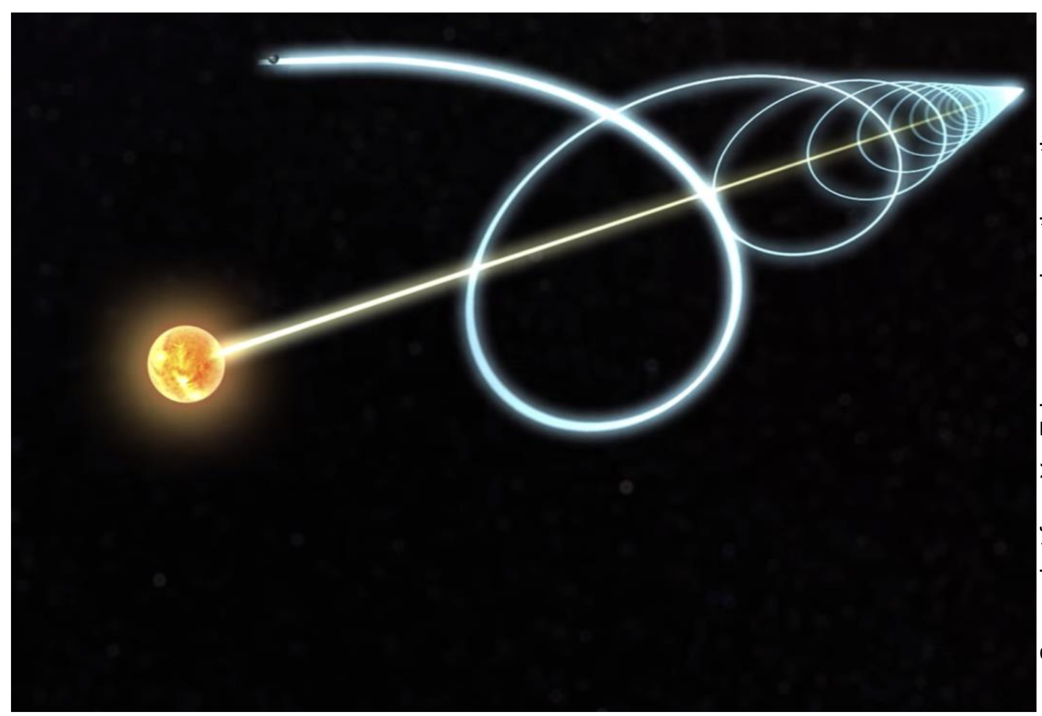
FIT1045: Algorithms and Programming Fundamentals in Python

Lecture 10
Invariants



Screenshot from YouTube: vsauce, how the earth moves https://www.youtube.com/watch?v=IJhgZBn-LHg

Acknowledgment: Some of the slides are prepared by staff at Monash College

Objectives

- Formulate assertions about program states
- Demonstrate that truth of certain assertions is unchanged (invariant) by program (specifically by a loop)
- Relate invariants to computational problem to demonstrate correctness of algorithm

Covered learning outcomes:

3 – Analyse the behaviour of programs and data structures

Concrete goal:

convince ourselves that Prim's algorithm is correct

Programs with simple flow are easy to recognise as correct

```
def number_of_days(month, year):
    if month == 2:
        if is_leap_year(year):
            return 29
        else:
            return 28
    elif month in THIRTY_DAYS_MONTH:
        return 30
    else:
        return 31
```

```
def valid_date(day, month, year):
    if month not in VALID_MONTHS:
        return False
    elif day not in range(1, number_of_days(month, year)):
        return False
    else:
        return True
```

…but is this really computing a spanning tree?

```
def spanning_tree(graph):
    """Input : adjacency matrix of graph
       Output: adj. mat. of spanning tree of graph"""
    n = len(graph)
    tree = empty_graph(n)
    conn = {0}
    while len(conn) < n:
        found = False
        for i in conn:
            for j in range(n):
                if j not in conn and graph[i][j]==1:
                    tree[i][j] = 1
                    tree[j][i] = 1
                    conn = conn.add(j)
                    found = True
                    break
            if found:
                break
    return tree
```

Decomposition helps but loops with *re-assignments/mutation* remain tricky

```
def extension(c, g):
    """I: connec. vertices (c), graph (g)
       O: extension edge (i, j)"""
    n = len(g)
    for i in vertices:
        for j in range(n):
            if j not in c and g[i][j]:
                return i, j
```

```
def spanning_tree(graph):
    """input: graph given as adj. matrix
       output: spanning tree of graph"""
    tree = empty_graph(len(graph))
    conn = {0}
    while len(conn) < n:
        i, j = extension(conn, graph)
        tree[i][j], tree[j][i] = 1, 1
        conn = conn.add{j}
return tree
```

values behind names change all the time

Outline

- Assertions and invariants
- Analysing Insertion Sort
- Analysing Min Index Selection
- Analysing Prim’s Algorithm

Cutting the Chocolate Block

A chocolate block is divided into squares by horizontal and vertical grooves. The object is to cut the chocolate block into individual pieces.

Assume each cut is made on a **single piece** along a groove. How many cuts are needed?

How many cuts does it take to divide the following block into squares?

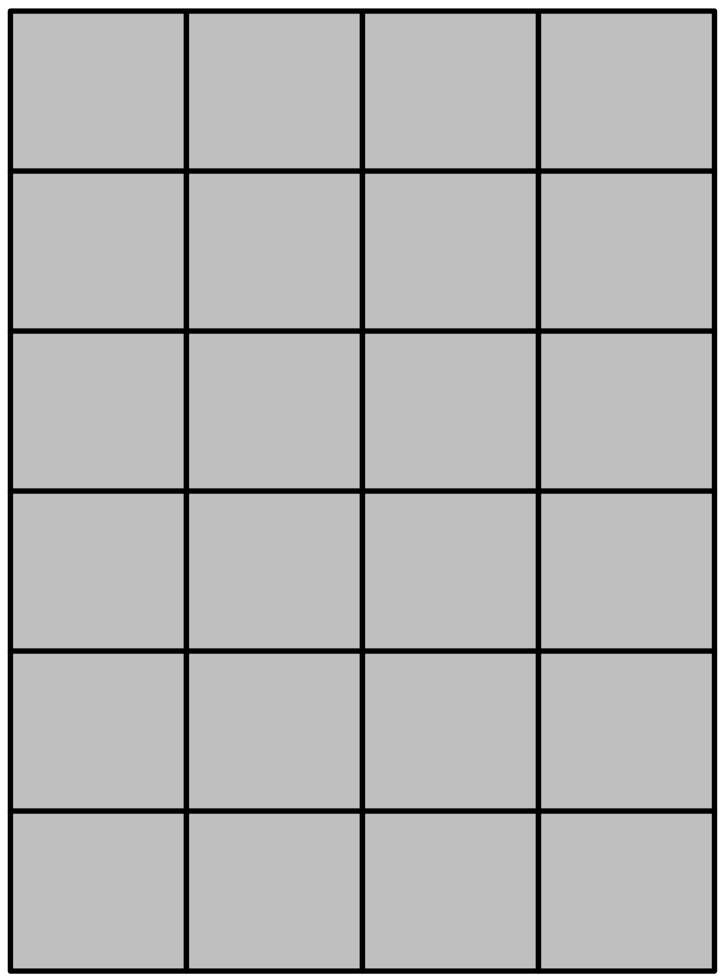
A. 8
B. 3
C. 24
D. 23
E. None of the above

