COMP 211
Fall 2014

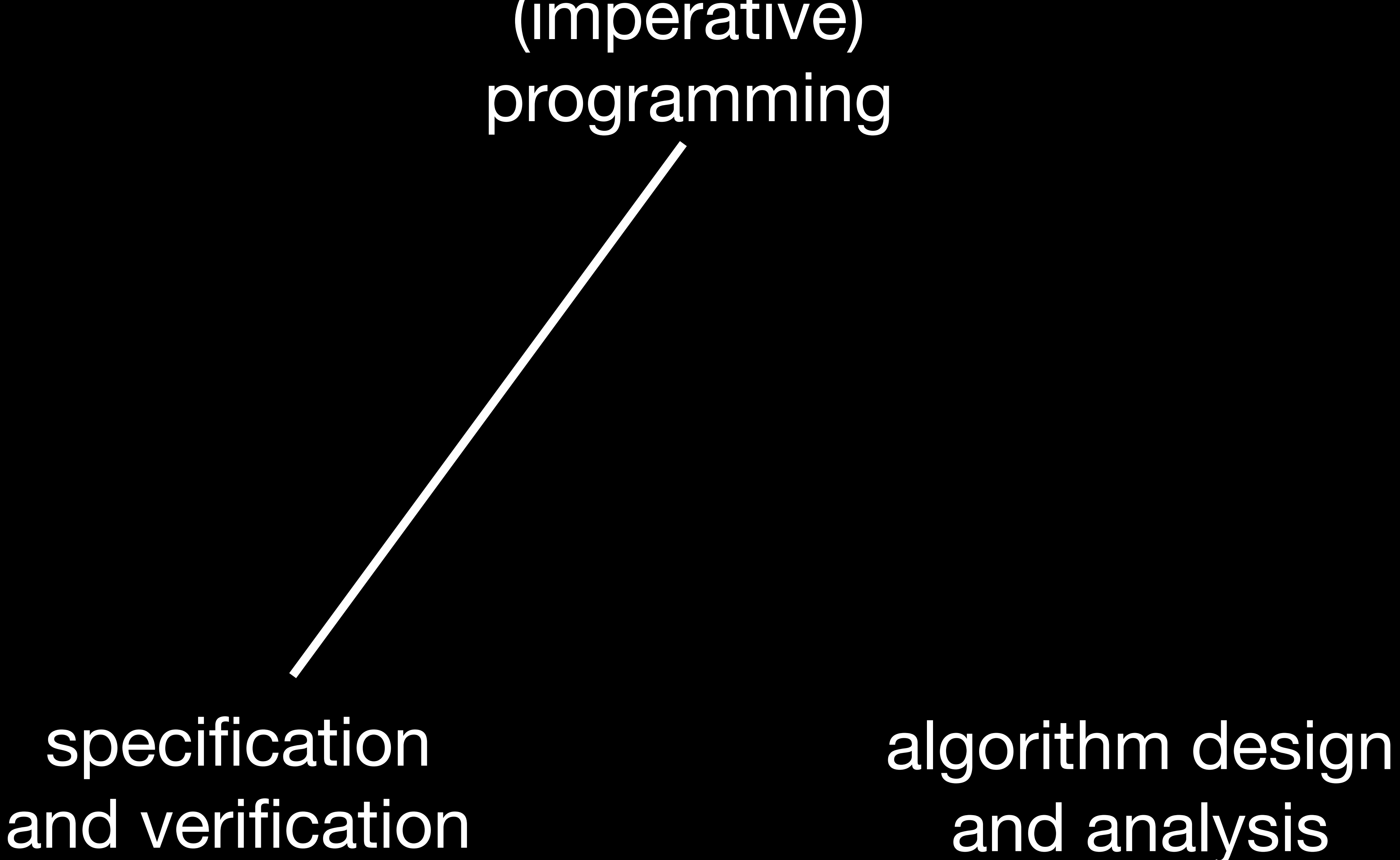
Objectives

(imperative)
programming

specification
and verification

algorithm design
and analysis

Objectives

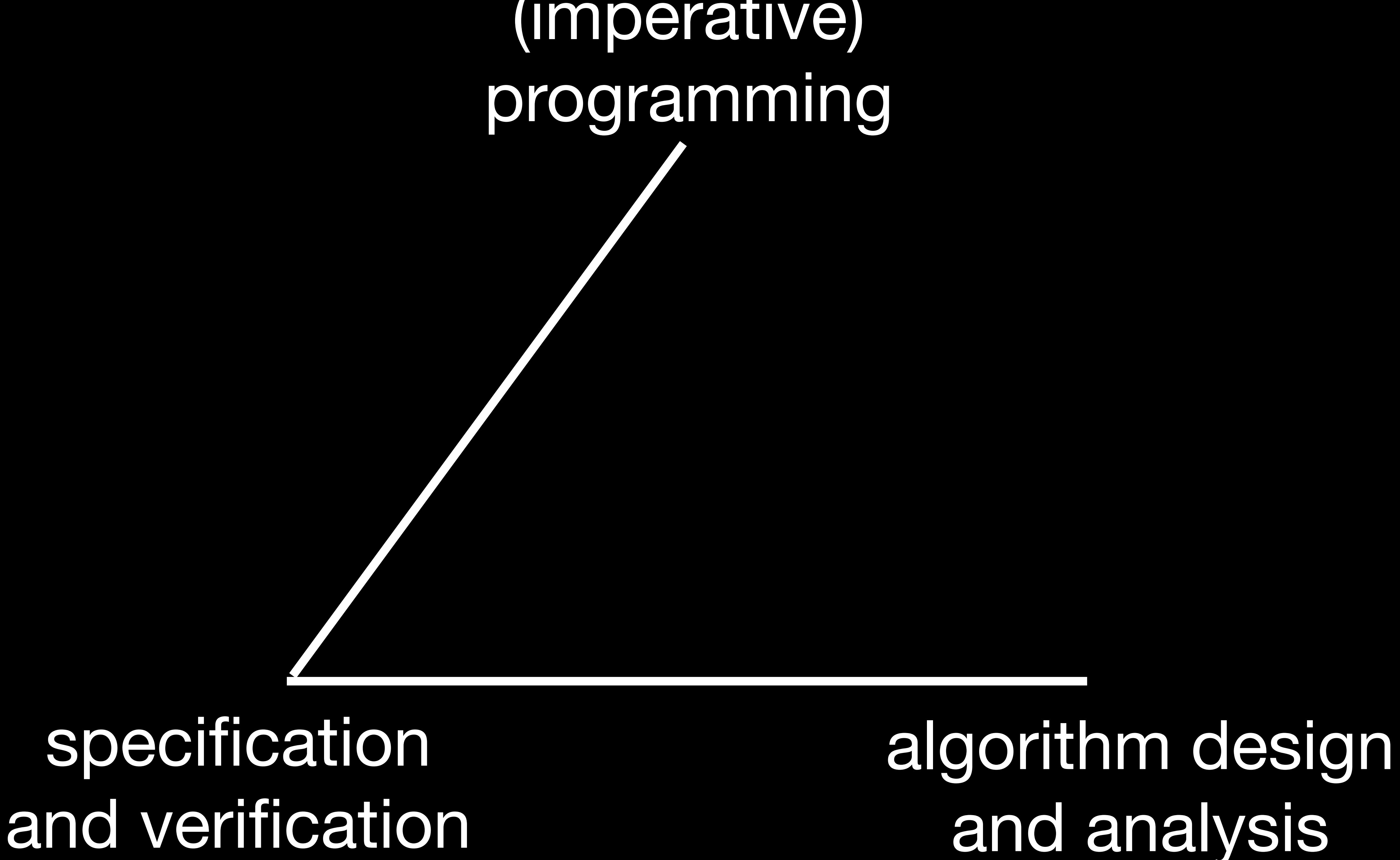


(imperative)
programming

specification
and verification

algorithm design
and analysis

Objectives

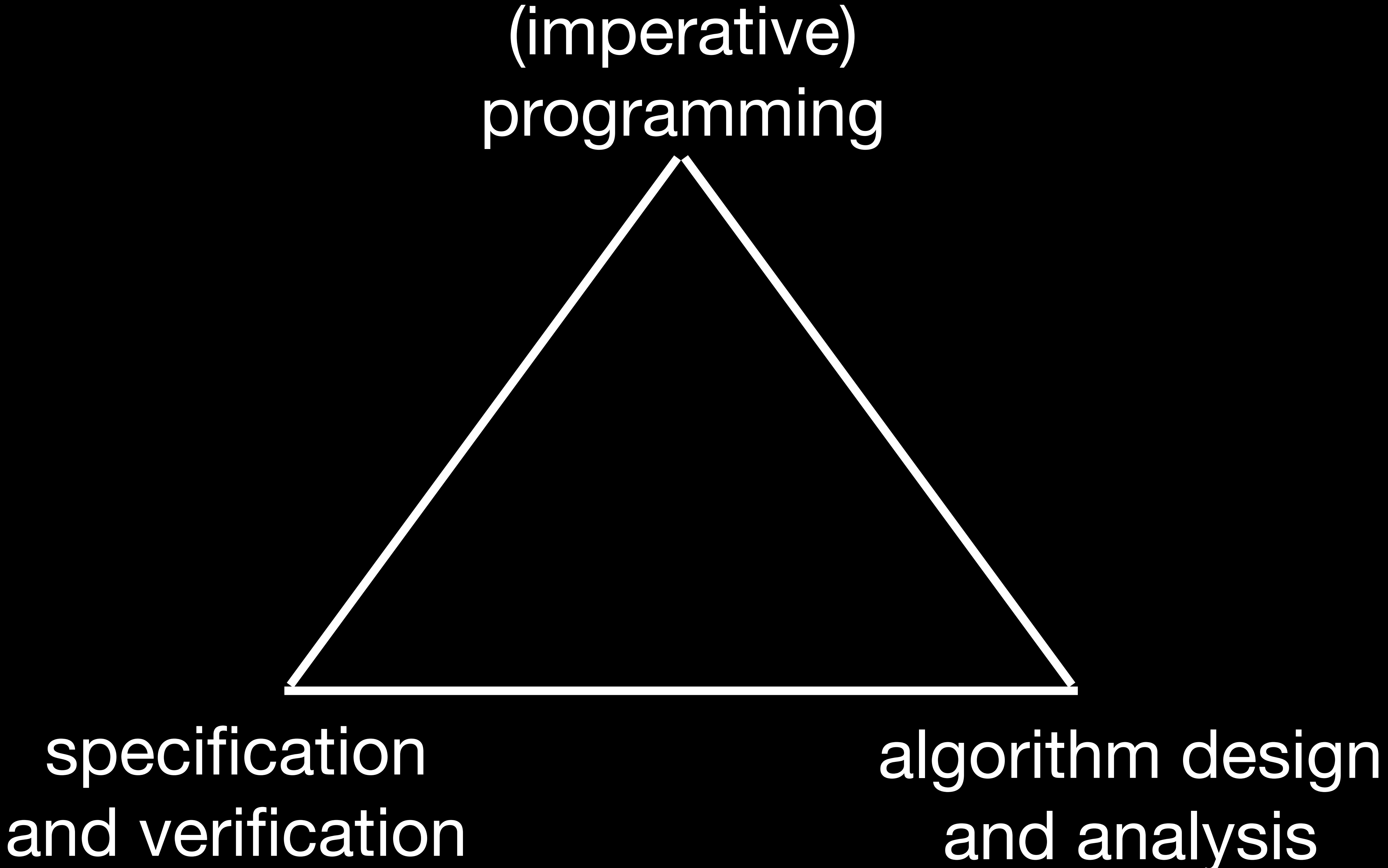


(imperative)
programming

specification
and verification

algorithm design
and analysis

Objectives



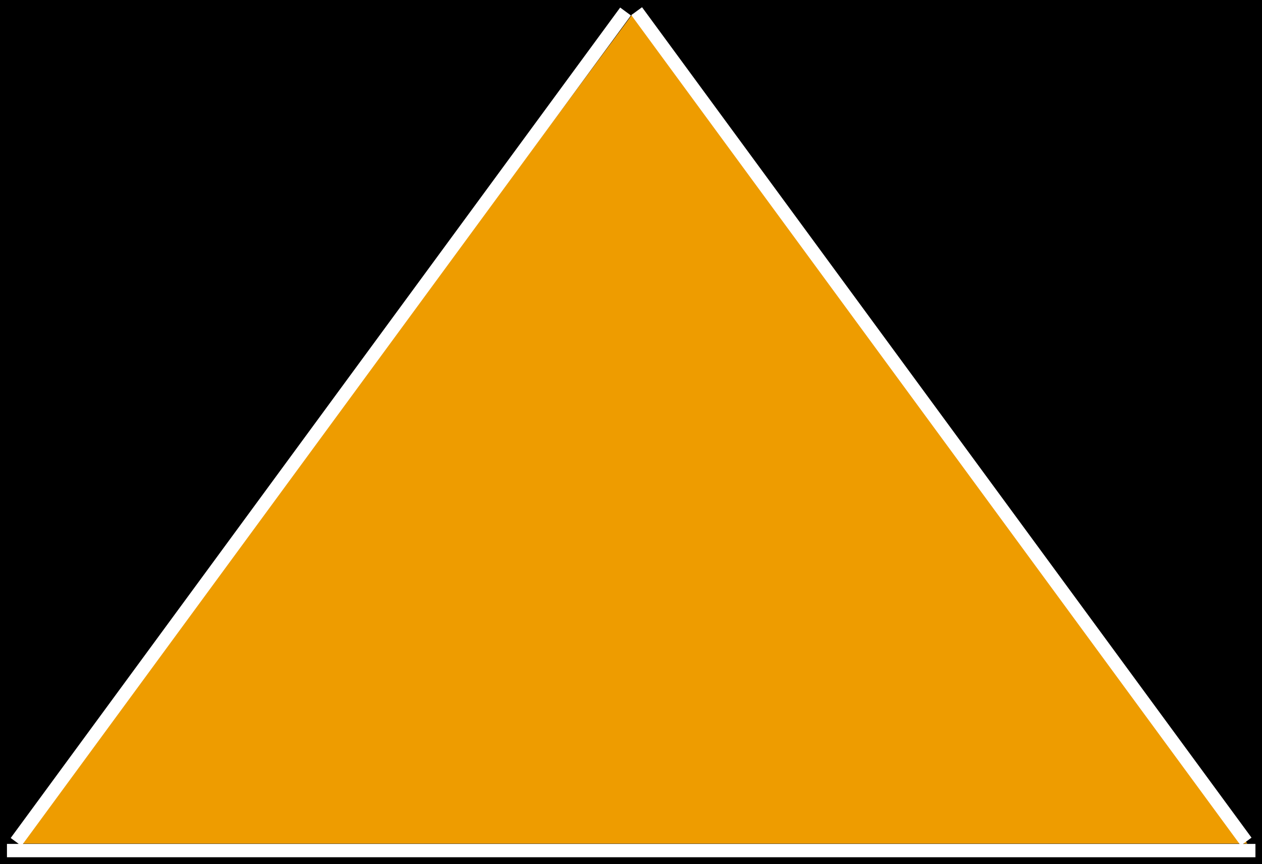
(imperative) programming

specification and verification

algorithm design and analysis

Objectives

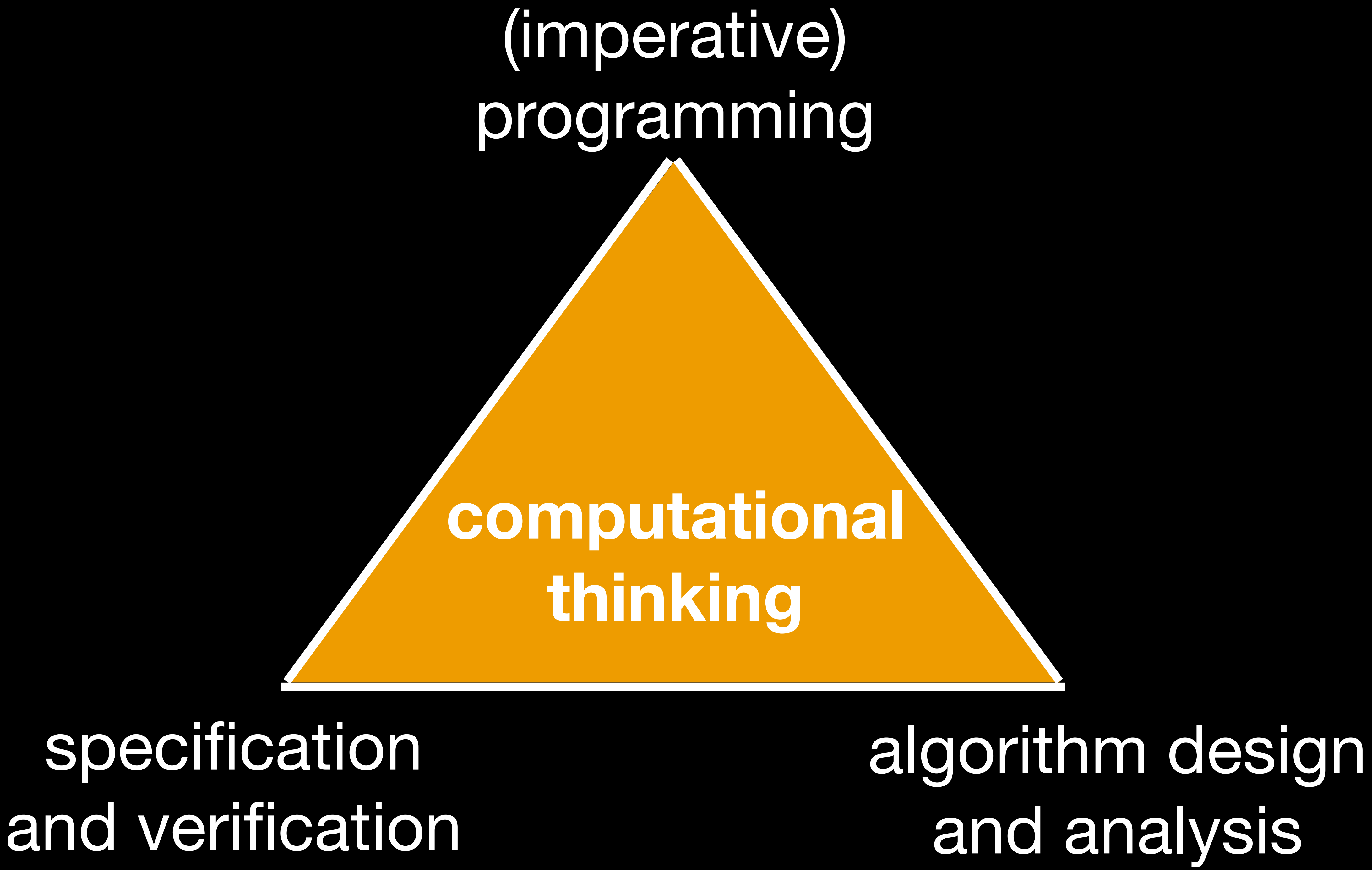
(imperative) programming



specification and verification

algorithm design and analysis

Objectives



(imperative)
programming

computational
thinking

specification
and verification

algorithm design
and analysis

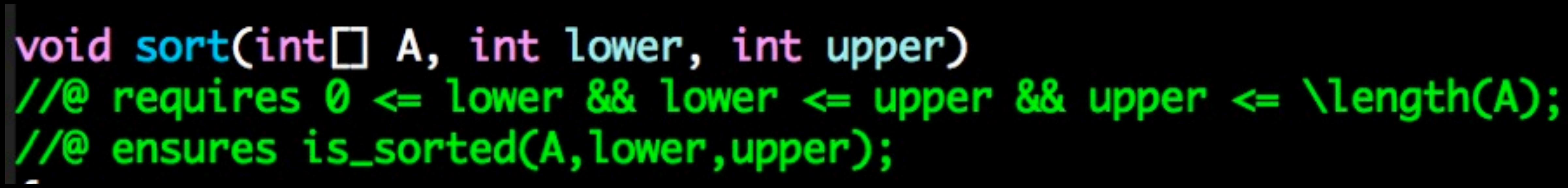
What vs How

requires
ensures
loop invariants
interfaces
big-O

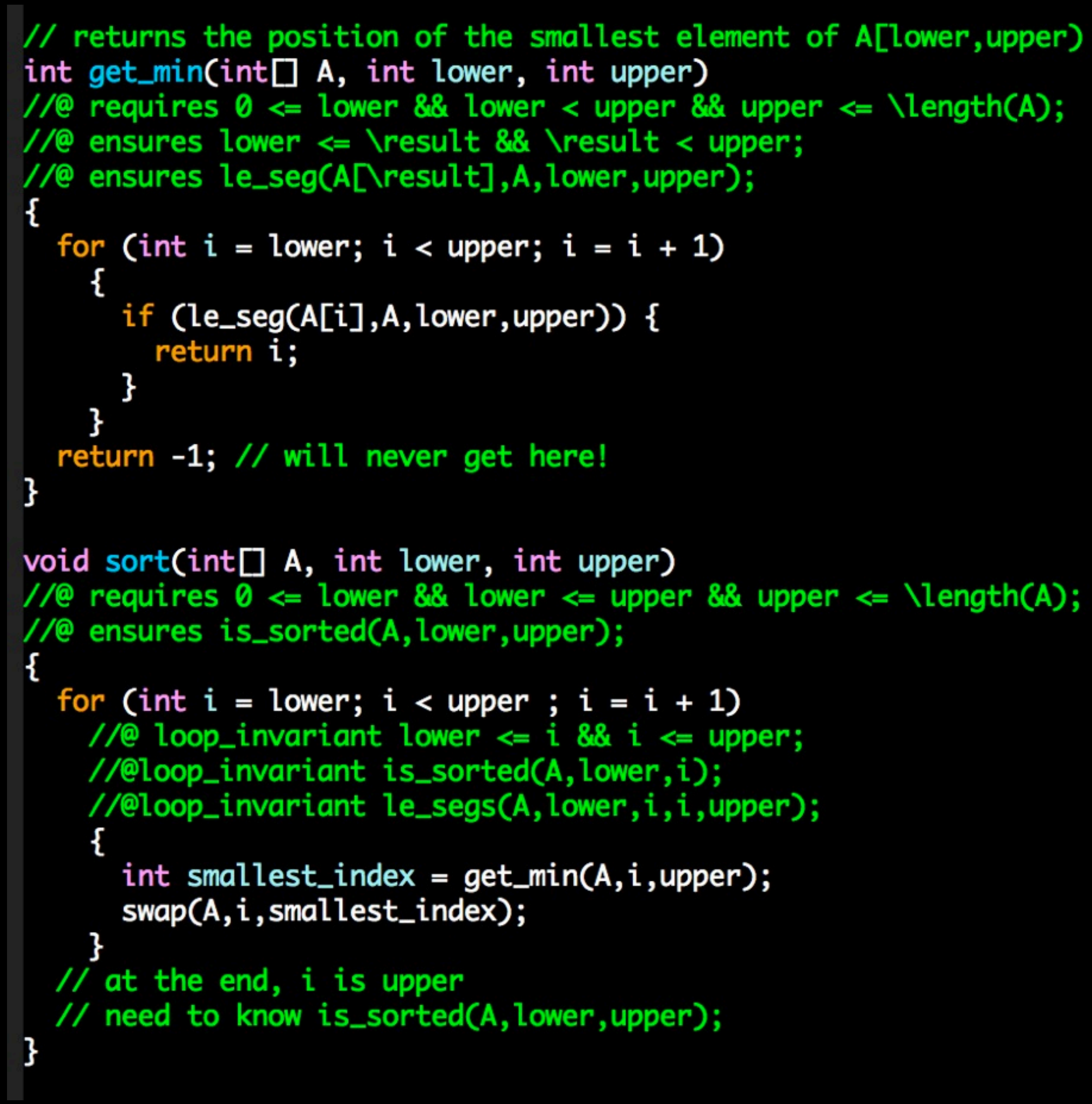
algorithm
code

many hows for the same what

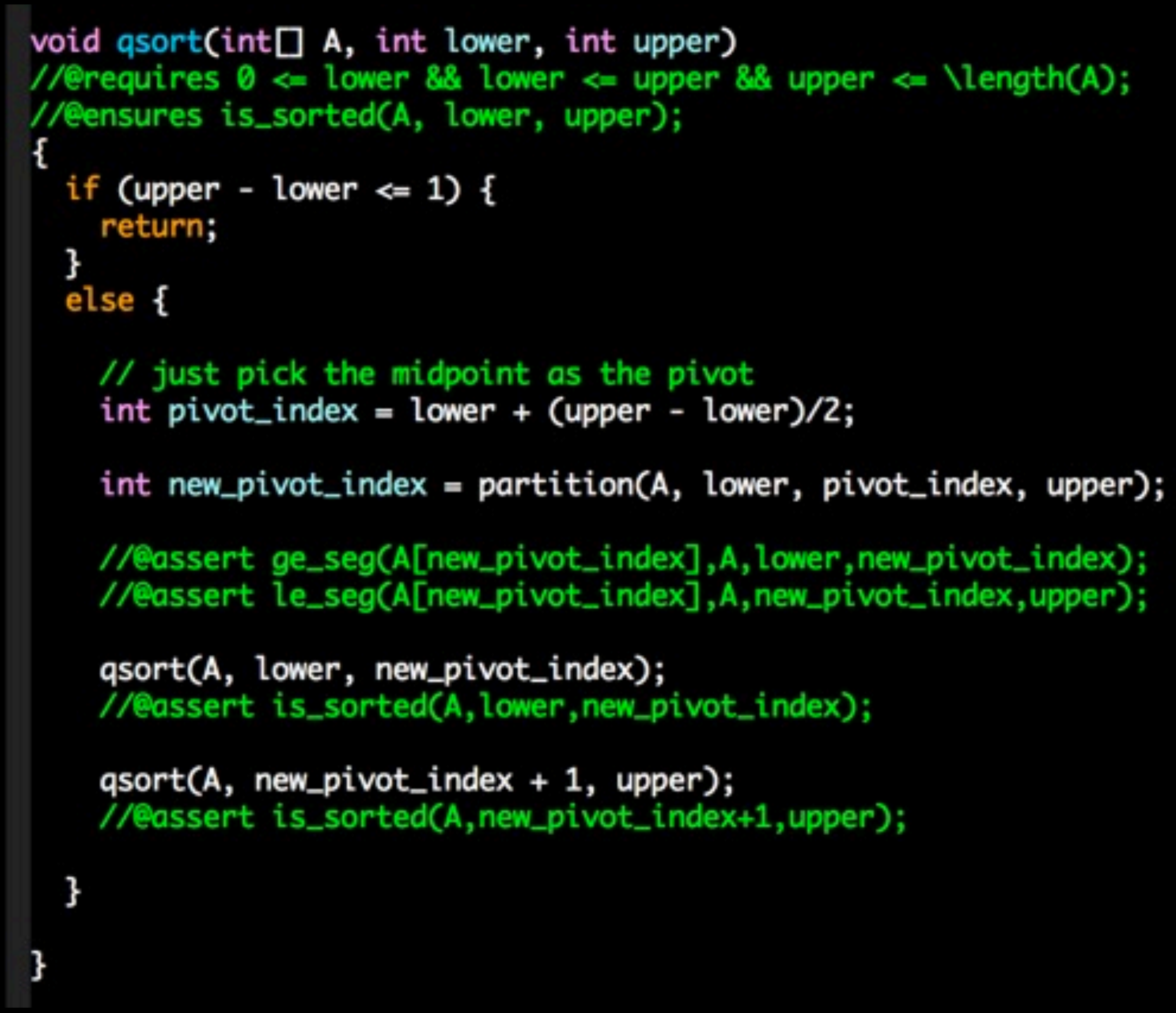
proof-oriented programming



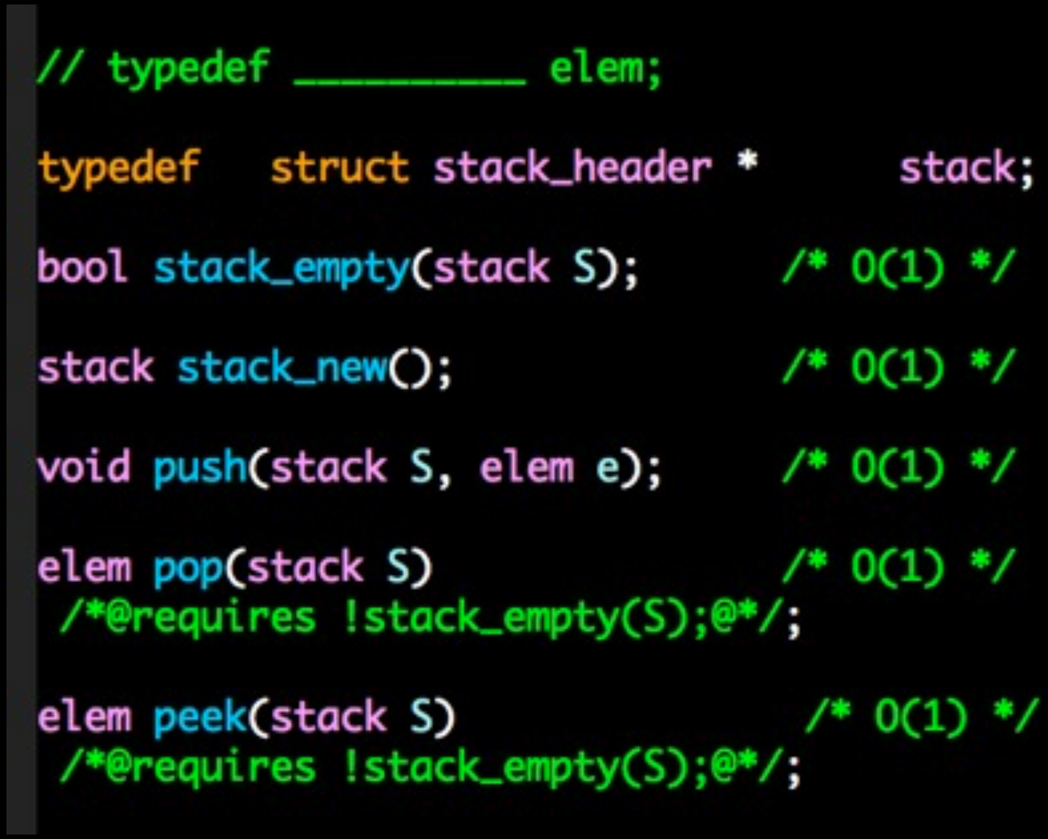
```
void sort(int[] A, int lower, int upper)
