Next: Correctness

- How can we show that the algorithm works correctly for all possible inputs of all possible sizes?
- Exhaustive testing not feasible.
- Analytical techniques are ~~useful~~ essential here.

Find-max revisited

Q1. Find the max of n numbers (stored in array A)

Formal specs:

INPUT: A[1..n] - an array of integers

OUTPUT: an element m of A such that $A[j] \leq m$, $1 \leq j \leq \text{length}(A)$

Find-max (A)

```
max ← A[1]
for j ← 2 to length(A)
    do if (max < A[j])
            max ← A[j]
return max
```

Correctness Proof 1

INPUT: A[1..n] - an array of integers

OUTPUT: an element m of A such that $m \leq A[j]$,
$1 \leq j \leq \text{length}(A)$

```
Find-max (A)
 1. max ← A[1]
 2. for j ← 2 to length(A)
 3.     do if (max < A[j])
 4.             max ← A[j]
 5. return max
```

Prove that for any valid Input, the output of Find-max satisfies the output condition.

Proof 1 [by contradiction]: Suppose the algorithm is incorrect. Then for some input A,

(a) max is not an element of A or

(b) $(\exists j \mid max < A[j])$.

max is initialized to and assigned to elements of A – so (a) is impossible. WHY?

(b) After the j^{th} iteration of the for-loop (lines 2 – 4), $max \geq A[j]$. From lines 3,4, max only increases.

Therefore, upon termination, $max \geq A[j]$, which contradicts (b).

Correctness Proof 1 - comments

- The preceding proof reasons about the whole algorithm
- It is possible to prove correctness by induction as well: this is left as an exercise for you.
- What if the algorithm/program was very big and had many function calls, nested loops, if-then’s and other standard features?
- Need a simpler, more “modular” strategy.

Correctness Proof – 2 (typos fixed)

INPUT: A[1..n] - an array of integers
OUTPUT: an element m of A such that $m \leq A[j]$,
$1 \leq j \leq \text{length}(A)$

```
Find-max (A)
 1. max $\leftarrow$ A[1]
 2. for j $\leftarrow$ 2 to length(A)
 3.     do if (max < A[j])
 4.             max $\leftarrow$ A[j]
 5. return max
```

Prove that for any valid Input, the output of Find-max satisfies the output condition.

Proof 2 [use loop invariants]:

(identify invariant) At the beginning of iteration j of for loop, max contains the maximum of A[1..j-1].

(Proof) Clearly true for j=2. For j = 3,4,…, assume that invariant holds for j-1. So at the beginning of iteration j-1 max contains the maximum of A[1..j-2].

Case (a) A[j-1] is the maximum of A[1..j-1]. In lines 3,4, max is set to A[j-1].

Case (b) A[j-1] is not the maximum of A[1..j-1], so the maximum of A[1..j-1] is in A[1..j-2]. By our assumption max already has this value and by lines 3-4 max is unchanged in this iteration.

Correctness Proof – continued

INPUT: A[1..n] - an array of integers

OUTPUT: an element m of A such that $m \le A[j]$,
$1 \le j \le$ length(A)

```
Find-max (A)
 max ← A[1]
 for j ← 2 to length(A)
     do if (max < A[j])
             max ← A[j]
 return max
```

Proof using loop invariants - continued:

We proved that the invariant holds at the beginning of iteration j for each j used by Find-max.

Upon termination, j = length(A)+1. (WHY?)

The invariant holds, and so max contains the maximum of A[1..n]

-- STRUCTURED PROOF TECHNIQUE!

-- VERY SIMILAR TO INDUCTION!

We will see more non-trivial examples later.

More about correctness

- Don’t tack on a formal proof of correctness after coding to make the professor happy.
- It need not be mathematical mumbo jumbo.
- Goal: To think about algorithms in such way that their correctness is transparent.

1. Iterative Algorithms

“Take one step at a time towards the final destination”

```
loop (until done)
        take step
end loop
```

2. Recursive Algorithms

LATER.

Loop invariants

A good way to structure many programs:

- Store the key information you currently know in some data structure.
- In the main loop,
  - take a step forward towards destination

    by making a simple change to this data.

