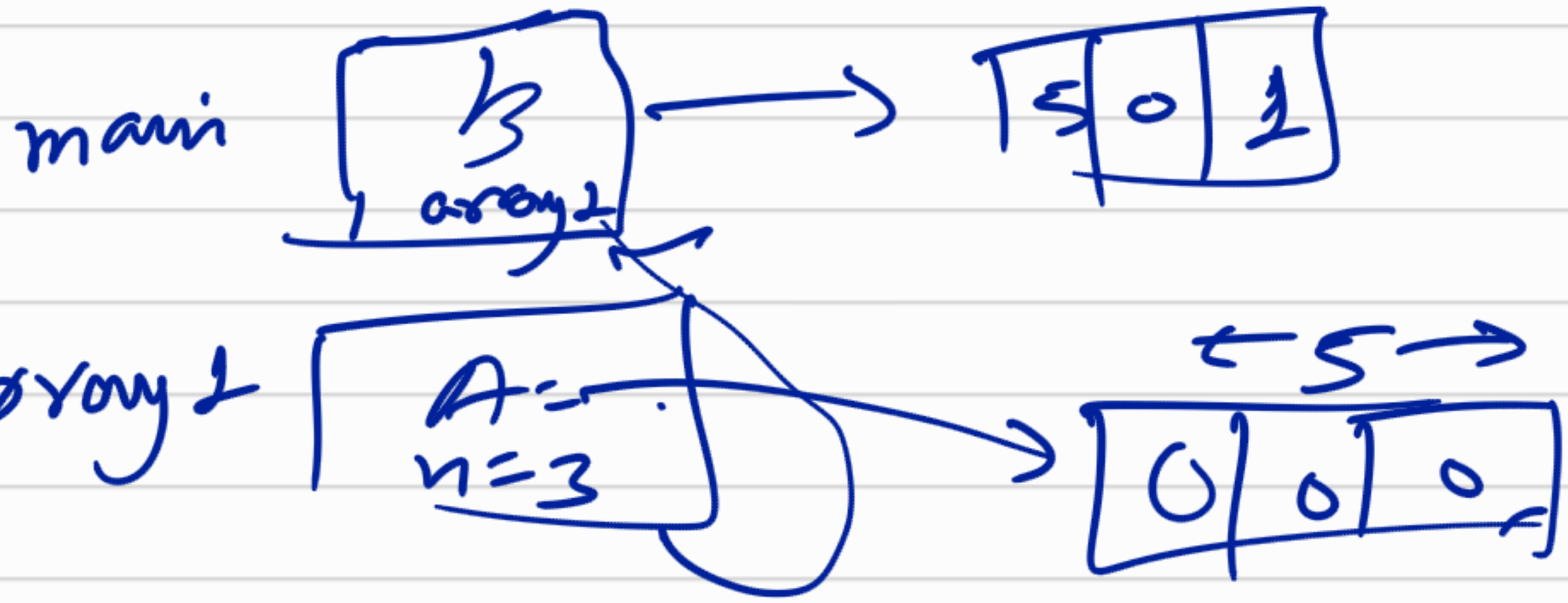
Arrays in C :

```
void array1(int[] A, int n){
→ A = alloc_array(int, 5);
}

int main(){
    int[] B = alloc_array(int, 3);
    B[0] = 5;
    B[2] = 1;
    array1(B, 3);
    // does B change ??
}
```

B → 0 | 2 → | 5 | 0 | 1 |



B does not change !!

void array2(int[] A, int n){

★ → A[0] = 2;

→ A = alloc_array(int, 5);

}

int main(){

int[] B = alloc_array(int, 3);

B[0] = 5;

B[2] = 1;

array1(B, 3);

// does B change??

A[0] = 2

→ no B has been

modified at ★ by

function array2.

// in summary: one "can" change array passed in a C0/C-function.

Discussion about representing "slices" of arrays

To iterate over an array B
length n,

```
for(int i=0; i<n; i++){
    // B[i] ....
}
```

Note that our loop guard is $i<n$.
The indices go from $0, 1, \ldots, n-1$
We represent such an interval
of nos. as $[0, n)$.