//@ requires 0 <= lower && lower <= upper && upper <= \length(A);
//@ ensures is_sorted(A,lower,upper);
```



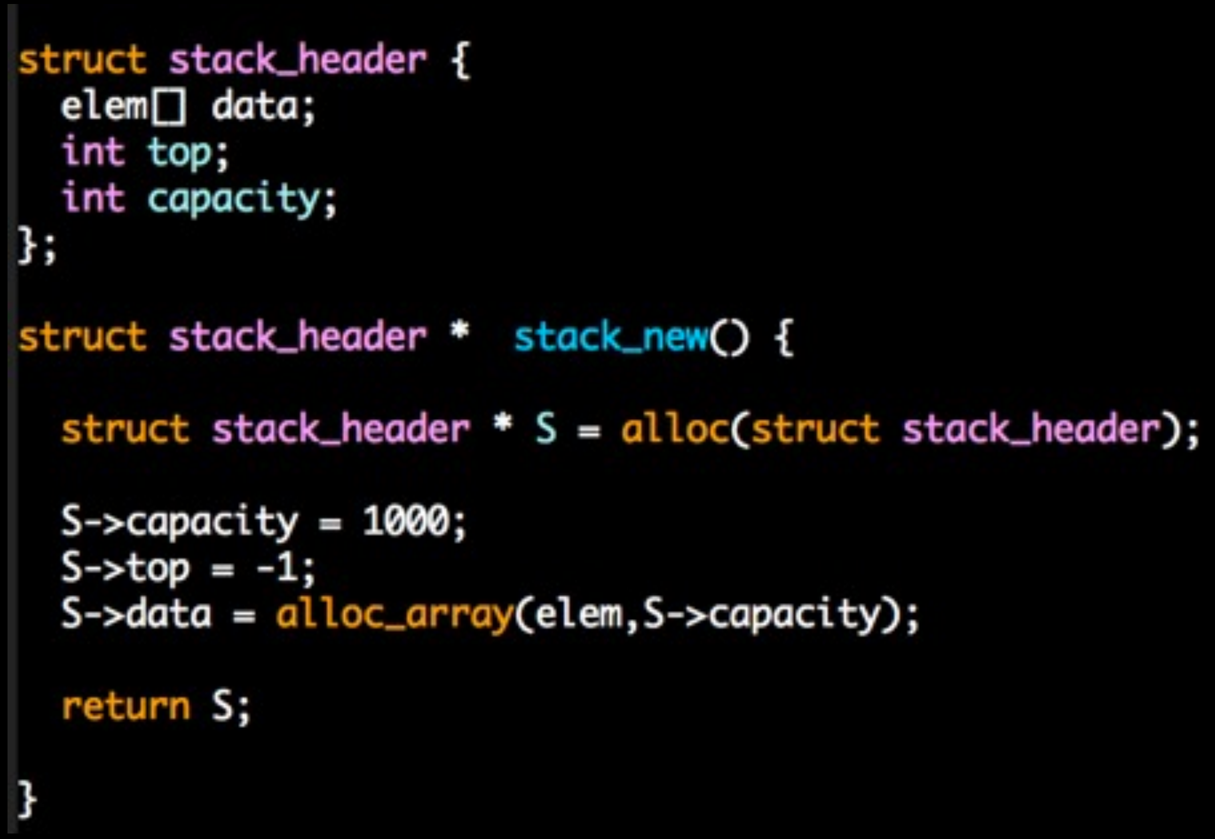
```
// returns the position of the smallest element of A[lower,upper)
int get_min(int[] A, int lower, int upper)
//@ requires 0 <= lower && lower < upper && upper <= \length(A);
//@ ensures lower <= \result && \result < upper;
//@ ensures le_seg(A[\result],A,lower,upper);
{
  for (int i = lower; i < upper; i = i + 1)
    {
      if (le_seg(A[i],A,lower,upper)) {
        return i;
      }
    }
  return -1; // will never get here!
}

void sort(int[] A, int lower, int upper)
//@ requires 0 <= lower && lower <= upper && upper <= \length(A);
//@ ensures is_sorted(A,lower,upper);
{
  for (int i = lower; i < upper ; i = i + 1)
    //@ loop_invariant lower <= i && i <= upper;
    //@loop_invariant is_sorted(A,lower,i);
    //@loop_invariant le_segs(A,lower,i,i,upper);
    {
      int smallest_index = get_min(A,i,upper);
      swap(A,i,smallest_index);
    }
  // at the end, i is upper
  // need to know is_sorted(A,lower,upper);
}
```



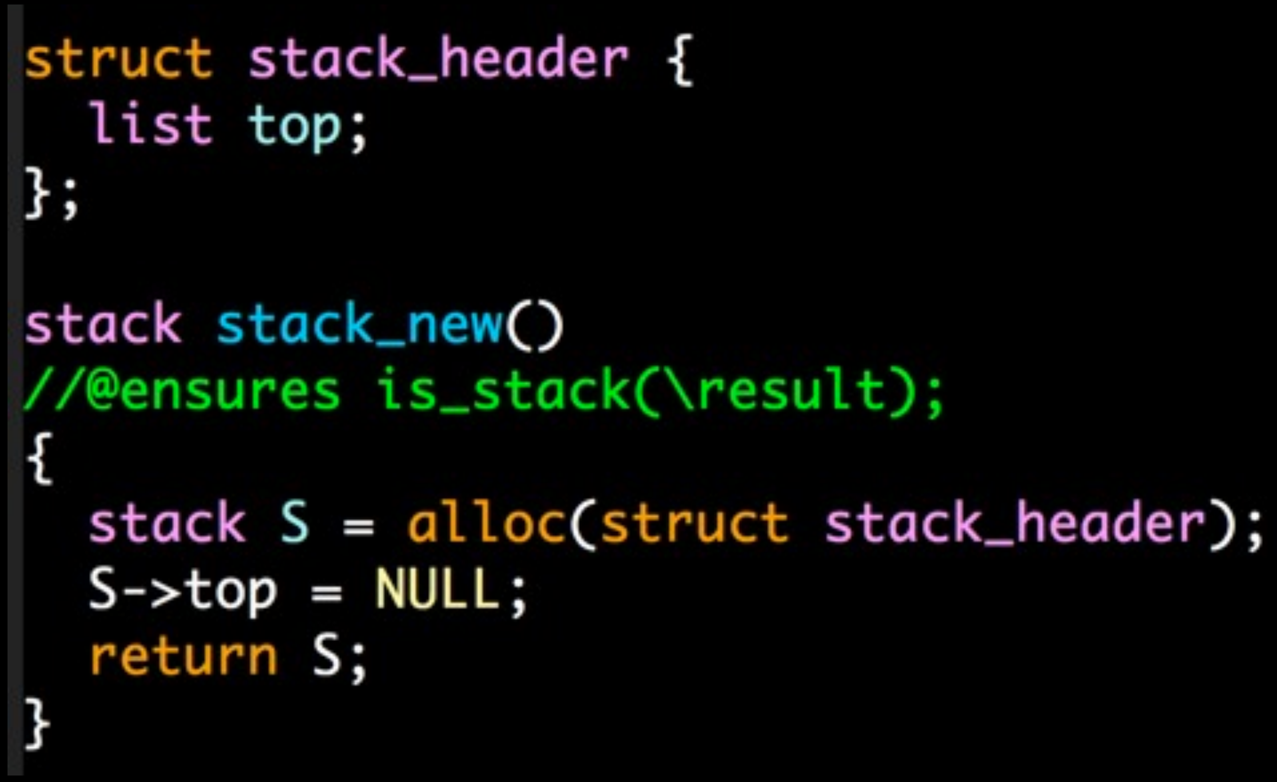
```
void qsort(int[] A, int lower, int upper)
//@requires 0 <= lower && lower <= upper && upper <= \length(A);