Quiz time (https://flux.qa)
Clayton: AXXULH
Malaysia: LWERDE

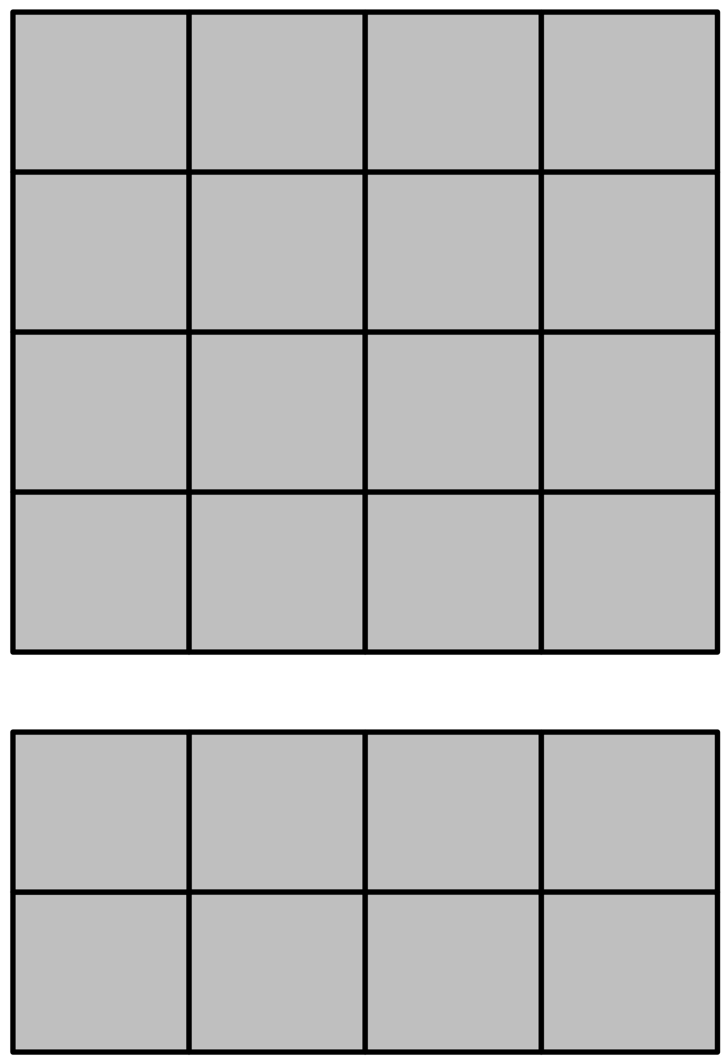
How many cuts does it take to divide a 100 X 50 block of chocolate?

A. 5000
B. 4999
C. 4900
D. 4950
E. None of the above

What is the relationship between cuts and number of pieces?

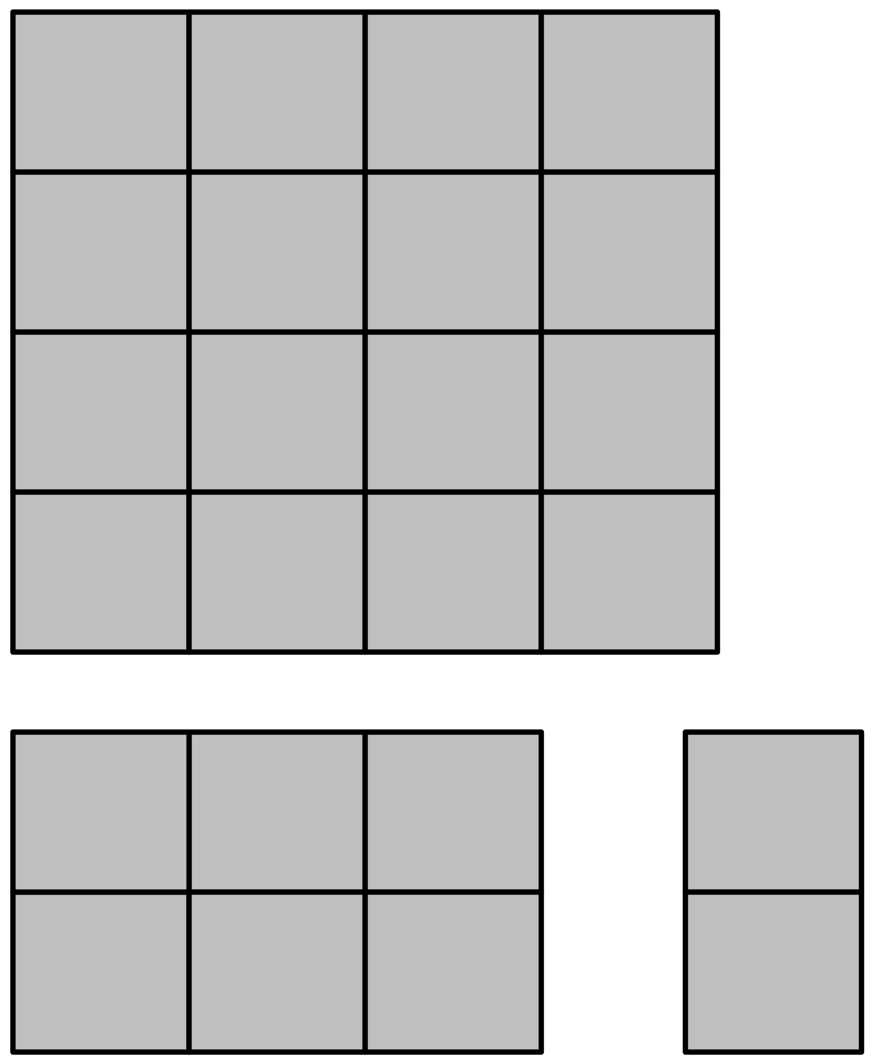


What is the relationship between cuts and number of pieces?



1 cut
2 pieces

What is the relationship between cuts and number of pieces?



2 cuts
3 pieces

Statement
"number of pieces equals number of cuts plus one"
…holds throughout cutting process

Let's bring this concept into the world of programs

```
cut = … #some arbitrary cutting strategy
pieces = [(m, n)]
num_cuts = 0
while len(pieces)<n*m:
    cut(pieces)
    num_cuts += 1
```

Example cutting strategy
(we know it doesn't matter)

```
def cut_first_possible(pieces):
    for i in range(pieces):
        m, n = pieces[i]
        m, n = max(m,n), min(m,n)
        if m > 1:
            pieces.pop(i)
            pieces.append[(m-1,n), (1,n)]
            break
```

Let’s analyse this program by stating *assertions*

```
cut = … #some arbitrary cutting strategy
pieces = [(m, n)]
num_cuts = 0
while len(pieces)<n*m:
    cut(pieces)
    num_cuts += 1
```