(A) In general for two integers
$lo \leq hi$ we use $[lo, hi)$ to
represent $\{lo, lo+1, \ldots, hi-1\}$

(B) when $lo = hi$ it is empty.

Prove the following (for $x, y, z \ldots$ ints)

① $[x, y) \cup [y, z) = [x, z)$.

② no. of elements in $[x, y) = y - x$.

return true if $x \in \{A[lo], \ldots, A[hi-1]\}$

$[lo, hi) \rightarrow$

$lo <= hi$

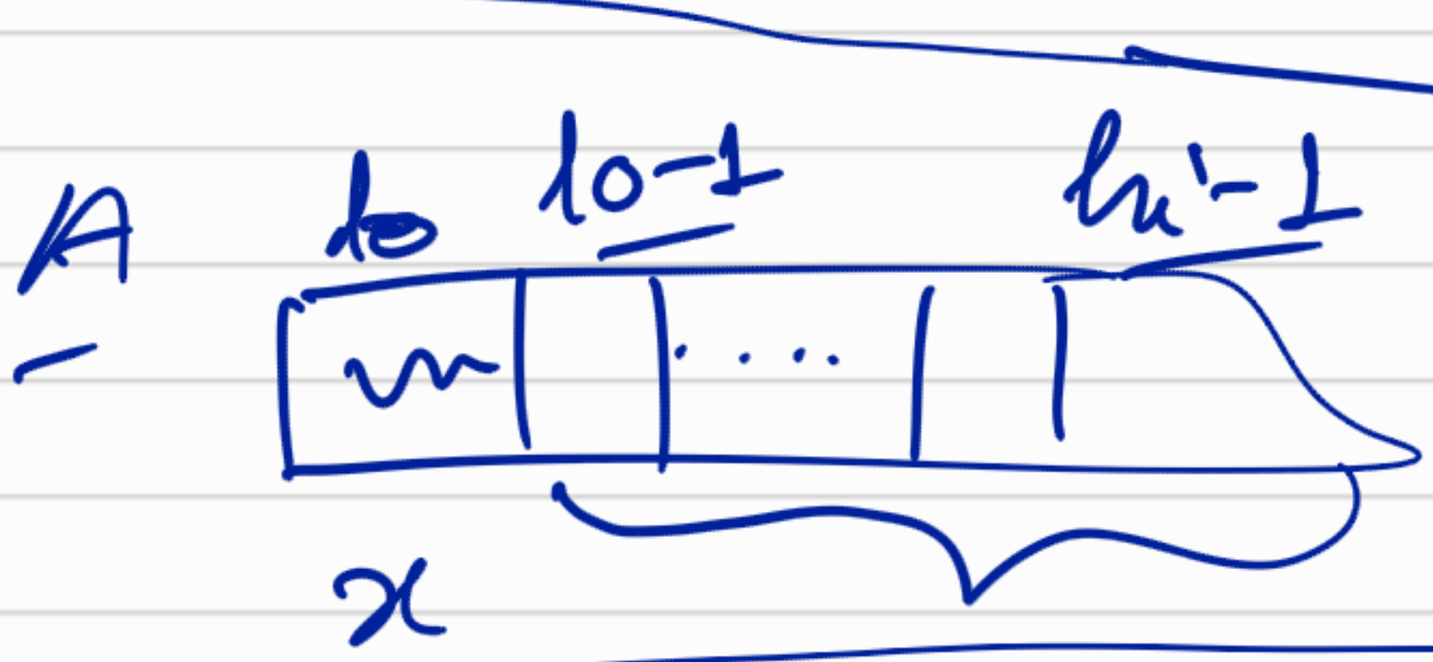
★ correction :-

// @requires 0<=lo && lo<=hi && hi<=\length(A);

```
1  bool is_in(int x, int[] A, int lo, int hi)
2  //@requires 0 <= lo <= hi <= \length(A);
3  {
4      if ( lo == hi ) {
5          return false;
6      }
7      else {
8          return A[lo]==x || is_in(x, A, lo+1, hi);
9      }
10 }
```

is_in(x, A, lo, hi)

$$= \begin{cases} \text{false} & \text{if } lo == hi \\ A[lo]==x \;||\; \text{is\_in}(x, A, lo+2, hi), & \text{o/w.} \end{cases}$$