//@ensures is_sorted(A, lower, upper);
{
  if (upper - lower <= 1) {
    return;
  }
  else {

    // just pick the midpoint as the pivot
    int pivot_index = lower + (upper - lower)/2;

    int new_pivot_index = partition(A, lower, pivot_index, upper);

    //@assert ge_seg(A[new_pivot_index],A,lower,new_pivot_index);
    //@assert le_seg(A[new_pivot_index],A,new_pivot_index,upper);

    qsort(A, lower, new_pivot_index);
    //@assert is_sorted(A,lower,new_pivot_index);

    qsort(A, new_pivot_index + 1, upper);
    //@assert is_sorted(A,new_pivot_index+1,upper);

  }

}
```



```
// typedef __________ elem;

typedef   struct stack_header *      stack;

bool stack_empty(stack S);       /* O(1) */

stack stack_new();               /* O(1) */

void push(stack S, elem e);      /* O(1) */

elem pop(stack S)                /* O(1) */
 /*@requires !stack_empty(S);@*/;

elem peek(stack S)                /* O(1) */
 /*@requires !stack_empty(S);@*/;
```



```
struct stack_header {
  elem[] data;
  int top;
  int capacity;
};

struct stack_header *  stack_new() {

  struct stack_header * S = alloc(struct stack_header);

  S->capacity = 1000;
  S->top = -1;
  S->data = alloc_array(elem,S->capacity);

  return S;

}
```