Insertion sort - correctness

```
for j=2 to length(A)
   do key=A[j]
      i=j-1
      while i>0 and A[i]>key
        do A[i+1]=A[i]
           i--
      A[i+1]:=key
```

What is a good loop invariant?

It is easy to write a loop invariant if you understand what the algorithm does.

Use assertions.

Assertions

An assertion is a statement about the current state of the data structure that is either true or false.

Useful for

- thinking about algorithms
- developing
- describing
- proving correctness

An assertion need not consist of formal/math mumbo jumbo

Use an informal description

An assertion is not a task for the algorithm to perform.

It is only a comment that is added for the benefit of the reader.

Assertions – contd.

Example of Assertions

- Preconditions: Any assumptions that must be true about the input instance.
- Postconditions: The statement of what must be true when the algorithm/program returns.

Correctness:

$$\langle \text{PreCond} \rangle \ \& \ \langle \text{code} \rangle \Rightarrow \langle \text{PostCond} \rangle$$

If the input meets the preconditions,
then the output must meet the postconditions.

If the input does not meet the preconditions,
then nothing is required.

Assertions – contd.

Example of Assertions

```
<preCond>
codeA
loop
    <loop-invariant>
    exit when <exit Cond>
    codeB
endloop
codeC
<postCond>
```

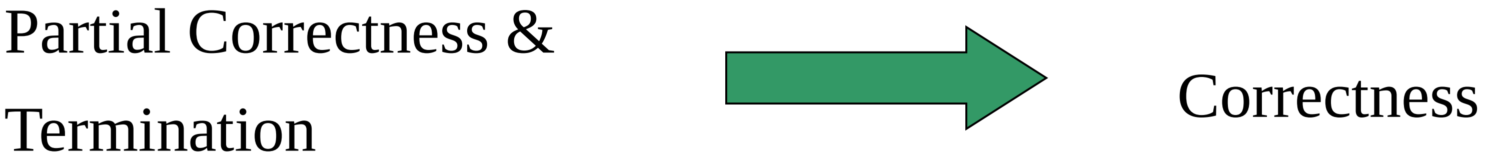
Partial correctness

We must show three things about loop invariants:

- **Initialization** – it is true prior to the first iteration
- **Maintenance** – if it is true before an iteration, it remains true before the next iteration

Termination – when loop terminates the invariant gives a useful property to show the correctness of the algorithm

Proves that IF the program terminates then it works



Partial Correctness &

Termination

Correctness

Correctness of Insertion sort

```
for j=2 to length(A)
  do key=A[j]
    //Insert A[j] into the sorted
    //sequence A[1..j-1]
    i=j-1
    while i>0 and A[i]>key
      do A[i+1]=A[i]
         i--
    A[i+1]:=key
```

Invariant: *at the start of* ***for*** *loop iteration j, A[1…j-1] consists of elements originally in A[1…j-1] but in sorted order*

Initialization: *j = 2,* the invariant trivially holds because *A*[1] is a sorted array ☺

Correctness of Insertion sort – contd.

```
for j=2 to length(A)
  do key=A[j]
    i=j-1
    while i>0 and A[i]>key
      do A[i+1]=A[i]
         i--
    A[i+1]:=key
```

Invariant: *at the start of* ***for*** *loop iteration j, A[1…j-1] consists of elements originally in A[1…j-1] but in sorted order*

Maintenance: the inner **while** loop moves elements $A[j-1]$, $A[j-2]$, …, $A[k]$ one position right without changing their order. Then the former $A[j]$ element is inserted into k^{th} position so that $A[k-1] \leq A[k] \leq A[k+1]$.

$$A[1\ldots j-1] \text{ sorted} + A[j] \rightarrow A[1\ldots j] \text{ sorted}$$

Correctness of Insertion sort – contd.

```
for j=2 to length(A)
   do key=A[j]
      Insert A[j] into the sorted
      sequence A[1..j-1]
      i=j-1
      while i>0 and A[i]>key
        do A[i+1]=A[i]
           i--
      A[i+1]:=key
```

Invariant: *at the start of*

***for** loop iteration j, A[1…j-1]*

consists of elements

originally in A[1…j-1] but in

sorted order

Termination: the loop terminates, when *j=n+1*.