✓

is_in(x, A, lo, hi) $\equiv$ $x \in A[lo, hi)$

Problem:- find if an element x is in $A[0,n)$ and return its index.

```
int search(int x, int[] A, int n)
{
    for(int i=0; i<n; i++) {
        if ( A[i] == x ) {
            return i;
        }
    }
}
```

adding the contracts we get

```
int search(int x, int[] A, int n)
//@requires n <= \length(A);
{
    for(int i=0; i<n; i++)
    //@loop_invariant 0 <= i && i <= n;
    {
        if ( A[i] == x ) {
            return i;
        }
    }
}
```

remove bug

when $x \notin A(0,n)$ return -1;

```
int search(int x, int[] A, int n)
//@requires n <= \length(A);
{
    for(int i=0; i<n; i++)
    //@loop_invariant 0 <= i && i <= n;
    {
        if ( A[i] == x ) {
            return i;
        }
    }
    return -1; //x not in A
}
```

~ int search (int x, int[] A, int n);

~ requires n == \length(A)

&

ensures
ans == -1 || A[ans]
== x;

int search (x, A, n) {

return -1;

}

≡

```
int search(int x, int[] A, int n)
//@requires n == \length(A);
//@ensures \result == -1 || A[\result] == x;
{
    for(int i=0; i<n; i++)
    //@loop_invariant 0 <= i && i <= n;
    {
        if ( A[i] == x ) {
            return i;
        }
    }
    return -1;
}
```

```
int search(int x, int[] A, int n)
//@requires n == \length(A);
//@ensures \result == -1 || A[\result] == x;
{
    for(int i=0; i<n; i++)
    //@loop_invariant 0 <= i && i <= n;
    {
        if ( A[i] == x ) {
            return i;
        }
    }
    return -1;
}
```

put requirements for array indexing

```
int search(int x, int[] A, int n)
//@requires n <= \length(A);
/*@ensures  \result == -1
            || ( 0 <= \result && \result < n && A[\result] == x );
@*/
{
    for(int i=0; i<n; i++)
    //@loop_invariant 0 <= i && i <= n;
    {
        if ( A[i] == x ) {
            return i;
        }
    }
    return -1;
}
```

add something here to capture the fact that $x \notin A[0, n)$

is_in(x, A, 0, n);

There are two places where the function returns (R1, R2)
Our correctness proof handles both these cases.

```
int search(int x, int[] A, int n)
//@requires n <= \length(A);
/*@ensures  ( \result == -1  && !is_in(x, A, 0, n) )
            || ( 0 <= \result && \result < n && A[\result] == x );
@*/
{
    for(int i=0; i<n; i++)
    //@loop_invariant 0 <= i && i <= n;
    {
        if ( A[i] == x ) {
            return i;
        }
    }
    return -1;
}
```

L1

R1

R2

Proving correctness of search.

Case I: Suppose search outputs at R1.

① $0 \le i$ by L1.

② $i < n$ by L7

③ $A[i] = x$ by R1

④ $\result = i$ by R1

①+②+④ $\Rightarrow 0 \le \result < n$

③+④ $\Rightarrow A[\result] = x$ ∎

Case II:

Suppose search returned at R2

```
int search(int x, int[] A, int n)
//@requires n <= \length(A);
/*@ensures  ( \result == -1  && !is_in(x, A, 0, n) )
            || ( 0 <= \result && \result < n && A[\result] == x );
@*/
{
    for(int i=0; i<n; i++)
    //@loop_invariant 0 <= i && i <= n;
    {
        if ( A[i] == x ) {
            return i;
        }
    }
    return -1;
}
```

$i \geq n$, $i \leq n$ $\Rightarrow i = n$ ← R2

$\equiv$

make sure that !is_in(x, A, 0, n)

$\equiv$ is_in(x, A, 0, n) is false

$x \notin A[0, n)$

$x \notin A[0, i)$

↳ looks like a good candidate for loop invariant-

If $x \notin A[0, i)$ is a loop invariant then $x \notin A[0, n)$ must be true when R2 is executed.

```
int search(int x, int[] A, int n)
//@requires n == \length(A);
/*@ensures  ( \result == -1  && !is_in(x, A, 0, n) )
            || ( 0 <= \result && \result < n && A[\result] == x );
@*/
{
    for(int i=0; i<n; i++)
    //@loop_invariant 0 <= i && i <= n;
    //@loop_invariant !is_in(x,A,0,i);
    {
        if ( A[i] == x ) {
            return i;
        }
    }
    return -1;
}
```

1 2 ← L1

3 4 5

R1

R2

Proving the loop invariant candidate

INIT: $x \notin A[0, i)$ before L1 is reached.