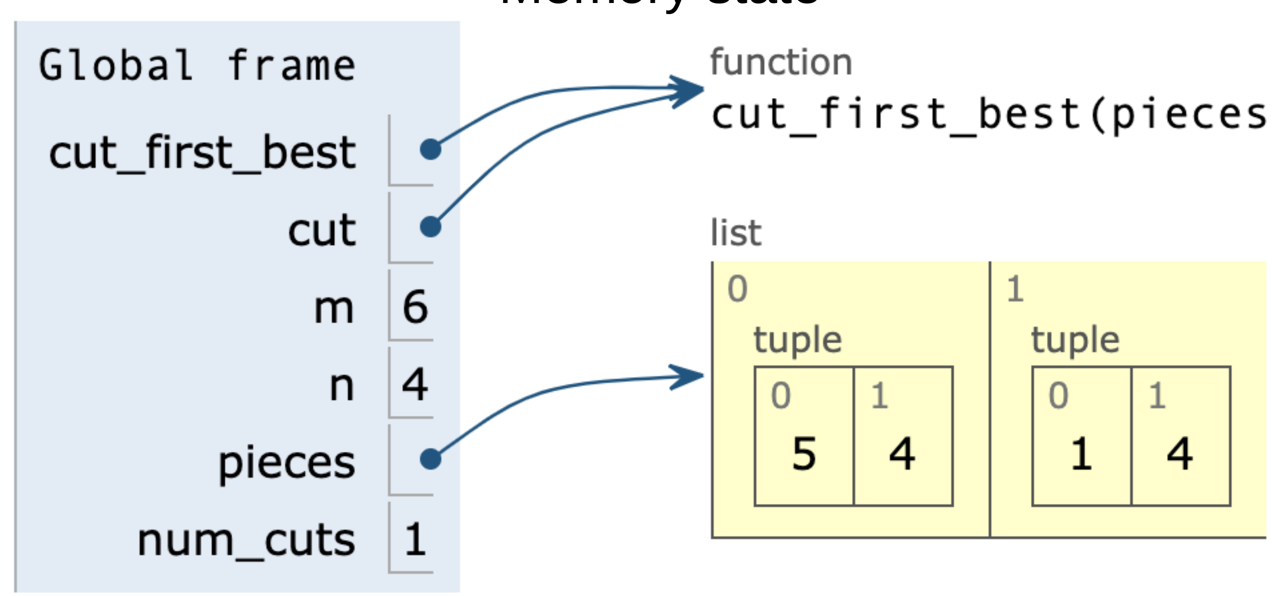
https://goo.gl/Mkvzjm

An **assertion** is a logical statement on a *program (execution) state*.

Instruction pointer

```
cut = cut_first_best
m, n = 6, 4
pieces = [(m, n)]
num_cuts = 0
while len(pieces)<n*m:
    cut(pieces)
    num_cuts += 1
```

Memory state



Let's analyse this program by stating *assertions*

```
cut = … #some arbitrary cutting strategy
pieces = [(m, n)]
num_cuts = 0
#PRC: len(pieces)==num_cuts+1
while len(pieces)<n*m:
    cut(pieces)
    num_cuts += 1
```

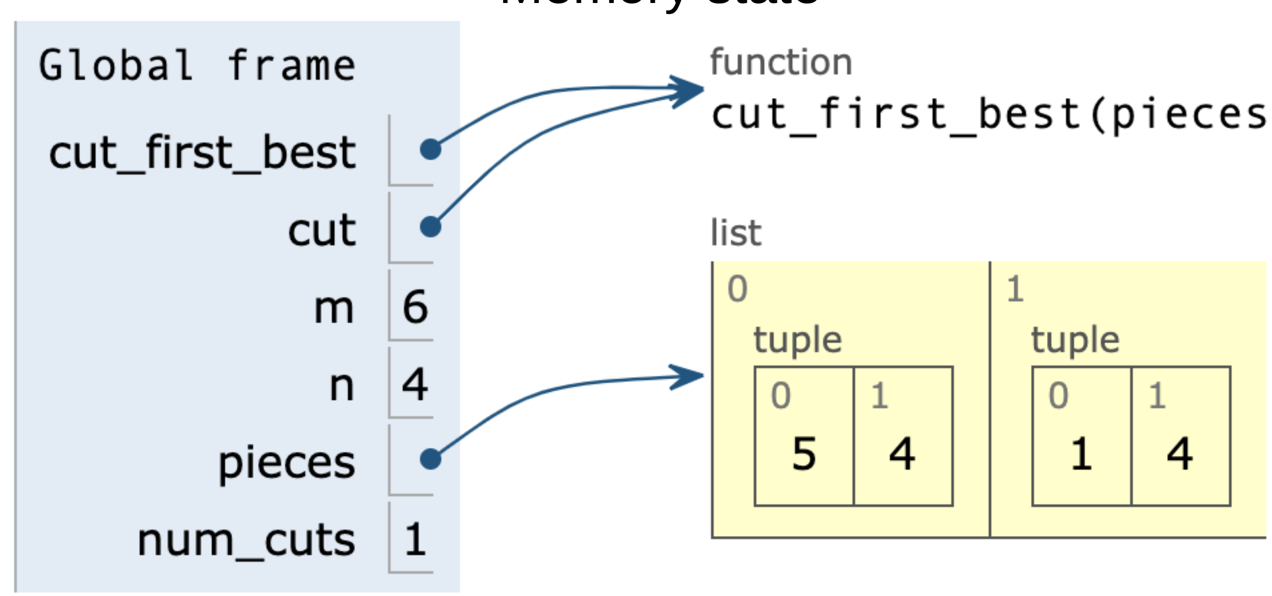
Example:
loop precondition

An **assertion** is a logical statement on a *program (execution) state*.

Instruction pointer

```
cut = cut_first_best
m, n = 6, 4
pieces = [(m, n)]
num_cuts = 0
while len(pieces)<n*m:
    cut(pieces)
    num_cuts += 1
```

Memory state



What happens during the loop?

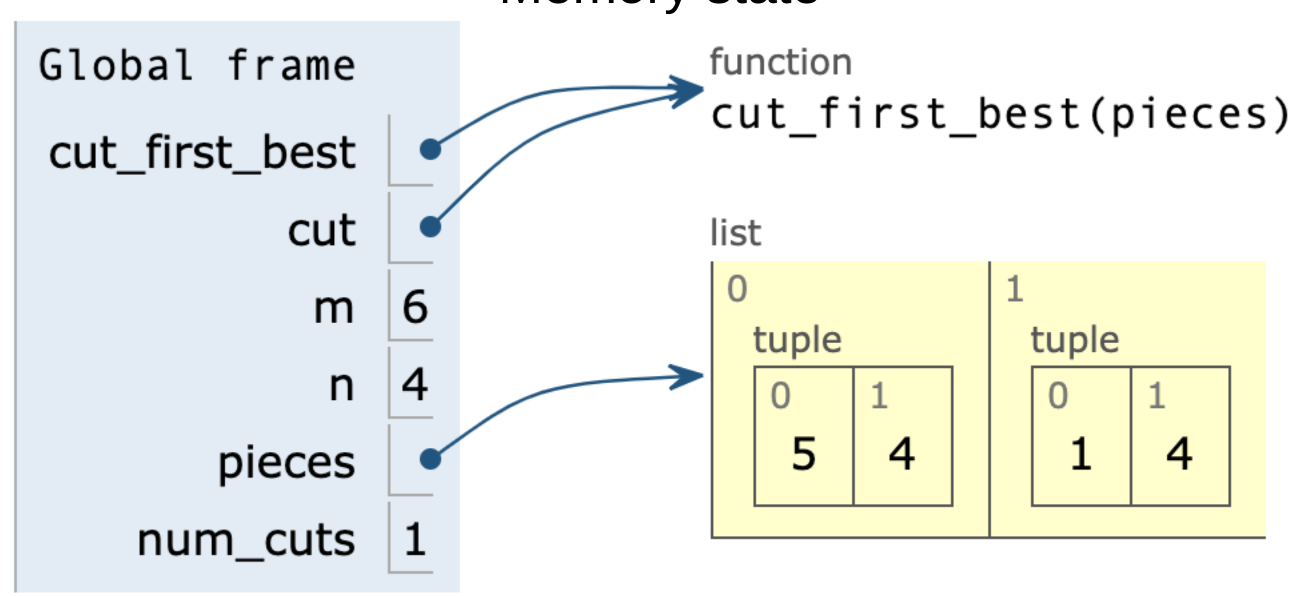
```
cut = … #some arbitrary cutting strategy
pieces = [(m, n)]
num_cuts = 0
#PRC: len(pieces)==num_cuts+1
while len(pieces)<n*m:
    #len(pieces)==num_cuts+1
    cut(pieces)
    num_cuts += 1
```

An **assertion** is a logical statement on a *program (execution) state*.