Then the invariant states: *“A[1…n] consists of elements originally in A[1…n] but in sorted order”* ☺

More on correctness of iterative algorithms

1. Spent some time formalizing asymptotic notation.
2. Have seen insertion-sort and loop invariants for it. The invariant falls under the "more of the input" class in Jeff Edmonds' notation.
3. Next, selection sort; the invariant for this falls under the "more of the output" class in Jeff Edmonds' notation.

Loop invariants

Recall that

1. Loop invariants allow you to reason about a single iteration of the loop.
2. The test condition of the loop is not part of the invariant.
3. Design the loop invariant so that when the termination condition is attained, and the invariant is true, then the goal is reached: invariant + termination => goal
4. Create invariants which are
   -- simple, and
   -- capture all the goals of the algorithm (except termination)

It takes practice

It is best to use mathematical symbols for loop invariants; when this is too complicated, use clear prose and common sense.

Selection sort

I/O specs: same as insertion sort

Algorithm: Given an array A of n integers, sort them by repetitively selecting the smallest among the yet unselected integers.

Is this precise enough?

Swap the smallest integer with the integer currently in the place where the smallest integer should go.

Loop invariant: at the beginning of the j^{th} iteration

- The smallest j-1 values are sorted in descending order in locations [1,j-1]

- Is this enough? No….

and the rest are in locations [n-j,n].

See if you can prove it.

Another kind of loop invariant

Narrowed the search space, e.g. Binary search

- Preconditions
  - –Key 25
  - –Sorted List

3	5	6	13	18	21	21	25	36	43	49	51	53	60	72	74	83	88	91	95

- Postcondition

–Find key in list (if present).

3	5	6	13	18	21	21	25	36	43	49	51	53	60	72	74	83	88	91	95

Define Loop Invariant

- Maintain a sublist.
- If the key is contained in the original list, then the key is contained in the sublist.

Define an iteration of loop

•Cut sublist in half.

•Determine which half the key would be in.

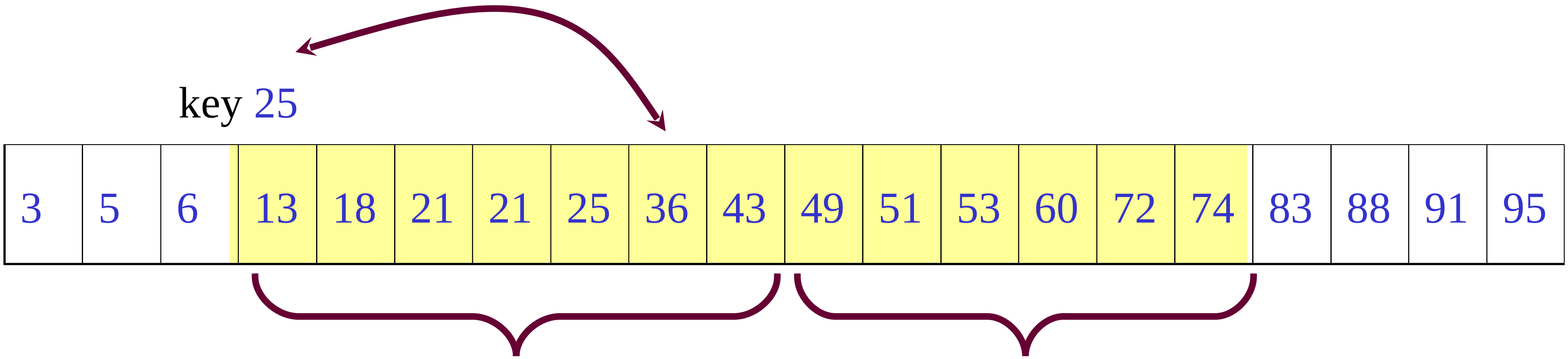
•Keep that half.

Caveat:

Invariant must not assume that the element is present in the list. So it should say something like

“If the key is contained in the original list, then the key is contained in the sublist.”