By L1 $i = 0$

$x \notin \underbrace{A[0, 0)}_{\text{No element here, so it does not have } x \text{ also.}}$ = true

PREServation: assume i' is the value of i after loop.

① assuming $x \notin A[0, i)$ we want to prove $x \notin A[0, i']$

② $i' \leftarrow i + 1$ by L1.

③ as we did not return at R1 the condition $A[i'] == x$ must be false.

$\Rightarrow A[i'] != x$. by ①

④ $A[i] \neq x$ & $x \notin A[0, i)$

④ $\Rightarrow x \notin A[0, i+1)$

$\Rightarrow x \notin A[0, i')$ using ②

hence proved the LI 2.

EXITing the loop:

at R2, we have $i = n$. } LG false, LI true

$\therefore x \notin A[0, i') = A[0, n)$

$x \notin A[0, n)$

$\therefore$ If the function returns at R2, it is correct.

$\therefore$ As both R1 & R2 return correctly, the correctness of search is proved. $\square$

Exercise:

Termination: for exiting the loop it is important to prove termination also. Show that the loop terminates.

Is the array sorted?

Ex: Prove correctness of is_sorted.

1)

```
1  bool is_sorted(int[] A, int n)
2  /*  checks if a non-empty array A is sorted in
3      ascending order*/
4  //@requires n==\length(A) && n>=1;
5  {
6      int i;
7      for(i=0; i<n-1; i++)
8      //@loop_invariant 0<=i && i<=n-1;
9      //prove the loop invariant that A[0,i) is sorted
10     {
11         if(!(A[i] <= A[i+1])
12             return false;
13     }
14     return true;
15 }
```

2) The code above also works for the case when A is empty. Modify your proof of 1 to prove the correctness of following.

```
1  bool is_sorted(int[] A, int n)
2  /* checks if the array A is sorted in ascending order*/
3  //@requires n==\length(A) && n>=0;
4  {
5      int i;
6      for(i=0; i<n-1; i++)
7      //@loop_invariant n==0 || (0<=i && i<=n-1);
8      //prove the loop invariant that A[0,i) is sorted
9      {
10         if(!(A[i] <= A[i+1])
11             return false;
12     }
13     return true; //returns true when n==0
14 }
15
```

contracts changed.

```
int search(int x, int[] A, int n)
//@requires n == \length(A);
/*@ensures  ( \result == -1  && !is_in(x, A, 0, n) )
            || ( 0 <= \result && \result < n && A[\result] == x );
@*/
{
    for(int i=0; i<n; i++)
    //@loop_invariant 0 <= i && i <= n;
    //@loop_invariant !is_in(x,A,0,i);
    {
        if ( A[i] == x ) {
            return i;
        }
    }
    return -1;
}
```

$O(n)$:

$n \to 2n$.

runtime doubles.

$O(n^2)$:

$n \to 2n$.

$n^2 \to (2n)^2$

$n^2 \to 4n^2$

runtime quadruples

Binary search :

$O(\log n)$

$n \to 2n$.

$\log n \to \log(2n)$

$= 1 + \log n$

$0 \le i < 2^{31} - 1 \simeq$ close to billion

31 steps.

Exercises :-

Debug & Prove correctness of the following by adding contract:

1:
```
int arraysum(int [] A, int n)
{
int sum = 0 ;
for (int i=0; i<n ; i++) {
    sum = sum + A[i] ;
    }
    return sum ;
}
```

2:
```
int arraymax(int [] A, int n) {
    int max = int_max ();
for (int i=0; i <n ; i++){
    if (max < A[i])
        max = A[i];
}
return max ; }
```