Instruction pointer

```
cut = cut_first_best
m, n = 6, 4
pieces = [(m, n)]
num_cuts = 0
while len(pieces)<n*m:
    cut(pieces)
    num_cuts += 1
```

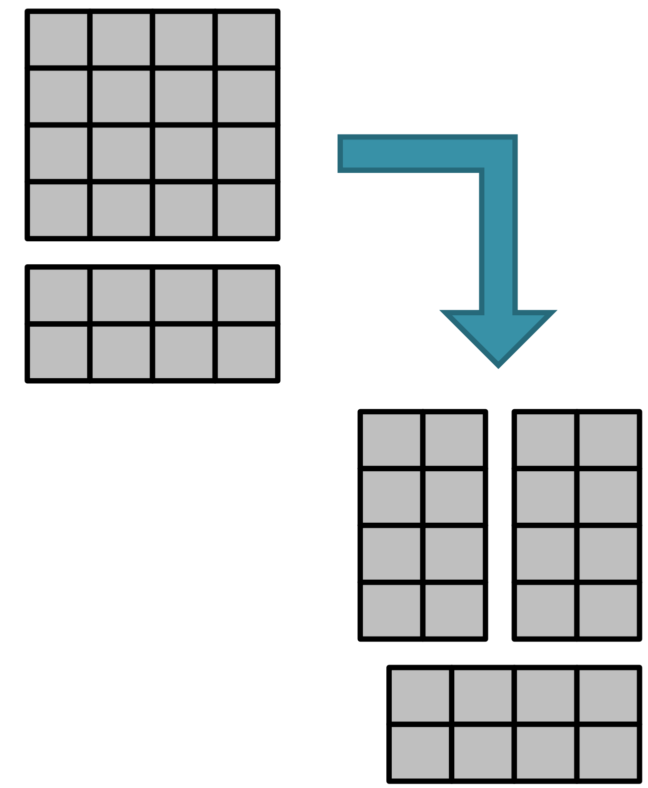
Memory state



What happens during the loop?

```
cut = … #some arbitrary cutting strategy
pieces = [(m, n)]
num_cuts = 0
#PRC: len(pieces)==num_cuts+1
while len(pieces)<n*m:
    #len(pieces)==num_cuts+1
    cut(pieces)
    num_cuts += 1
```

after this step, assertion is (temporarily) violated

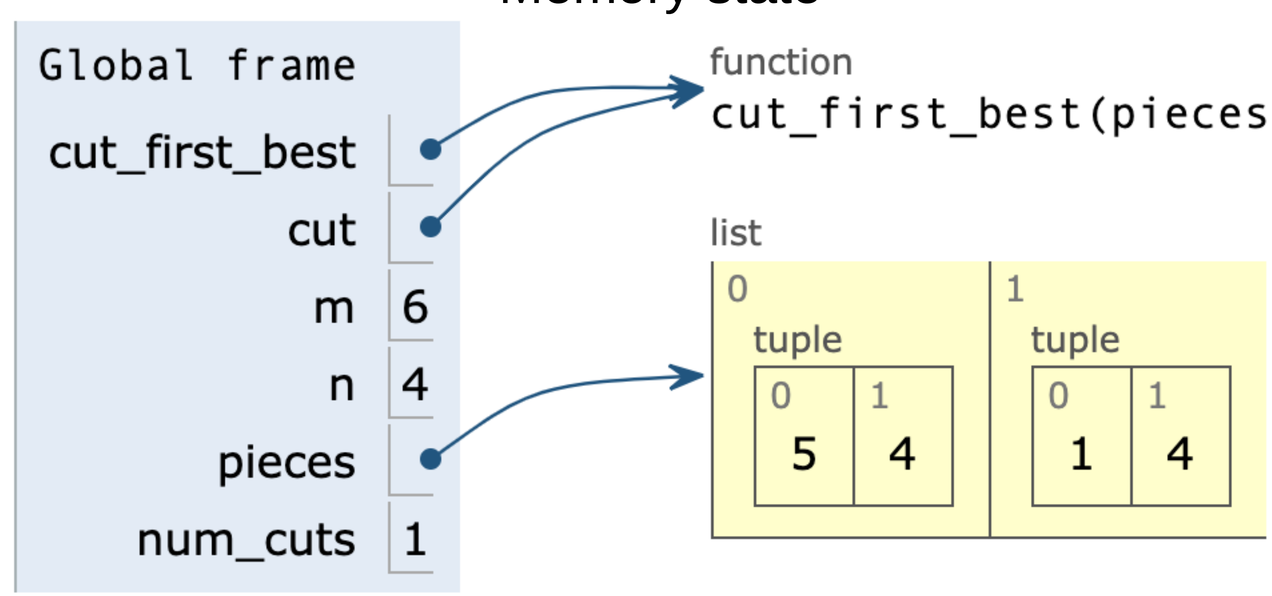


An **assertion** is a logical statement on a *program (execution) state.*

Instruction pointer

```
cut = cut_first_best
m, n = 6, 4
pieces = [(m, n)]
num_cuts = 0
while len(pieces)<n*m:
    cut(pieces)
    num_cuts += 1
```

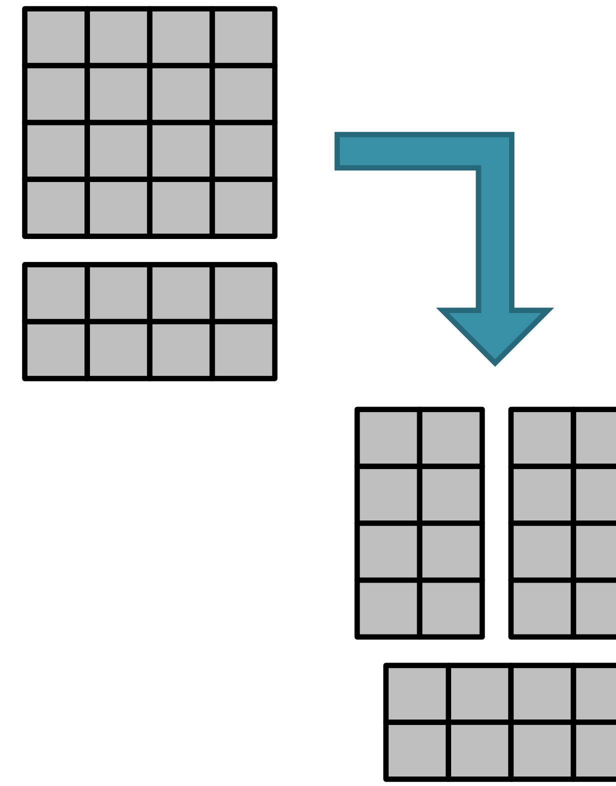
Memory state



What happens during the loop?

```
cut = … #some arbitrary cutting strategy
pieces = [(m, n)]
num_cuts = 0
#PRC: len(pieces)==num_cuts+1
while len(pieces)<n*m:
    #len(pieces)==num_cuts+1
    cut(pieces)
    num_cuts += 1
```