Define an iteration of loop – contd.



key 25

3	5	6	13	18	21	21	25	36	43	49	51	53	60	72	74	83	88	91	95

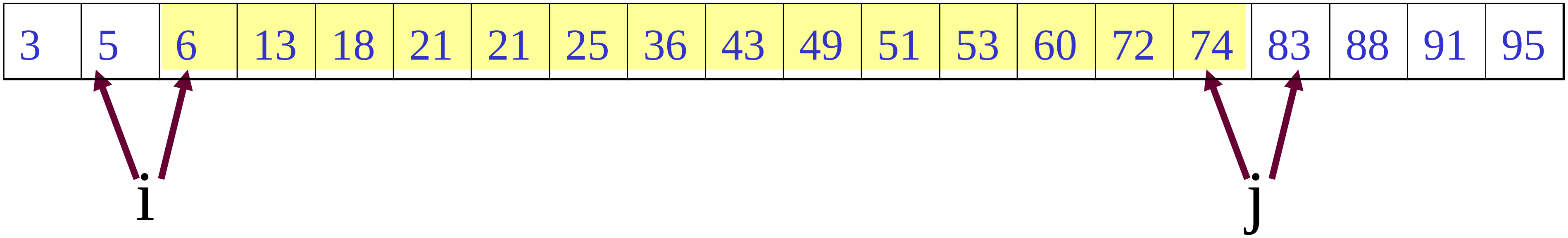
If key ≤ mid,
then key is
in
left half.

If key > mid,
then key is in
right half.

It is faster not to check if the middle element is the key.

The devil is in the details...

- Maintain a sublist with end points i & j



3	5	6	13	18	21	21	25	36	43	49	51	53	60	72	74	83	88	91	95

Does not matter which, but you need to be consistent.

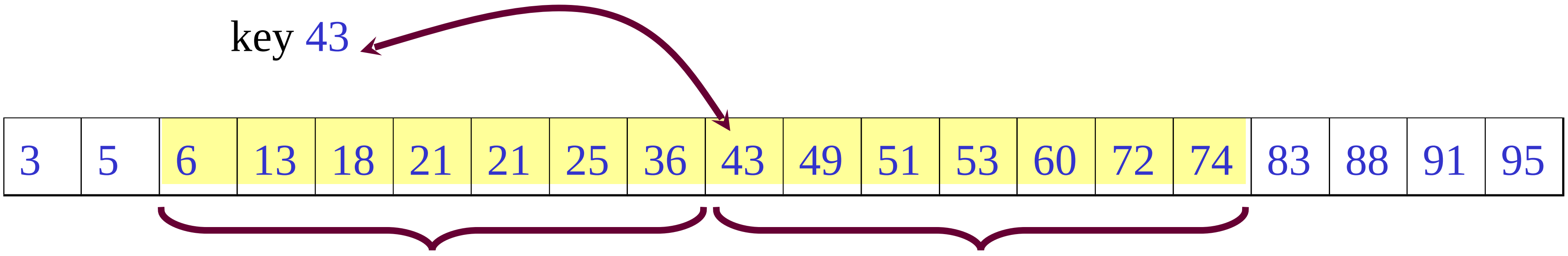
•If the sublist has even length, which element is taken to be mid?

Does not matter – choose **right**.

An easy mistake...

If key ≤ mid, then
key is in left half:
[i,mid-1].

If key > mid, then
key is in right half:
[mid,j]



key 43

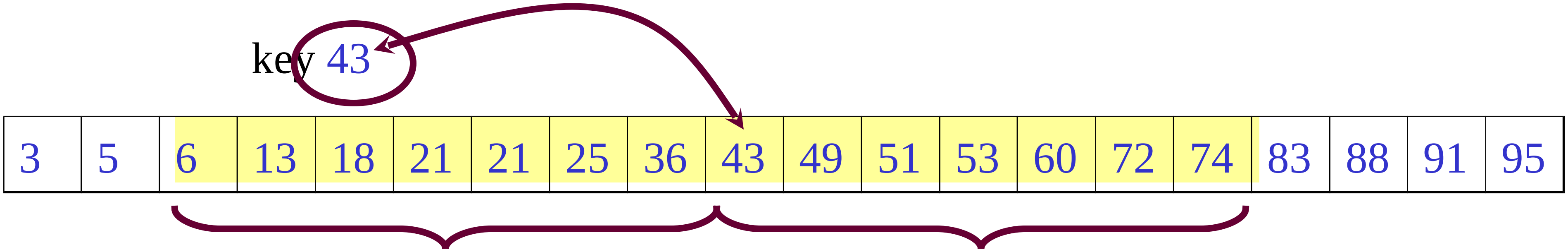
3	5	6	13	18	21	21	25	36	43	49	51	53	60	72	74	83	88	91	95

If the middle element is the key, it can be skipped over!