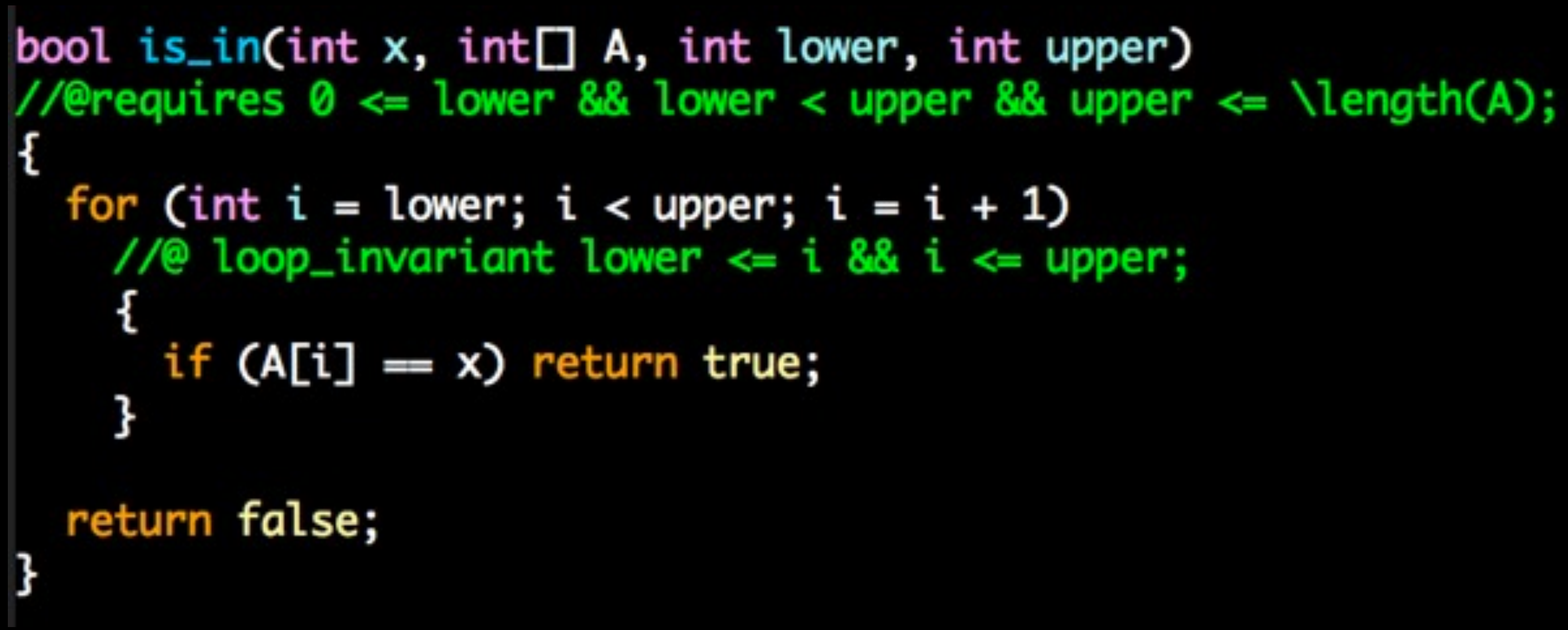
```
struct stack_header {
  list top;
};

stack stack_new()
//@ensures is_stack(\result);
{
  stack S = alloc(struct stack_header);
  S->top = NULL;
  return S;
}
```

Resources

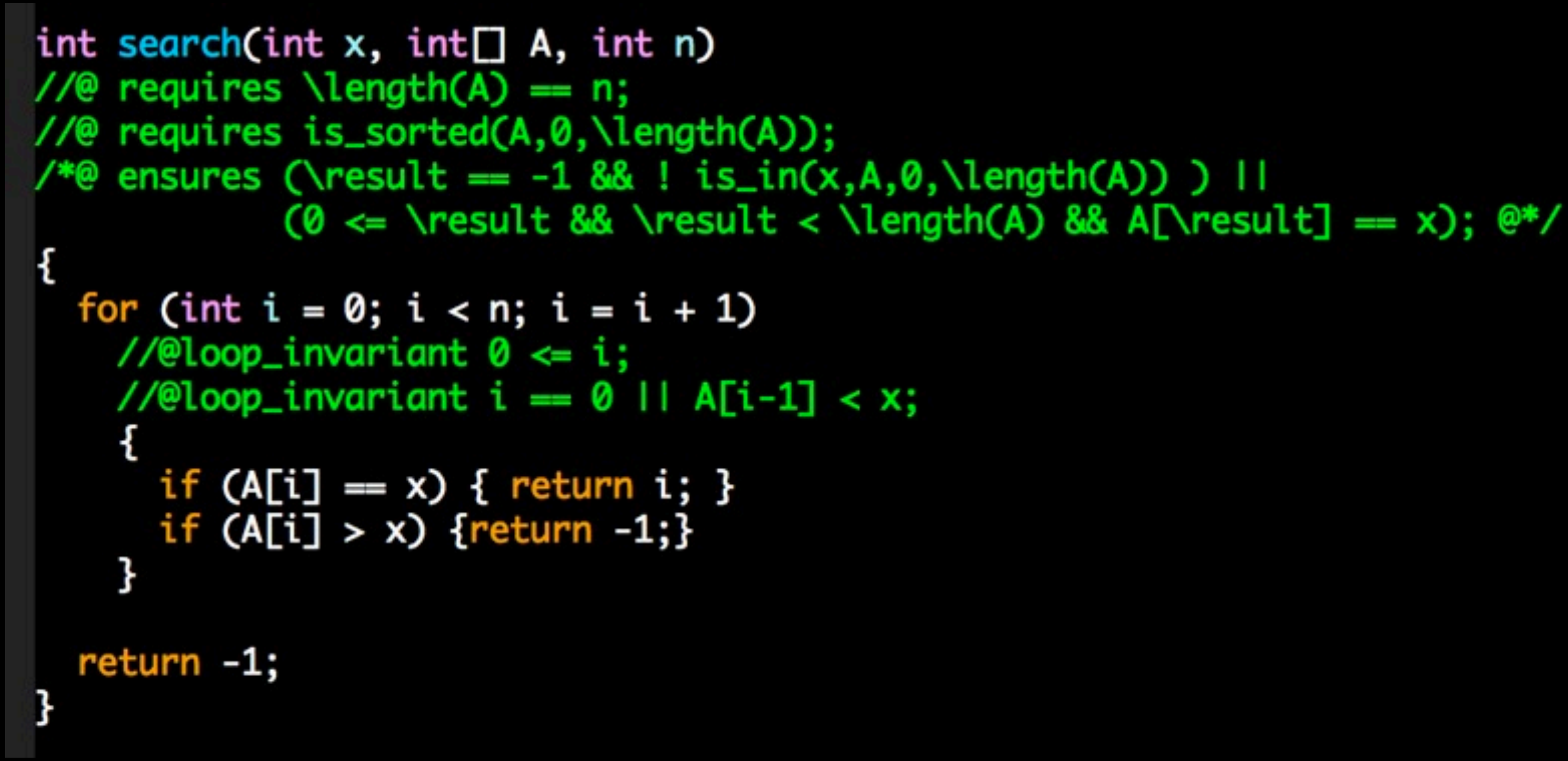
time

space

Worst case

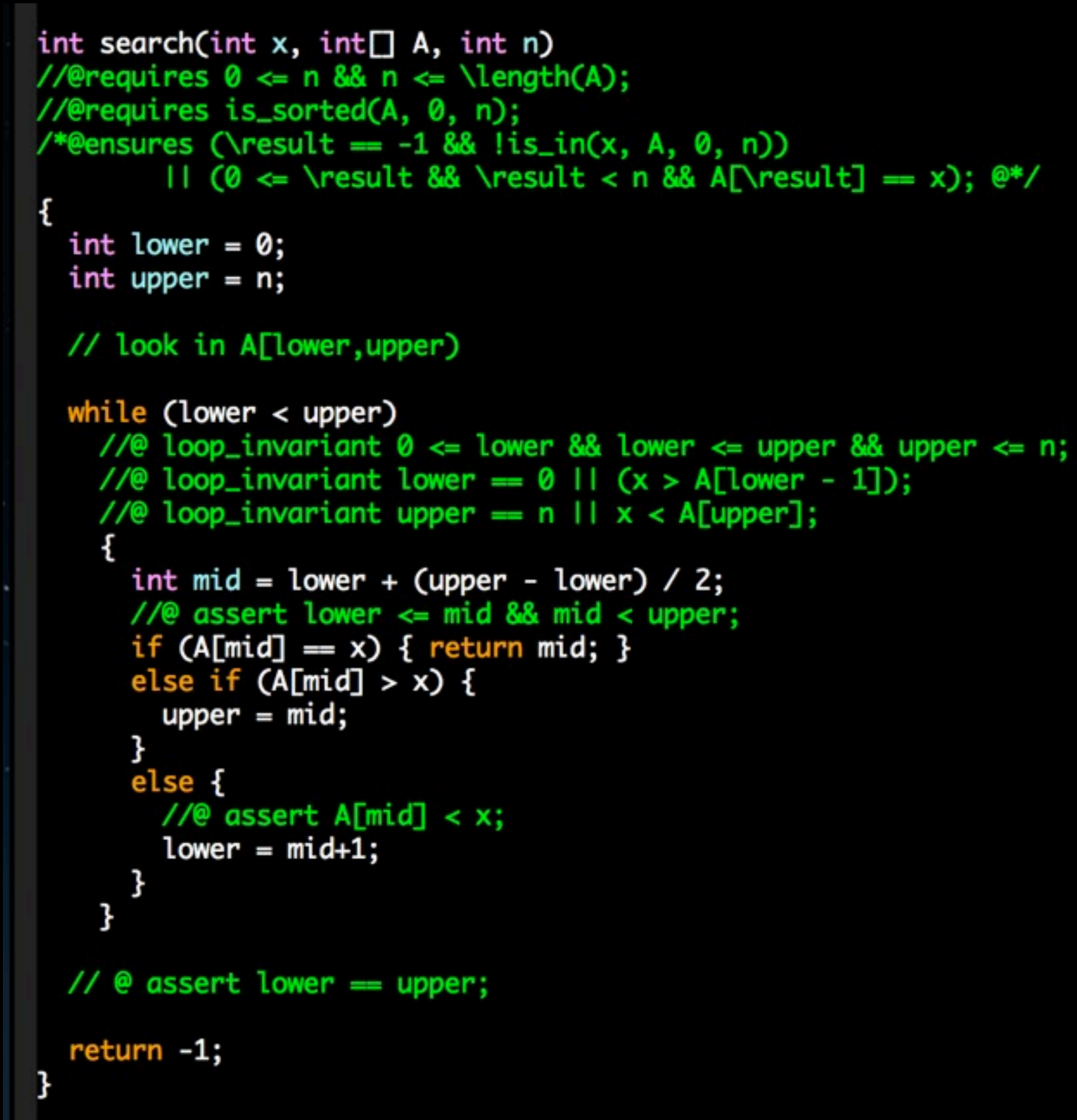


```
bool is_in(int x, int[] A, int lower, int upper)
//@requires 0 <= lower && lower < upper && upper <= \length(A);
{
  for (int i = lower; i < upper; i = i + 1)
    //@ loop_invariant lower <= i && i <= upper;
    {
      if (A[i] == x) return true;
    }

  return false;
}
```