after this step it is restored

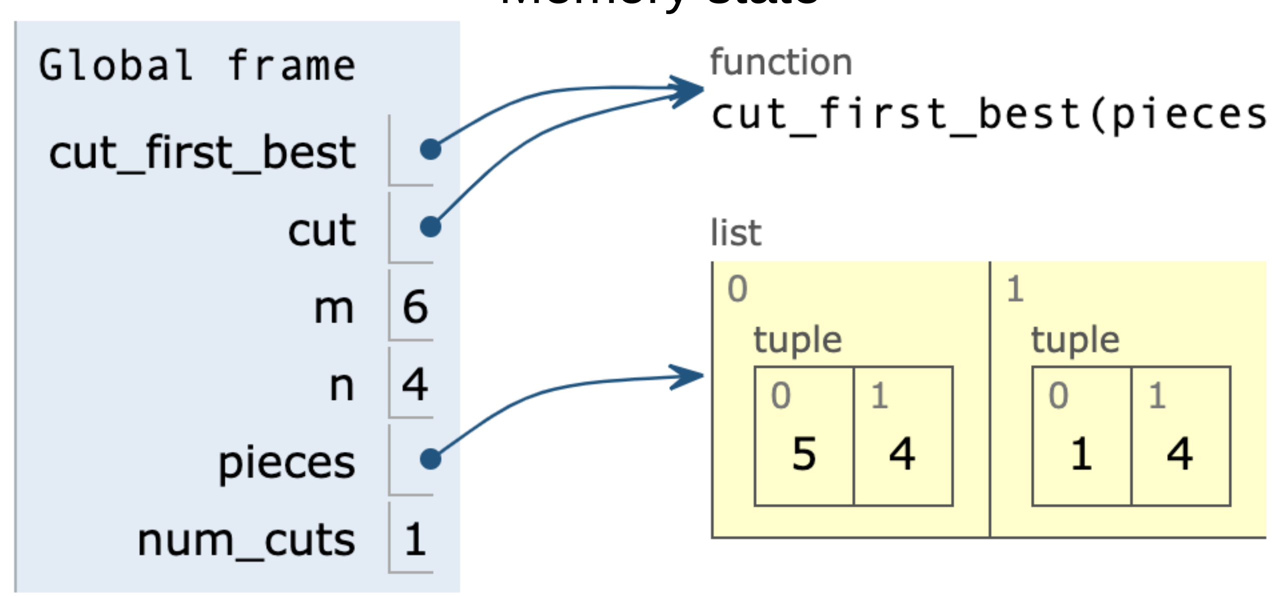


An **assertion** is a logical statement on a *program (execution) state.*

Instruction pointer

```
cut = cut_first_best
m, n = 6, 4
pieces = [(m, n)]
num_cuts = 0
while len(pieces)<n*m:
    cut(pieces)
    num_cuts += 1
```

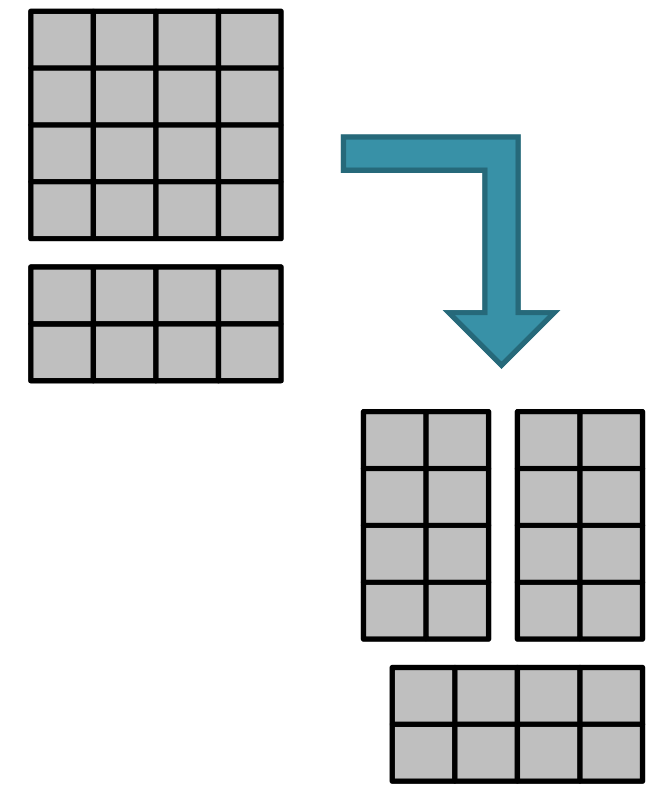
Memory state



What happens during the loop?

```python
cut = … #some arbitrary cutting strategy
pieces = [(m, n)]
num_cuts = 0
#PRC: len(pieces)==num_cuts+1
while len(pieces)<n*m:
    #len(pieces)==num_cuts+1
    cut(pieces)
    num_cuts += 1
    #len(pieces)==num_cuts+1
```

maintained by loop body

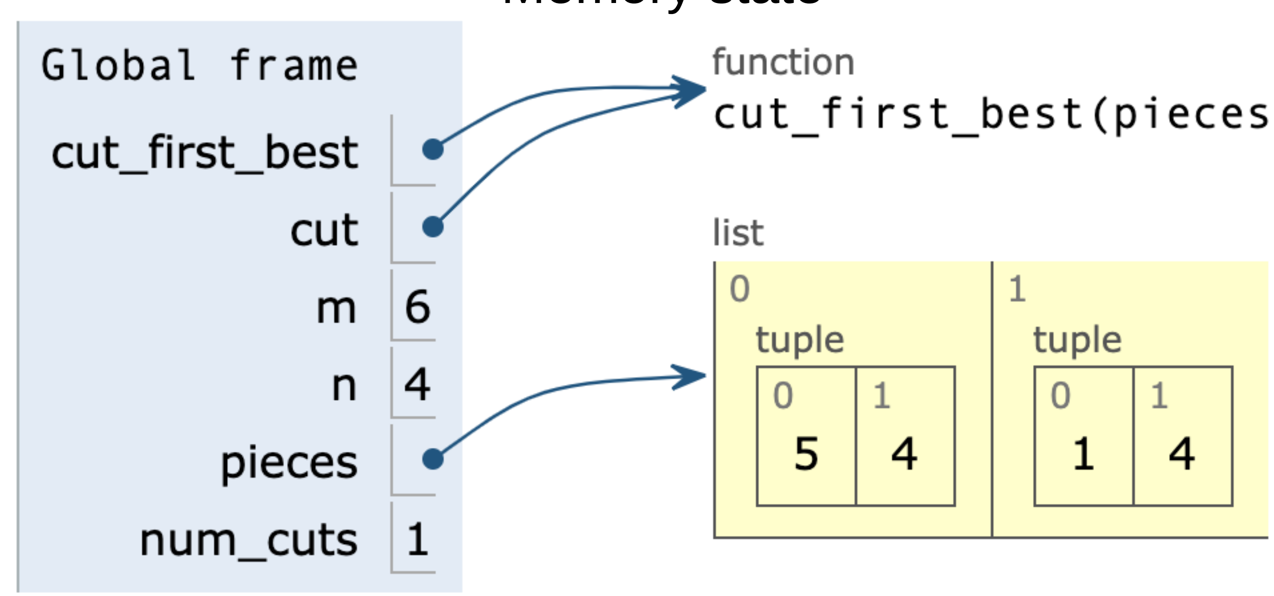


An **assertion** is a logical statement on a *program (execution) state*.