A fix…

If key ≤ mid,
then key is in
left half: [i,mid-1].

If key ≥ mid,
then key is in
right half: [mid,j].



key 43

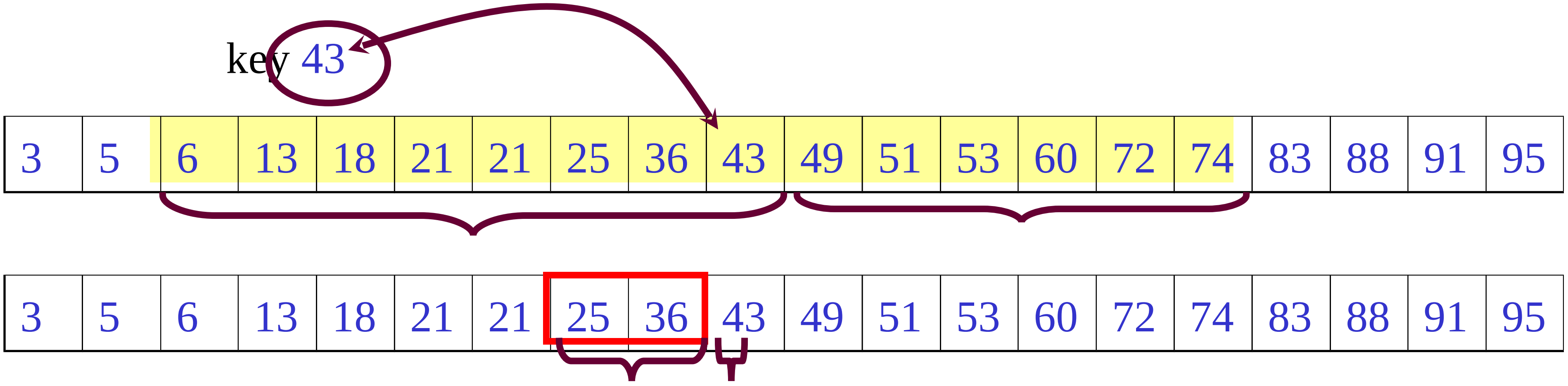
3	5	6	13	18	21	21	25	36	43	49	51	53	60	72	74	83	88	91	95

Another possible fix…

- making the left half slightly bigger.

If key ≤ mid, then key is in left half: [i,mid].

If key > mid, then key is in right half: [mid+1,j].



3	5	6	13	18	21	21	25	36	43	49	51	53	60	72	74	83	88	91	95

3	5	6	13	18	21	21	25	36	43	49	51	53	60	72	74	83	88	91	95

No progress is made. Loop for ever!

Lessons to be learnt

- Use the loop invariant method to think about algorithms.
- Be careful with your definitions.
- Be sure that the loop invariant is always maintained.
- Be sure progress is always made.