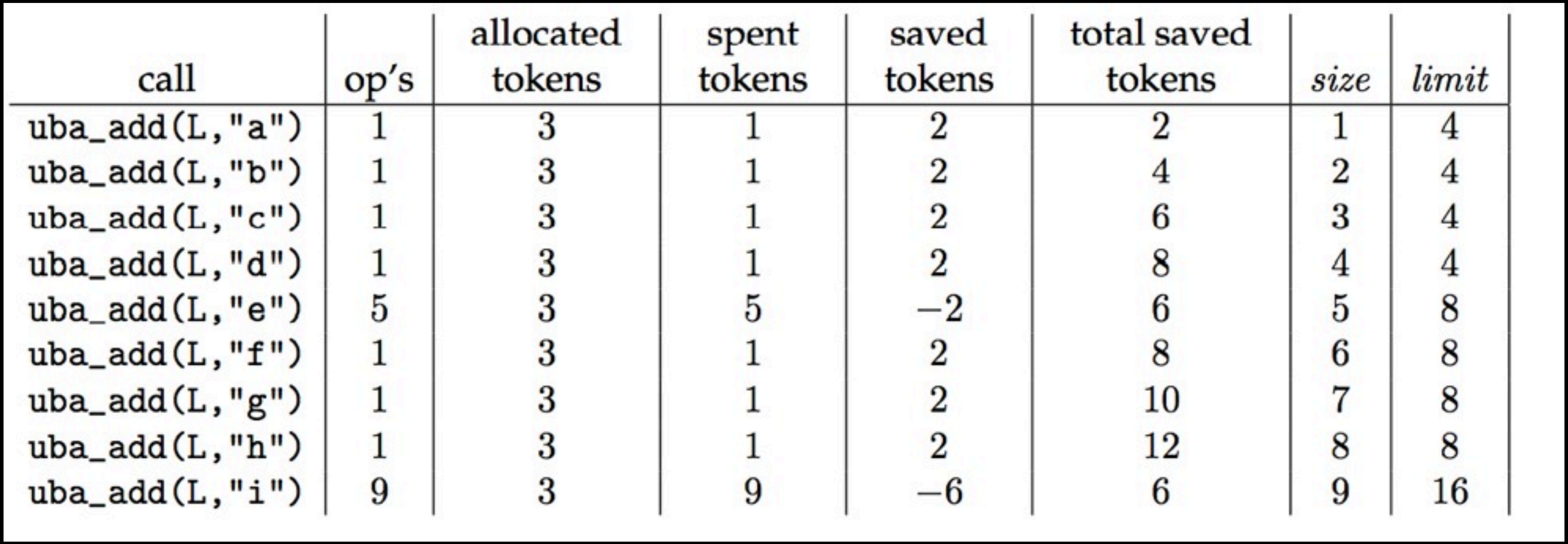
```
int search(int x, int[] A, int n)
//@ requires \length(A) == n;
//@ requires is_sorted(A,0,\length(A));
/*@ ensures (\result == -1 && ! is_in(x,A,0,\length(A)) ) ||
            (0 <= \result && \result < \length(A) && A[\result] == x); @*/
{
  for (int i = 0; i < n; i = i + 1)
    //@loop_invariant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

  return -1;
}
```

Worst case



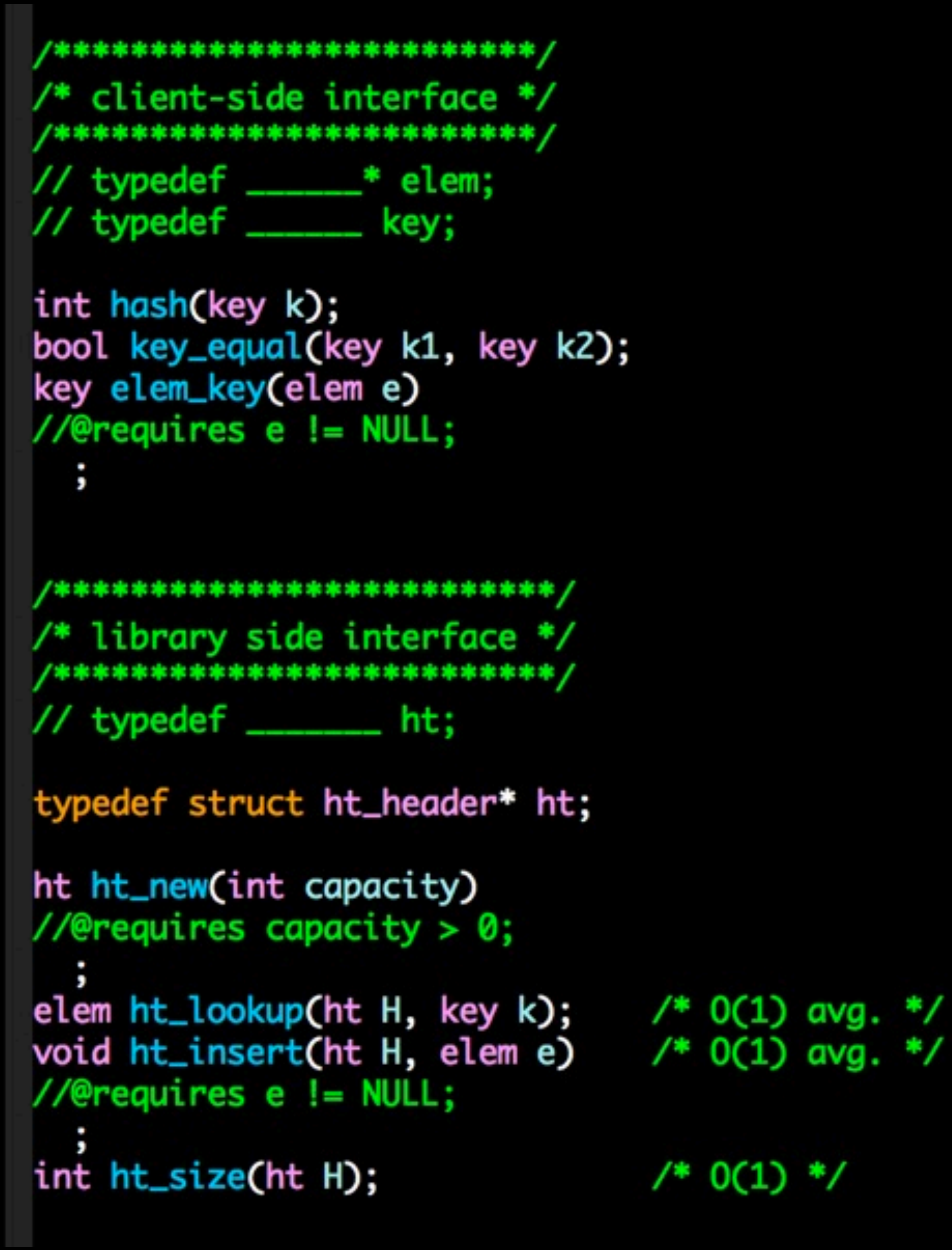
```
int search(int x, int[] A, int n)
//@requires 0 <= n && n <= \length(A);
//@requires is_sorted(A, 0, n);
/*@ensures (\result == -1 && !is_in(x, A, 0, n))
    || (0 <= \result && \result < n && A[\result] == x); @*/
{
  int lower = 0;
  int upper = n;

  // look in A[lower,upper)

  while (lower < upper)
    //@ loop_invariant 0 <= lower && lower <= upper && upper <= n;
    //@ loop_invariant lower == 0 || (x > A[lower - 1]);
    //@ loop_invariant upper == n || x < A[upper];
    {
      int mid = lower + (upper - lower) / 2;
      //@ assert lower <= mid && mid < upper;
      if (A[mid] == x) { return mid; }
      else if (A[mid] > x) {
        upper = mid;
      }
      else {
        //@ assert A[mid] < x;
        lower = mid+1;
      }
    }

  // @ assert lower == upper;

  return -1;
}
```

Amortized



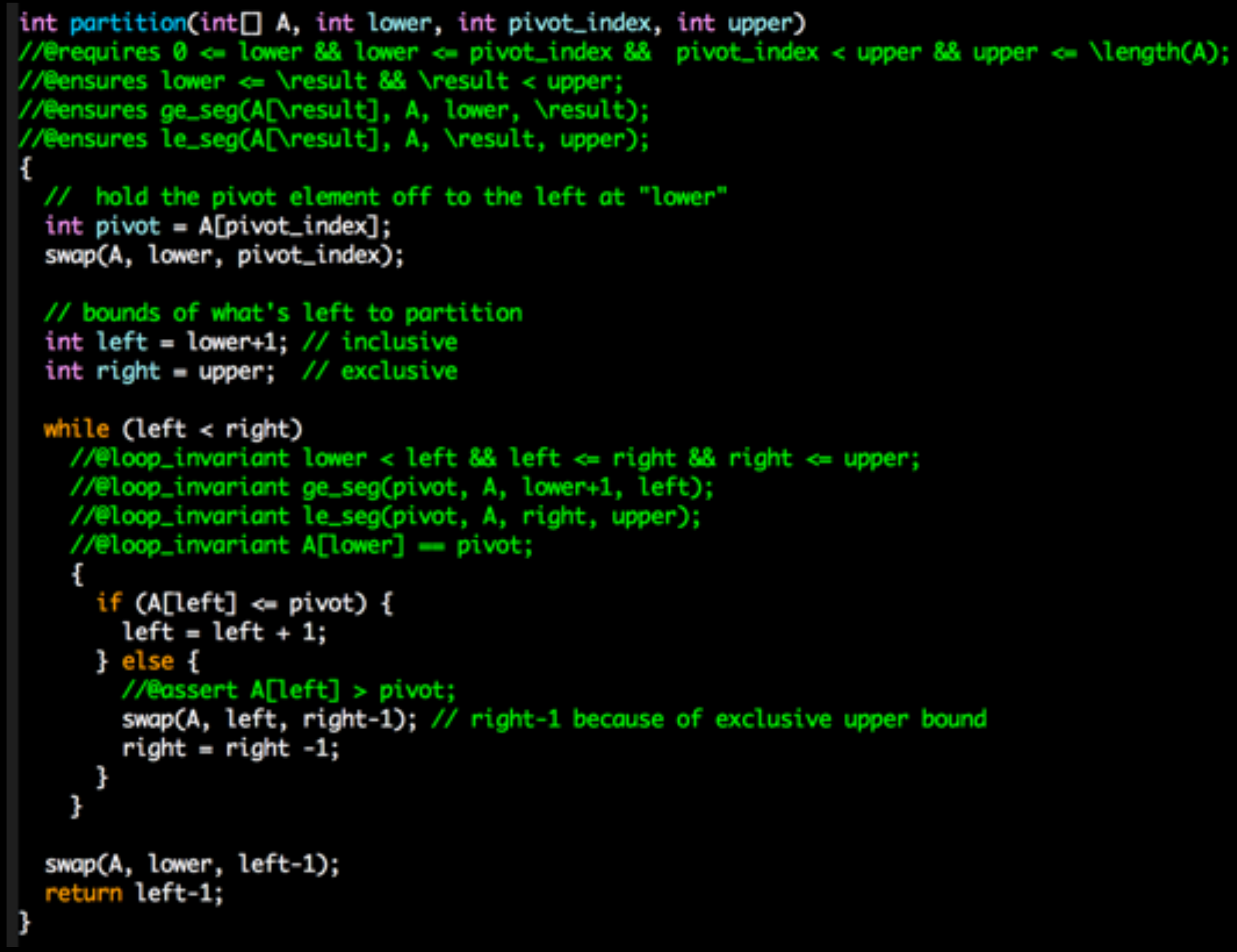
		allocated	spent	saved	total saved		
call	op's	tokens	tokens	tokens	tokens	*size*	*limit*
`uba_add(L,"a")`	1	3	1	2	2	1	4
`uba_add(L,"b")`	1	3	1	2	4	2	4
`uba_add(L,"c")`	1	3	1	2	6	3	4
`uba_add(L,"d")`	1	3	1	2	8	4	4
`uba_add(L,"e")`	5	3	5	−2	6	5	8
`uba_add(L,"f")`	1	3	1	2	8	6	8
`uba_add(L,"g")`	1	3	1	2	10	7	8
`uba_add(L,"h")`	1	3	1	2	12	8	8
`uba_add(L,"i")`	9	3	9	−6	6	9	16