Instruction pointer

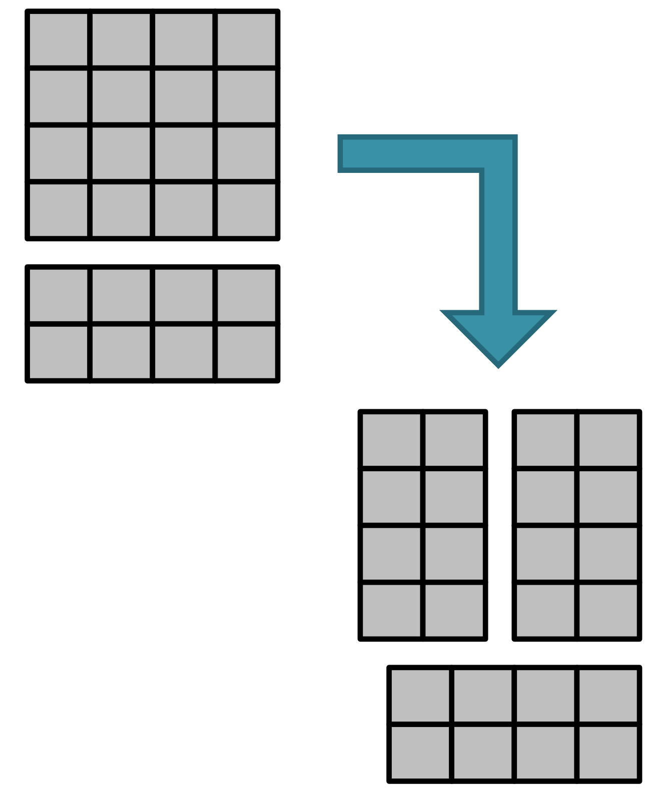
```python
cut = cut_first_best
m, n = 6, 4
pieces = [(m, n)]
num_cuts = 0
while len(pieces)<n*m:
    cut(pieces)
    num_cuts += 1
```

Memory state



Loop invariant is an assertion maintained by loop body

```
cut = … #some arbitrary cutting strategy
pieces = [(m, n)]
num_cuts = 0
#PRC: len(pieces)==num_cuts+1
while len(pieces)<n*m:
    #INV: len(pieces)==num_cuts+1
    cut(pieces)
    num_cuts += 1
    #INV: len(pieces)==num_cuts+1
```



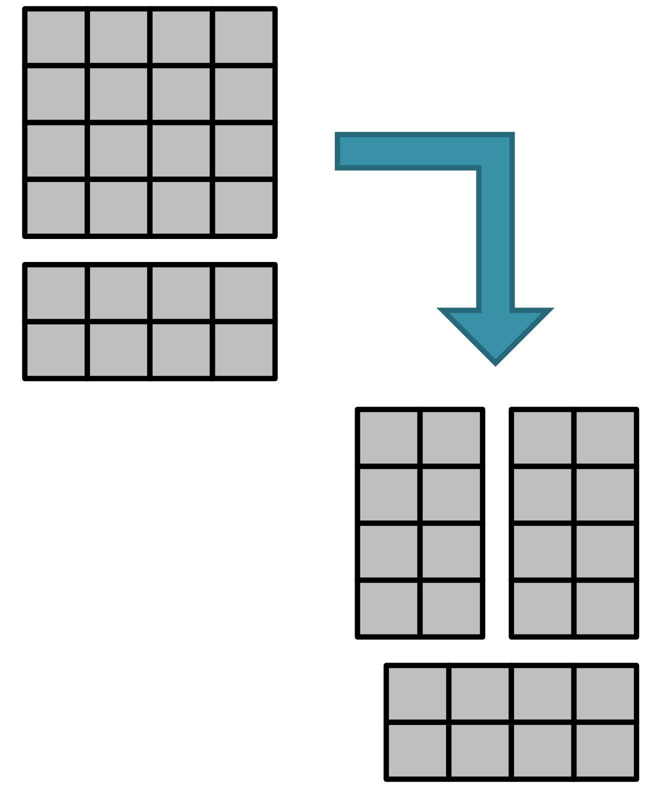
An **assertion** is a logical statement on a *program (execution) state*.

A **loop invariant** is an assertion inside a loop that is true every time it is reached by the program execution.

[Furia et al., 2014: Loop invariants: analysis, classification, and examples]

What are useful loop invariants?

```
cut = … #some arbitrary cutting strategy
pieces = [(m, n)]
num_cuts = 0
#PRC: len(pieces)==num_cuts+1
while len(pieces)<n*m:
    cut(pieces)
    num_cuts += 1
    #INV: len(pieces)==num_cuts+1
```



An **assertion** is a logical statement on a *program (execution) state*.

A **loop invariant** is an assertion inside a loop that is true every time it is reached by the program execution.

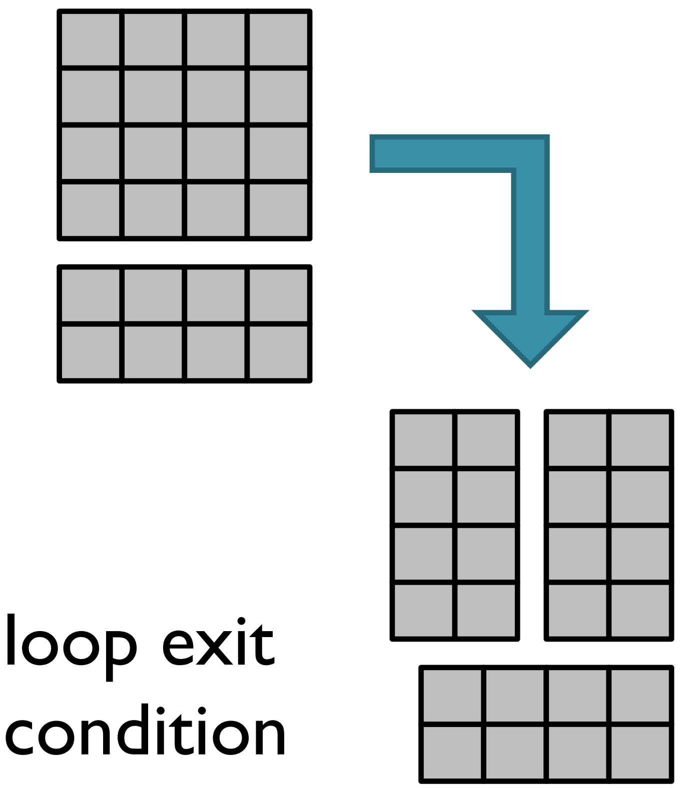
We want invariants at *end of loop* that together with loop exit condition *"turn into"* desired post-condition.