Running time of binary search

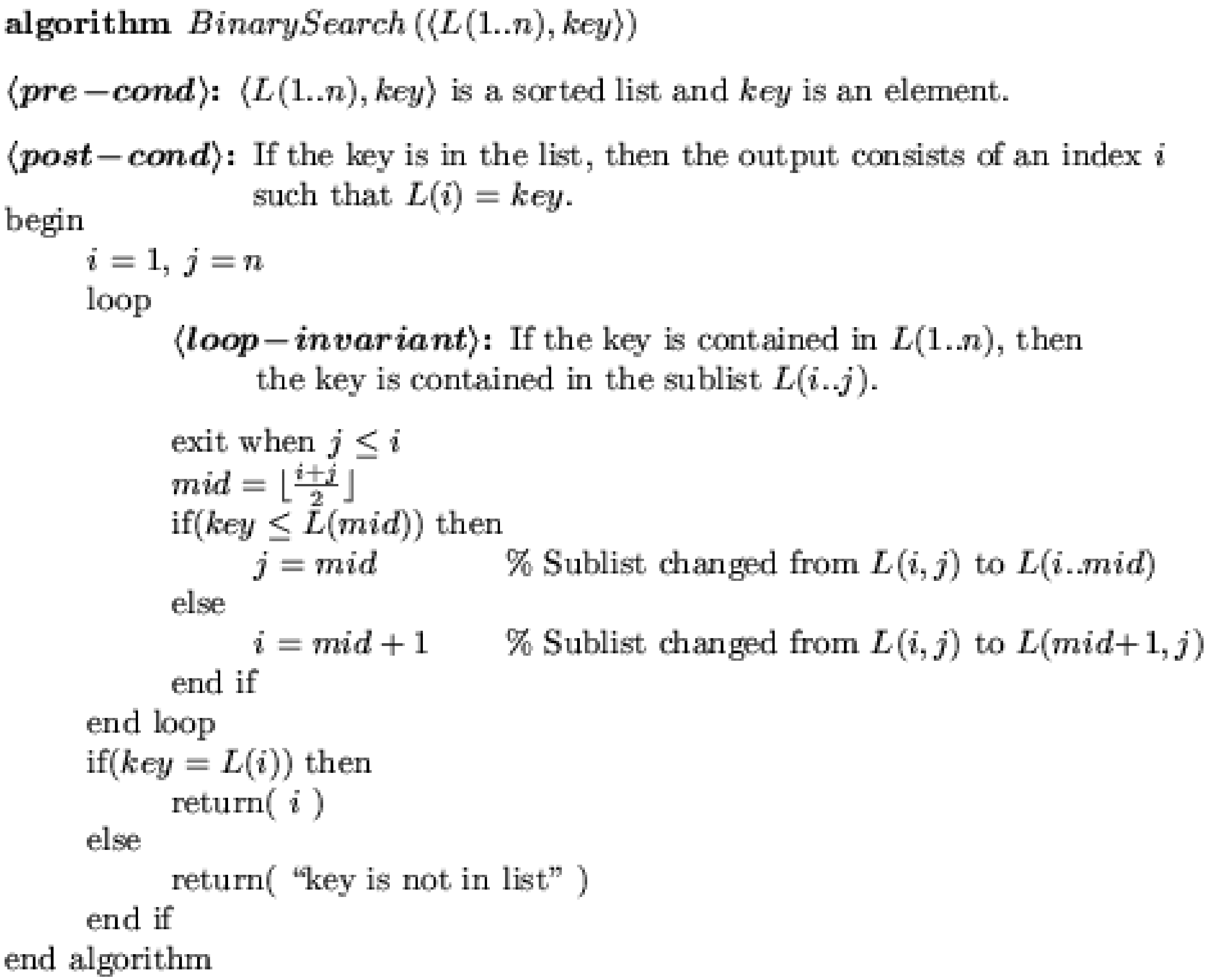
From now, we will omit details about accounting for running time as follows. The details are tedious but can be supplied easily. We will also ignore floors and ceilings. This usually makes no difference.

The sublist is of size n, ${}^{n}/_{2}$, ${}^{n}/_{4}$, ${}^{n}/_{8}$,…,1. How many steps is that?

Each step takes $\theta(1)$ time.

Total running time = $\theta(\log n)$

Pseudocode for binary search



```
algorithm BinarySearch (⟨L(1..n), key⟩)

⟨pre−cond⟩: ⟨L(1..n), key⟩ is a sorted list and key is an element.

⟨post−cond⟩: If the key is in the list, then the output consists of an index i
             such that L(i) = key.
begin
    i = 1, j = n
    loop
        ⟨loop−invariant⟩: If the key is contained in L(1..n), then
            the key is contained in the sublist L(i..j).

        exit when j ≤ i
        mid = ⌊(i+j)/2⌋
        if(key ≤ L(mid)) then
            j = mid            % Sublist changed from L(i, j) to L(i..mid)
        else
            i = mid + 1        % Sublist changed from L(i, j) to L(mid+1, j)
        end if
    end loop
    if(key = L(i)) then
        return( i )
    else
        return( "key is not in list" )
    end if
end algorithm
```