Expected

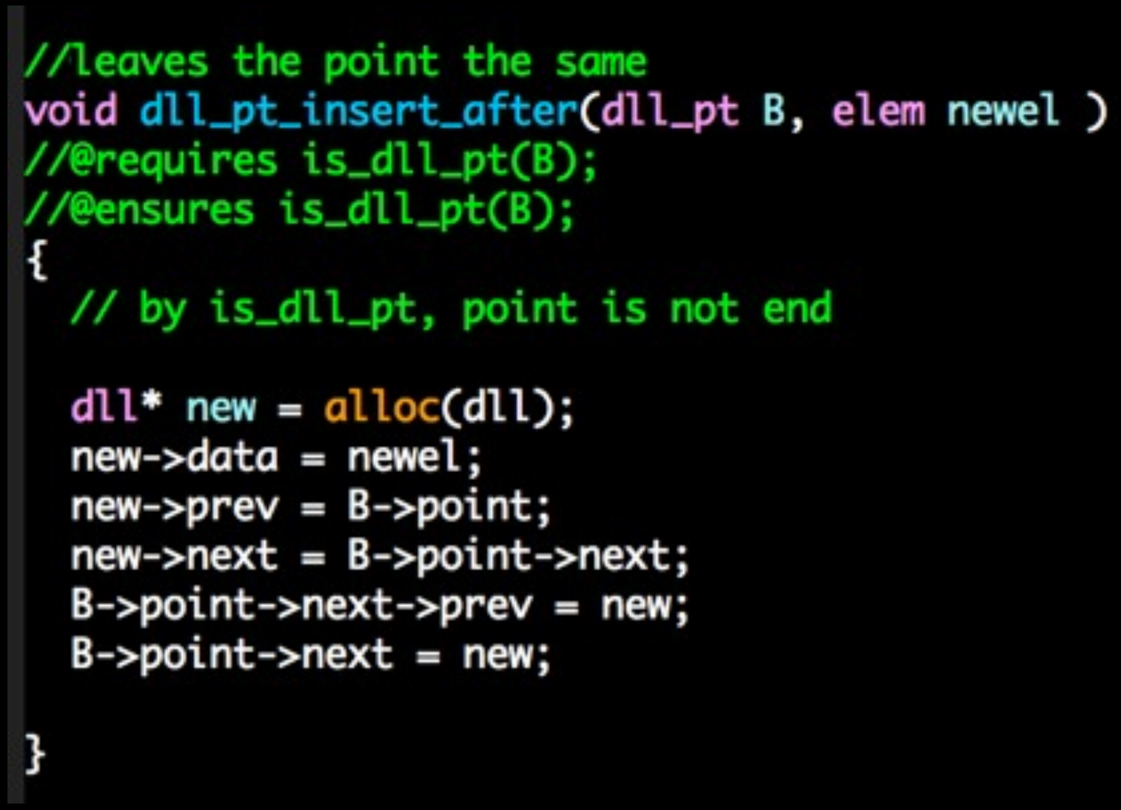


```
/*************************/
/* client-side interface */
/*************************/
// typedef ______* elem;
// typedef ______ key;

int hash(key k);
bool key_equal(key k1, key k2);
key elem_key(elem e)
//@requires e != NULL;
  ;


/**************************/
/* library side interface */
/**************************/
// typedef _______ ht;

typedef struct ht_header* ht;

ht ht_new(int capacity)
//@requires capacity > 0;
  ;
elem ht_lookup(ht H, key k);    /* O(1) avg. */
void ht_insert(ht H, elem e)    /* O(1) avg. */
//@requires e != NULL;
  ;
int ht_size(ht H);              /* O(1) */
```

Mutability for space

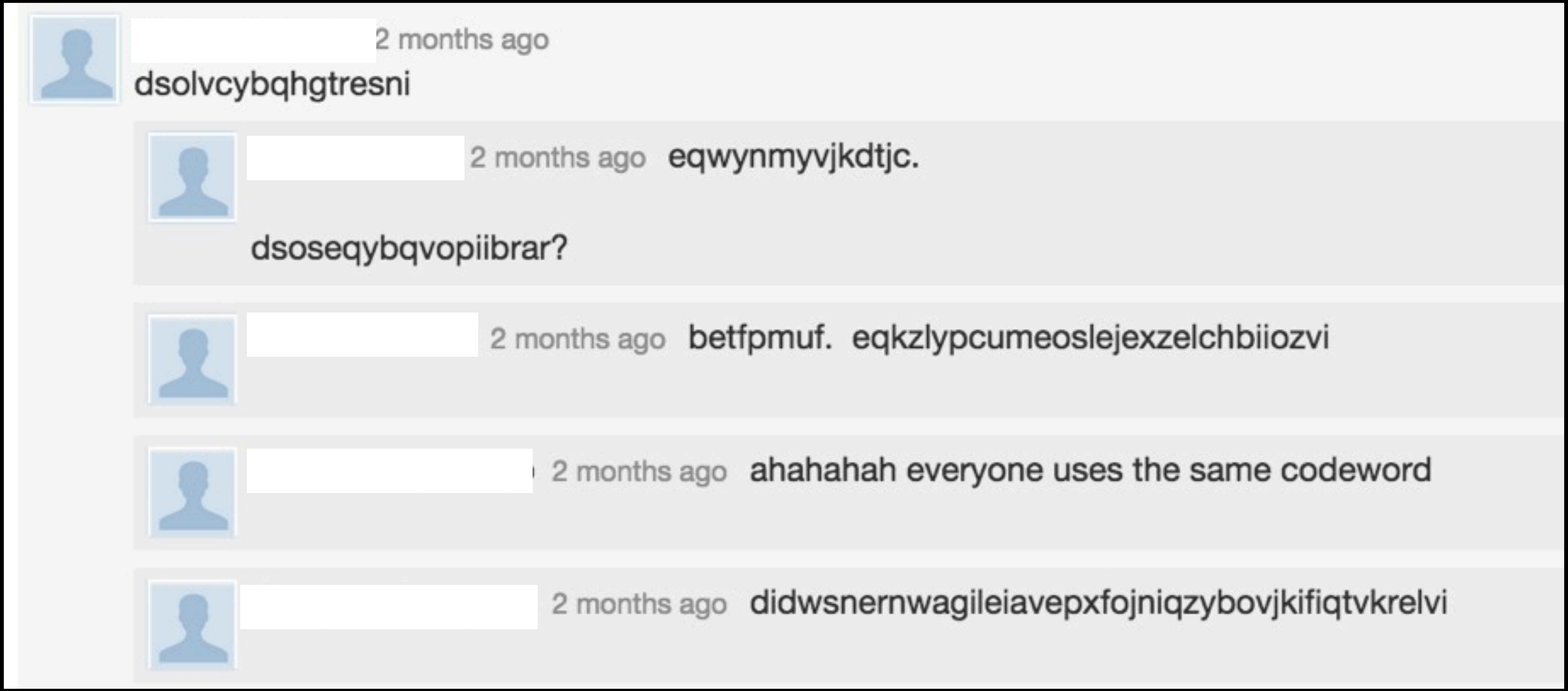


```
int partition(int[] A, int lower, int pivot_index, int upper)
//@requires 0 <= lower && lower <= pivot_index &&  pivot_index < upper && upper <= \length(A);
//@ensures lower <= \result && \result < upper;
//@ensures ge_seg(A[\result], A, lower, \result);
//@ensures le_seg(A[\result], A, \result, upper);
{
  //  hold the pivot element off to the left at "lower"
  int pivot = A[pivot_index];
  swap(A, lower, pivot_index);

  // bounds of what's left to partition
  int left = lower+1; // inclusive
  int right = upper;  // exclusive

  while (left < right)
    //@loop_invariant lower < left && left <= right && right <= upper;
    //@loop_invariant ge_seg(pivot, A, lower+1, left);
    //@loop_invariant le_seg(pivot, A, right, upper);
    //@loop_invariant A[lower] == pivot;
    {
      if (A[left] <= pivot) {
        left = left + 1;
      } else {
        //@assert A[left] > pivot;
        swap(A, left, right-1); // right-1 because of exclusive upper bound
        right = right -1;
      }
    }

  swap(A, lower, left-1);
  return left-1;
}
```



```
//leaves the point the same
void dll_pt_insert_after(dll_pt B, elem newel )
//@requires is_dll_pt(B);
//@ensures is_dll_pt(B);
{
  // by is_dll_pt, point is not end

  dll* new = alloc(dll);
  new->data = newel;
  new->prev = B->point;
  new->next = B->point->next;
  B->point->next->prev = new;
  B->point->next = new;

}
```

Representation



[redacted] 2 months ago
dsolvcybqhgtresni

[redacted] 2 months ago eqwynmyvjkdtjc.

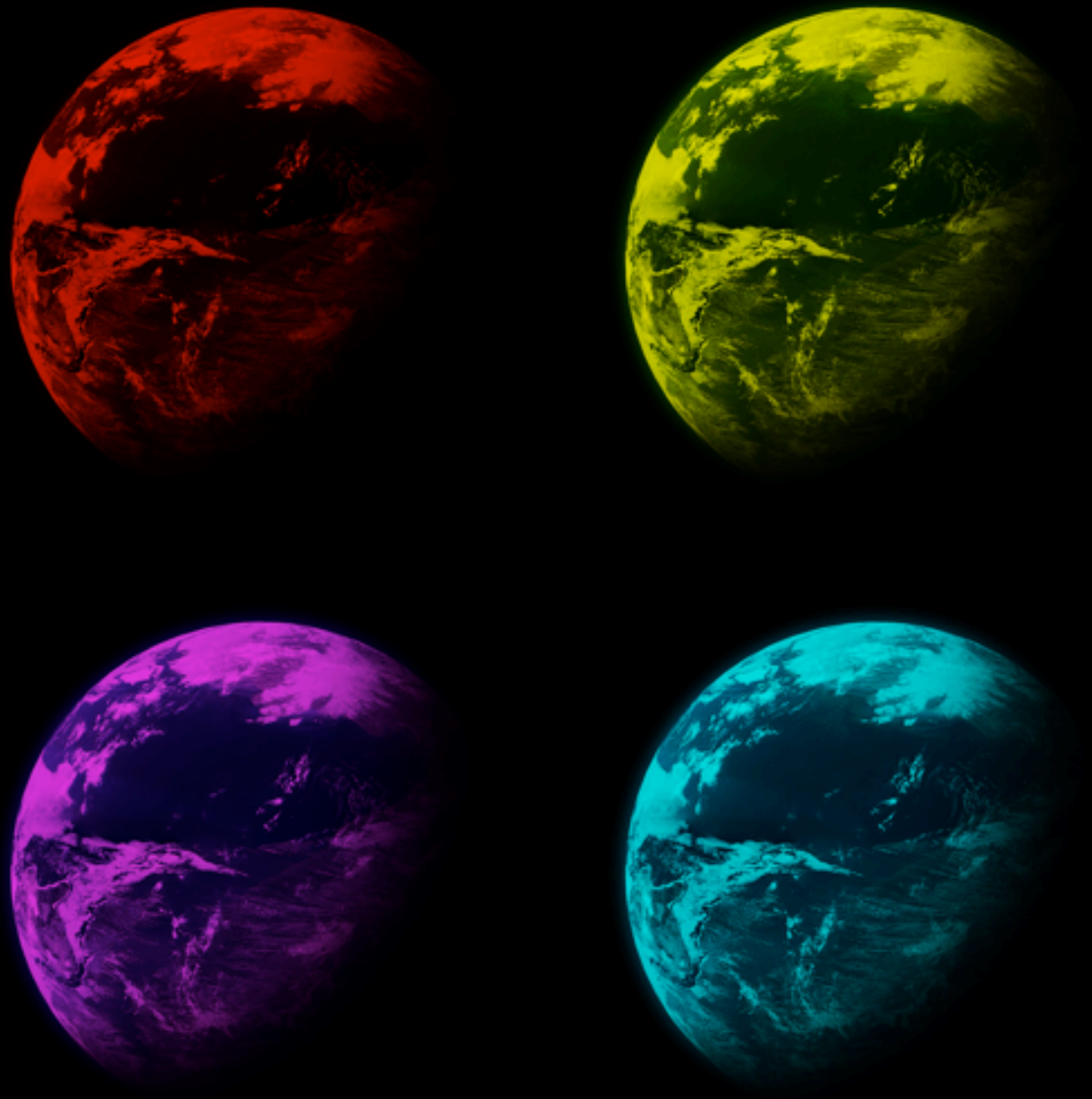
dsoseqybqvopiibrar?