[Furia et al., 2014: Loop invariants: analysis, classification, and examples]

What are useful loop invariants?

```
cut = … #some arbitrary cutting strategy
pieces = [(m, n)]
num_cuts = 0
#PRC: len(pieces)==num_cuts+1
while len(pieces)<n*m:
    cut(pieces)
    num_cuts += 1
    #INV: len(pieces)==num_cuts+1
#EXC: len(pieces) == n*m
```

loop exit condition



An **assertion** is a logical statement on a *program (execution) state*.

A **loop invariant** is an assertion inside a loop that is true every time it is reached by the program execution.

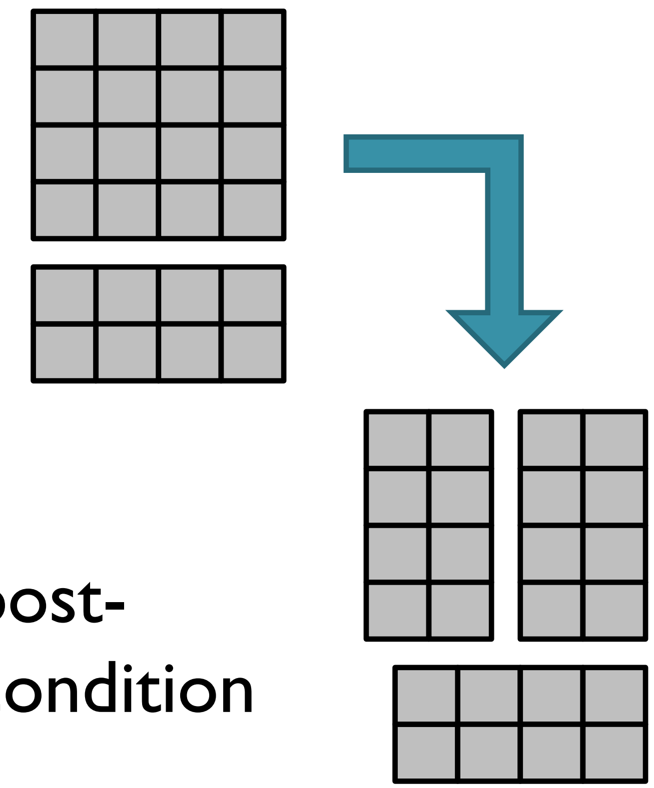
We want invariants at end of loop that together with **loop exit condition** *"turn into"* desired **post-condition**.

[Furia et al., 2014: Loop invariants: analysis, classification, and examples]

What are useful loop invariants?

```
cut = … #some arbitrary cutting strategy
pieces = [(m, n)]
num_cuts = 0
#PRC: len(pieces)==num_cuts+1
while len(pieces)<n*m:
    cut(pieces)
    num_cuts += 1
    #INV: len(pieces)==num_cuts+1
#EXC: len(pieces) == n*m
#POC: num_cuts == n*m - 1
```

post-
condition



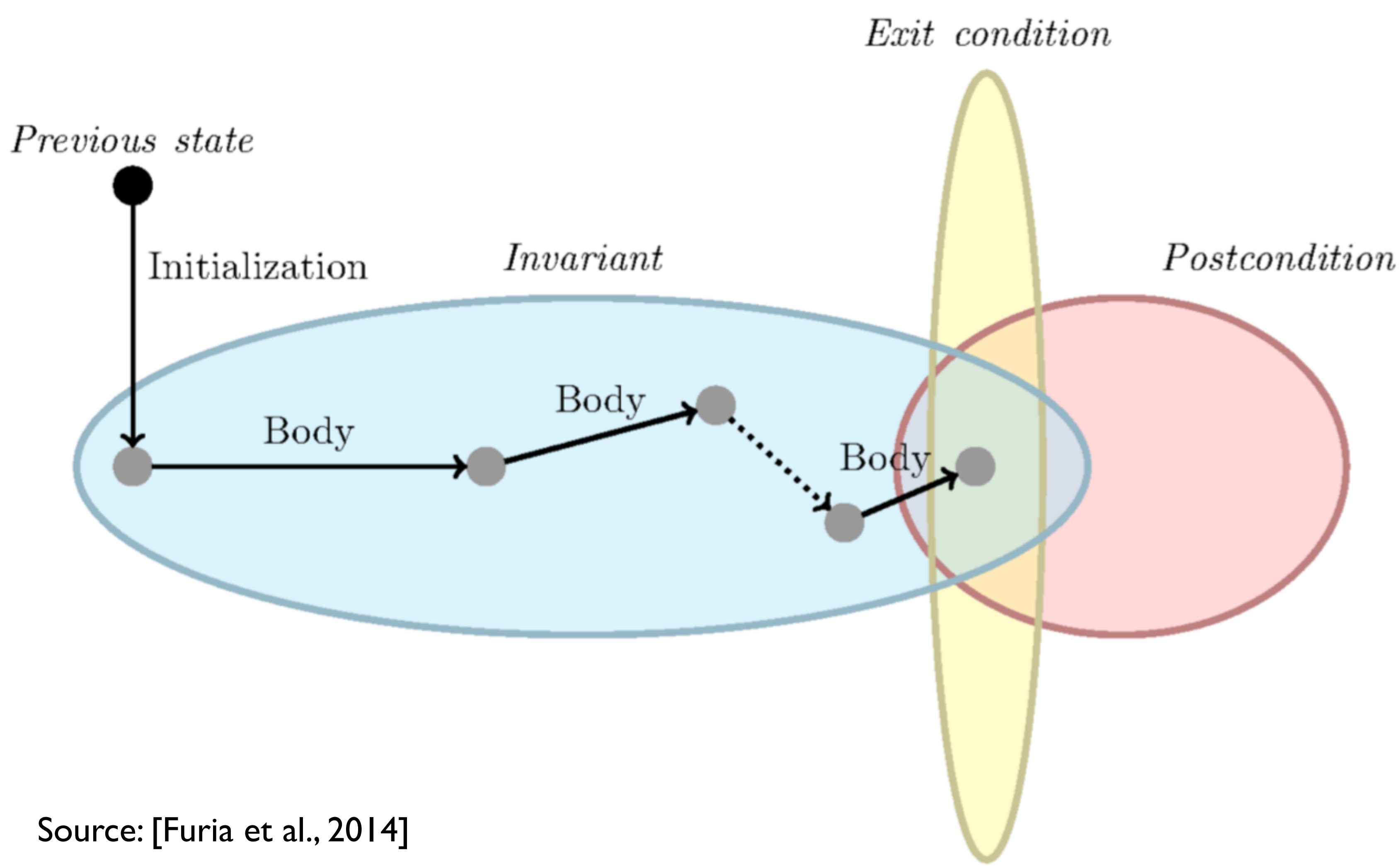
An **assertion** is a logical statement on a *program (execution) state*.

A **loop invariant** is an assertion inside a loop that is true every time it is reached by the program execution.

We want invariants at end of loop that together with **loop exit condition** "*turn into*" desired **post-condition**.

[Furia et al., 2014: Loop invariants: analysis, classification, and examples]

What are useful loop invariants?



Source: [Furia et al., 2014]

We are interested in loop invariants that together with **loop exit condition** *"turn into"* desired **post-condition**.