[redacted] 2 months ago betfpmuf. eqkzlypcumeoslejexzelchbiiozvi

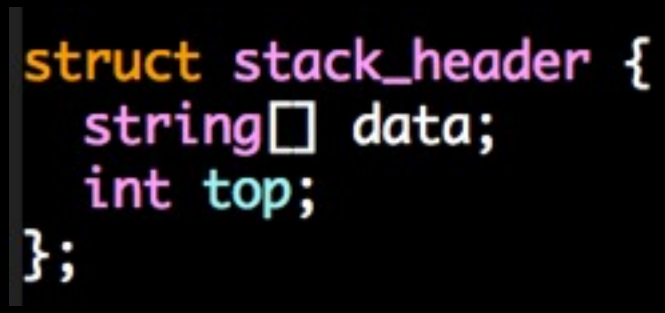
[redacted] 2 months ago ahahahah everyone uses the same codeword

[redacted] 2 months ago didwsnernwagileiavepxfojniqzybovjkifiqtvkrelvi

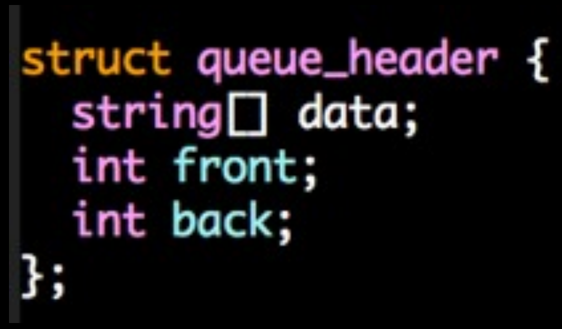
Representation



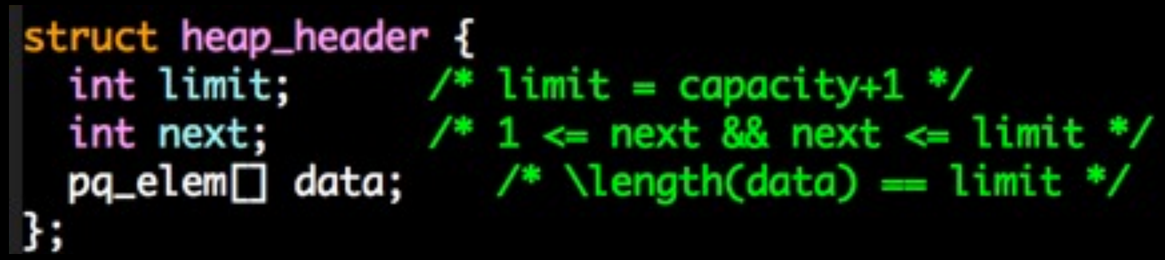
Representation



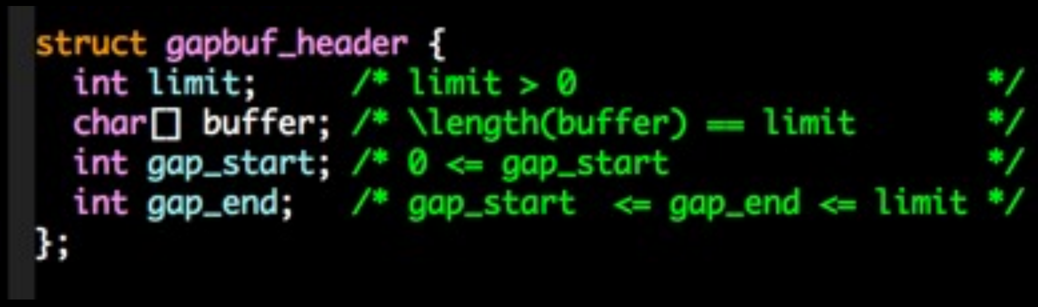
```
struct stack_header {
  string[] data;
  int top;
};
```



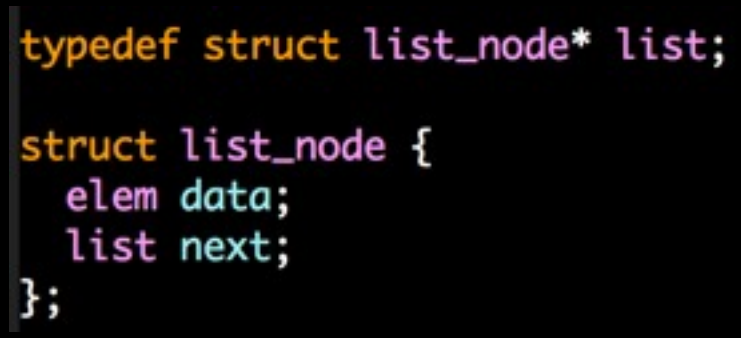
```
struct queue_header {
  string[] data;
  int front;
  int back;
};
```



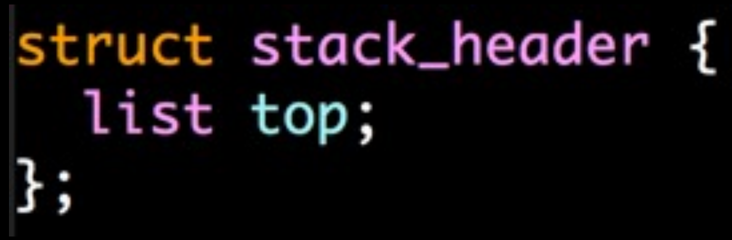
```
struct heap_header {
  int limit;        /* limit = capacity+1 */
  int next;         /* 1 <= next && next <= limit */
  pq_elem[] data;     /* \length(data) == limit */
};
```



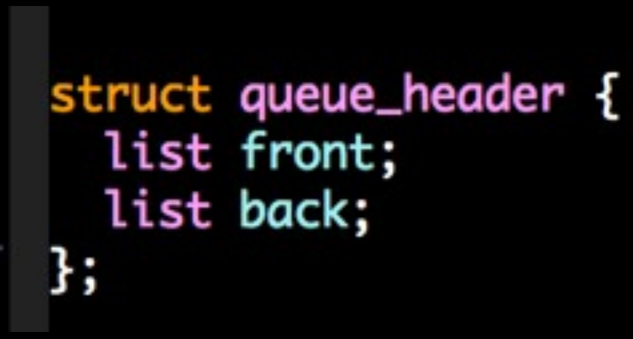
```
struct gapbuf_header {
  int limit;      /* limit > 0                   */
  char[] buffer; /* \length(buffer) == limit      */
  int gap_start; /* 0 <= gap_start                */
  int gap_end;   /* gap_start  <= gap_end <= limit */
};
```



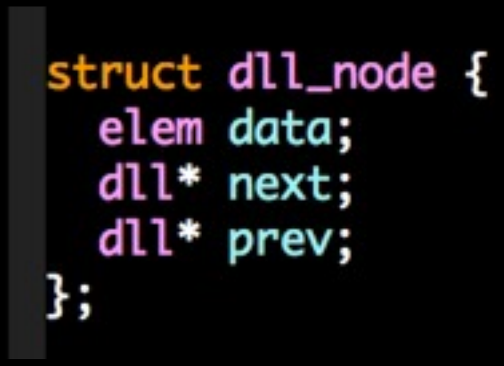
```
typedef struct list_node* list;

struct list_node {
  elem data;
  list next;
};
```



```
struct stack_header {
  list top;
};
```

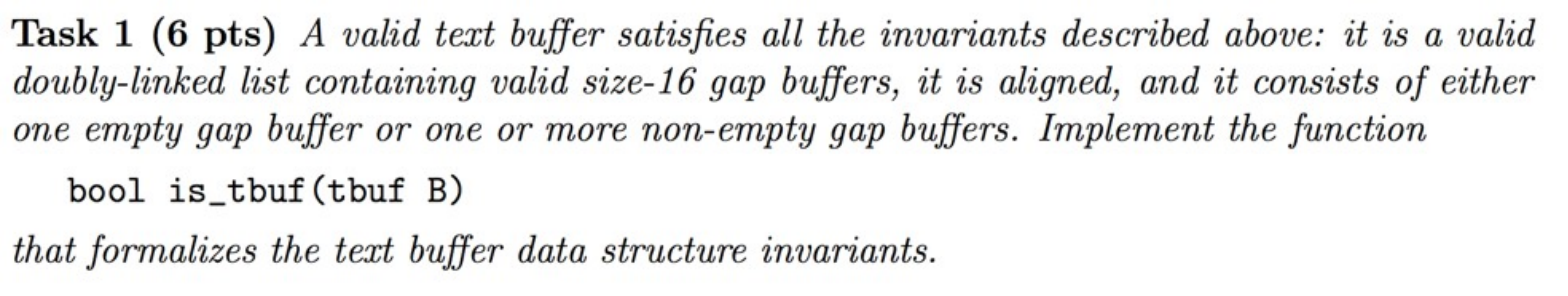


```
struct queue_header {
  list front;
  list back;
};
```



```
struct dll_node {
  elem data;
  dll* next;
  dll* prev;
};
```

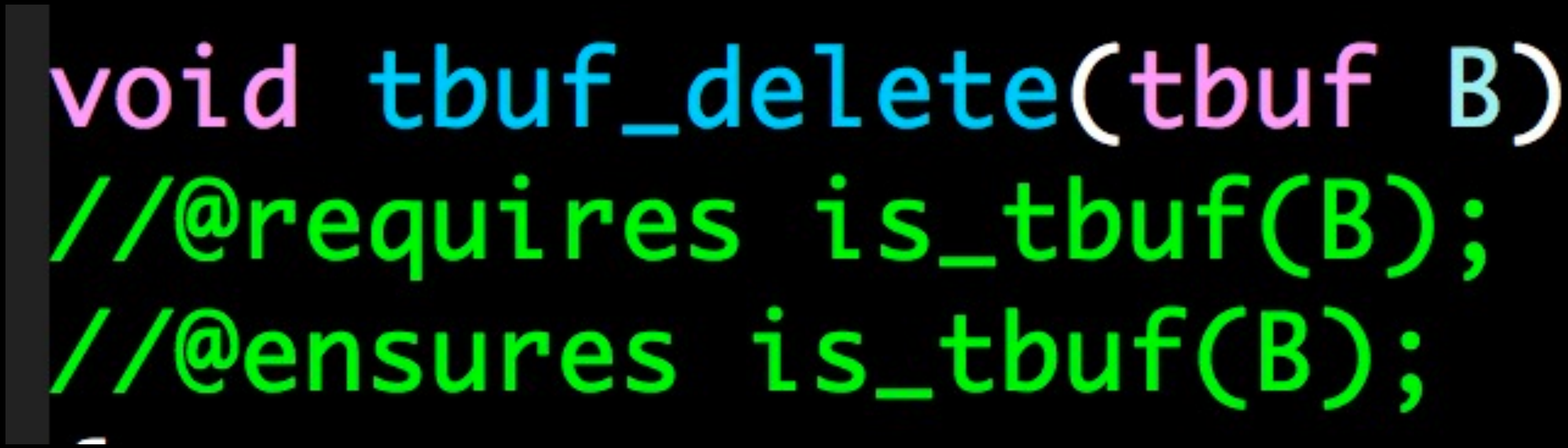
Representation invariants



Task 1 (6 pts) *A valid text buffer satisfies all the invariants described above: it is a valid doubly-linked list containing valid size-16 gap buffers, it is aligned, and it consists of either one empty gap buffer or one or more non-empty gap buffers. Implement the function*

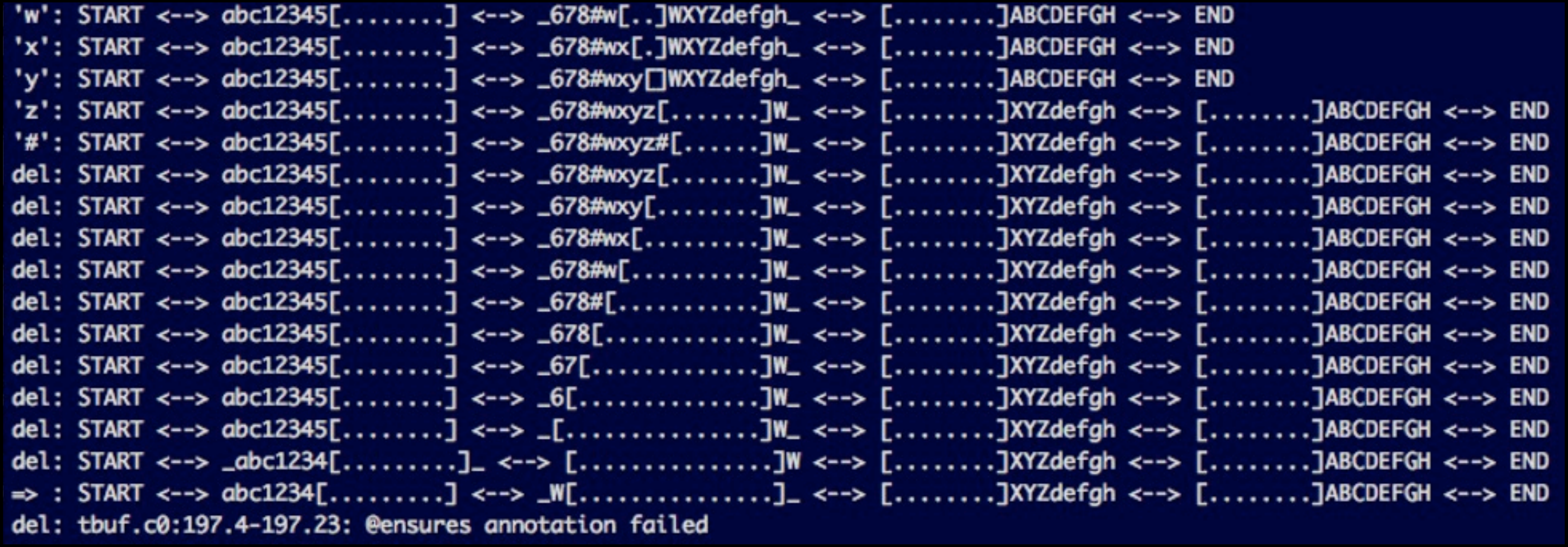
```
bool is_tbuf(tbuf B)
```

that formalizes the text buffer data structure invariants.



```
void tbuf_delete(tbuf B)
//@requires is_tbuf(B);
//@ensures is_tbuf(B);
```

Local Reasoning



```
'w': START <--> abc12345[........] <--> _678#w[..]WXYZdefgh_ <--> [........]ABCDEFGH <--> END
'x': START <--> abc12345[........] <--> _678#wx[.]WXYZdefgh_ <--> [........]ABCDEFGH <--> END
'y': START <--> abc12345[........] <--> _678#wxy□WXYZdefgh_ <--> [........]ABCDEFGH <--> END
'z': START <--> abc12345[........] <--> _678#wxyz[.......]W_ <--> [........]XYZdefgh <--> [........]ABCDEFGH <--> END
'#': START <--> abc12345[........] <--> _678#wxyz#[......]W_ <--> [........]XYZdefgh <--> [........]ABCDEFGH <--> END
del: START <--> abc12345[........] <--> _678#wxyz[.......]W_ <--> [........]XYZdefgh <--> [........]ABCDEFGH <--> END
del: START <--> abc12345[........] <--> _678#wxy[........]W_ <--> [........]XYZdefgh <--> [........]ABCDEFGH <--> END
del: START <--> abc12345[........] <--> _678#wx[.........]W_ <--> [........]XYZdefgh <--> [........]ABCDEFGH <--> END
del: START <--> abc12345[........] <--> _678#w[..........]W_ <--> [........]XYZdefgh <--> [........]ABCDEFGH <--> END
del: START <--> abc12345[........] <--> _678#[...........]W_ <--> [........]XYZdefgh <--> [........]ABCDEFGH <--> END
del: START <--> abc12345[........] <--> _678[............]W_ <--> [........]XYZdefgh <--> [........]ABCDEFGH <--> END
del: START <--> abc12345[........] <--> _67[.............]W_ <--> [........]XYZdefgh <--> [........]ABCDEFGH <--> END
del: START <--> abc12345[........] <--> _6[..............]W_ <--> [........]XYZdefgh <--> [........]ABCDEFGH <--> END
del: START <--> abc12345[........] <--> _[...............]W_ <--> [........]XYZdefgh <--> [........]ABCDEFGH <--> END
del: START <--> _abc1234[.........]_ <--> [...............]W <--> [........]XYZdefgh <--> [........]ABCDEFGH <--> END
=> : START <--> abc1234[.........] <--> _W[..............]_ <--> [........]XYZdefgh <--> [........]ABCDEFGH <--> END
del: tbuf.c0:197.4-197.23: @ensures annotation failed
```

Building Blocks

conceptual

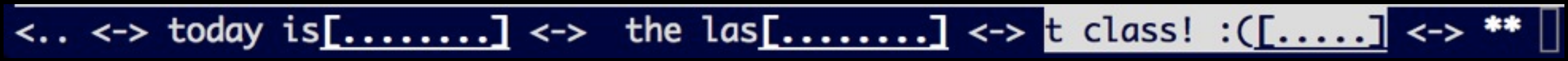
loops
sortedness
randomness
divide in half

concrete

(unbounded) arrays
searching
sorting
stacks/queues/PQs
(doubly) linked lists
dicts, hash tables
trees
graphs

Building Blocks

tbuf = gapbuf + DLL

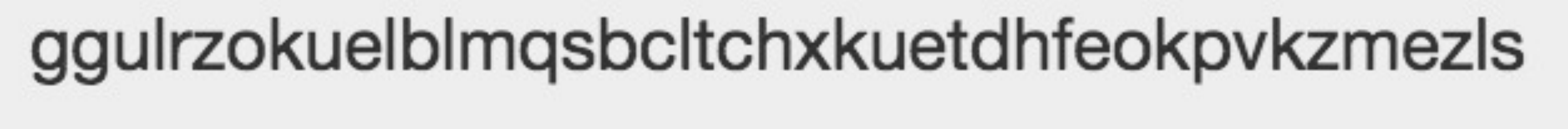


```
<.. <-> today is[........] <->  the las[........] <-> t class! :([.....] <-> ** 
```

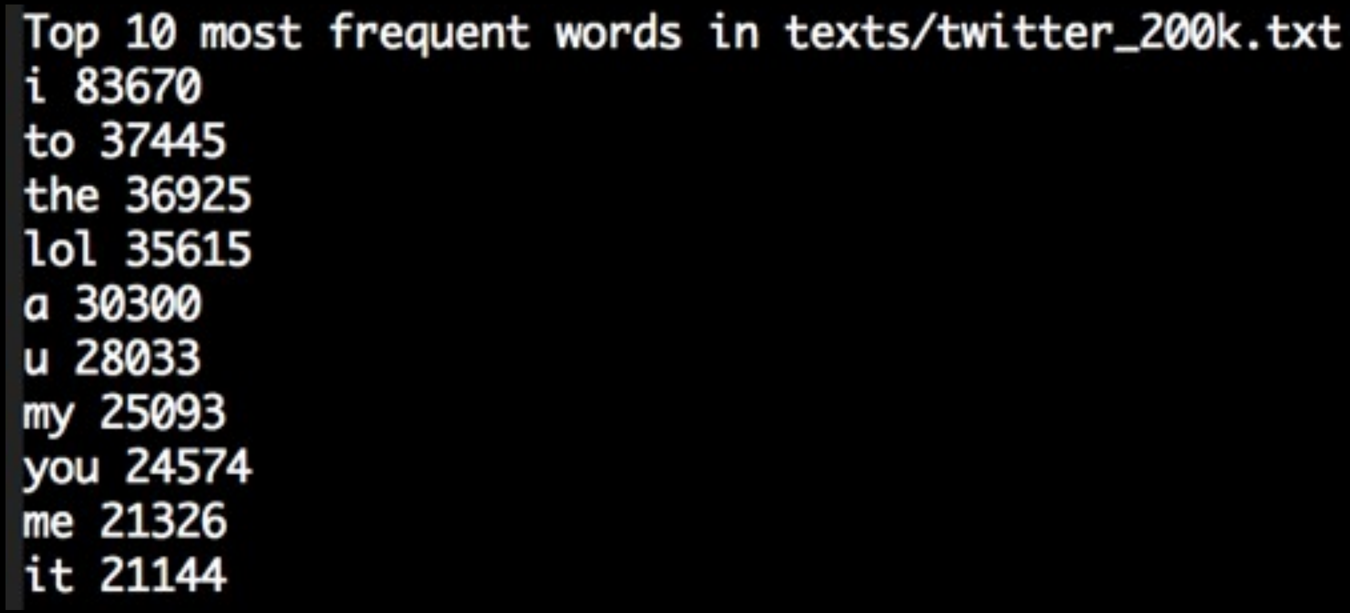
compression = tree + PQ + dict + stack

graph search = stack || queue || PQ

How does it work?

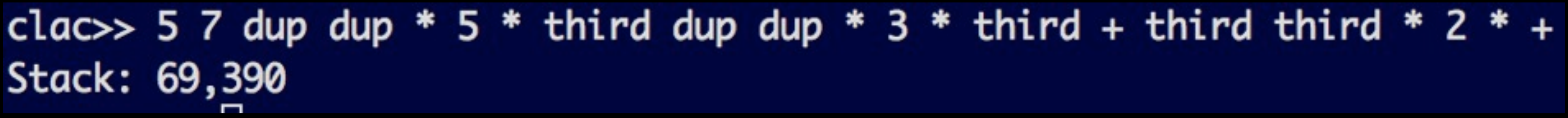


ggulrzokuelblmqsbcltchxkuetdhfeokpvkzmezls

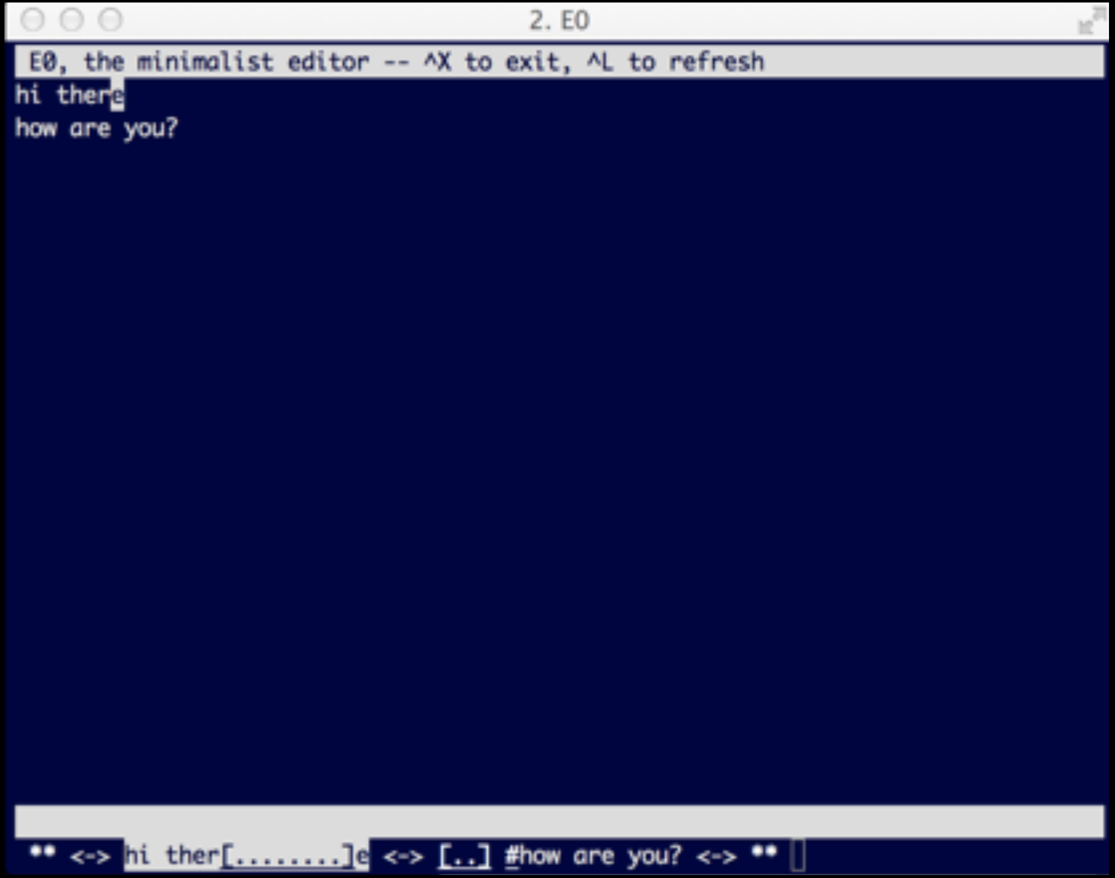


```
Top 10 most frequent words in texts/twitter_200k.txt
i 83670
to 37445
the 36925
lol 35615
a 30300
u 28033
my 25093
you 24574
me 21326
it 21144
```

How does it work?



```
clac>> 5 7 dup dup * 5 * third dup dup * 3 * third + third third * 2 * +
Stack: 69,390
```



```
 E0, the minimalist editor -- ^X to exit, ^L to refresh
hi there
how are you?



 ** <-> hi ther[........]e <-> [..] #how are you? <-> **
```

What’s next?

COMP 212: functional programming, parallelism

MATH 228: proofs, number theory, graphs, …

COMP 331: the machine

COMP 321: programming languages

COMP 312: algorithms and complexity

COMP 301: models and limits of computation

Electives: software engineering, bioinformatics, artificial intelligence, proof assistants