[Furia et al., 2014: Loop invariants: analysis, classification, and examples]

Outline

- Assertions and invariants
- Analysing Insertion Sort
- Analysing Min Index Selection
- Analysing Prim’s Algorithm

Does Insertion Sort always create sorted list?

```
def insert(i, lst):
    “””accepts: int i and list lst of length n>i>0
                of comp. elements with lst[:i] is sorted
       postcon: lst[:i+1] is sorted”””
    j = i
    while j > 0 and lst[j - 1] > lst[j]:
        lst[j - 1], lst[j] = lst[j], lst[j - 1]
        j = j - 1
```

```
def insertion_sort(lst):
    “””accepts: list lst of length n of comp. elements
       postcon: lst has same elements as on call but
                is sorted                                ”””
    for i in range(1, len(lst)):
        insert(i, lst)
```

Situation at start of execution

lst

?	?	?	?	…													…	?
0	1	2	3	…													…	n-1

```
def insertion_sort(lst):
    """accepts: list lst of length n of comp. elements
       postcon: lst has same elements as on call but
                is sorted                                   """
    for i in range(1, len(lst)):
        insert(i, lst)
```

Loop initialisation

lst

?	?	?	?	…													…	?
0	i=1	2	3	…													…	n-1

```
def insertion_sort(lst):
    """accepts: list lst of length n of comp. elements
       postcon: lst has same elements as on call but
                is sorted                                  """
    for i in range(1, len(lst)):
        insert(i, lst)
```

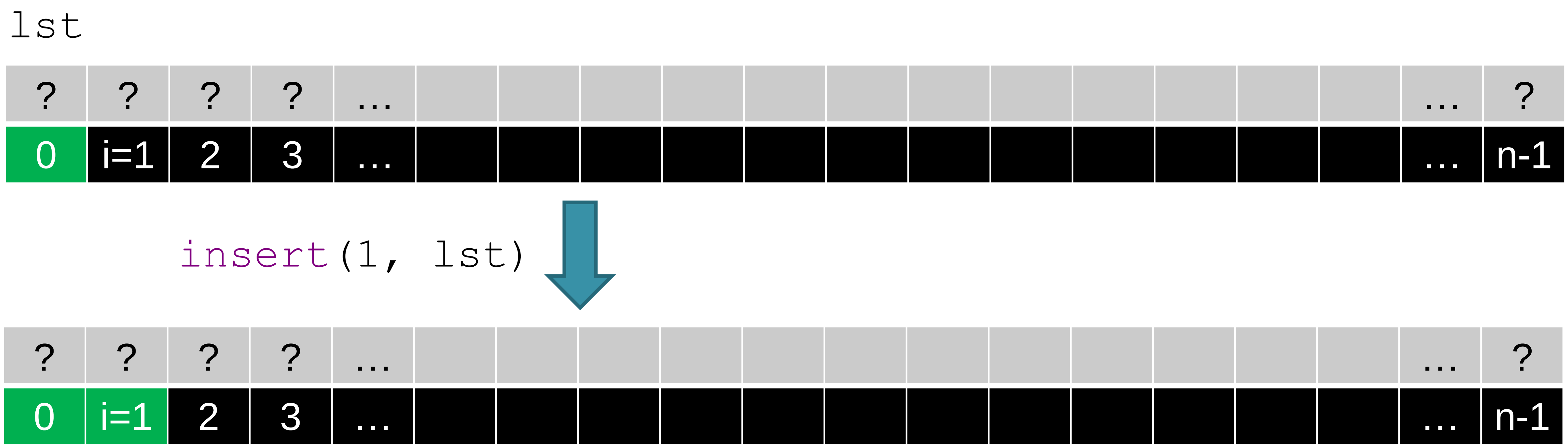
What is true at this point?

lst

?	?	?	?	…													…	?
0	i=1	2	3	…													…	n-1

```
def insertion_sort(lst):
    “””accepts: list lst of length n of comp. elements
       postcon: lst has same elements as on call but
                is sorted                                   ”””
    for i in range(1, len(lst)):
        # lst[:1] is sorted
        insert(i, lst)
```

Insertion procedure extends sorted range by one




```
def insertion_sort(lst):
    """accepts: list lst of length n of comp. elements
       postcon: lst has same elements as on call but
                is sorted                                """
    for i in range(1, len(lst)):
        # lst[:1] is sorted
        insert(i, lst)
```

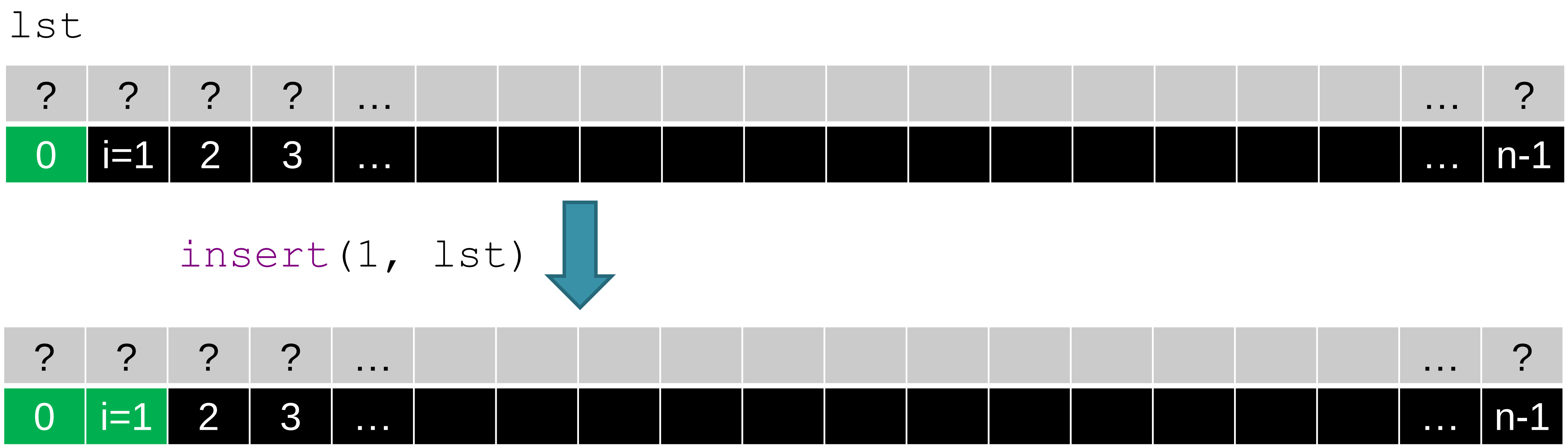
Insertion procedure extends sorted range by one


```
def insertion_sort(lst):
    """accepts: list lst of length n of comp. elements
       postcon: lst has same elements as on call but
                is sorted                            """
    for i in range(1, len(lst)):
        # lst[:1] is sorted
        insert(i, lst)
        # lst[:2] is sorted
```

Hold in first iteration (and further), but not enough do demonstrate post condition

Idea: generalise assertions so that they become stronger every iteration!

These general assertions seem much more useful




```
def insertion_sort(lst):
    """accepts: list lst of length n of comp. elements
       postcon: lst has same elements as on call but
                is sorted                                  """
    for i in range(1, len(lst)):
        # lst[:i] is sorted
        insert(i, lst)
        # lst[:i+1] is sorted
```

But are they preserved by general loop iteration?

lst

?	?	?	?	…													…	?
0	i=1	2	3	…													…	n-1

```
def insertion_sort(lst):
    """accepts: list lst of length n of comp. elements
       postcon: lst has same elements as on call but
                is sorted                                    """
    for i in range(1, len(lst)):
        # lst[:i] sorted
        insert(i, lst)
        # lst[:i+1] sorted
```

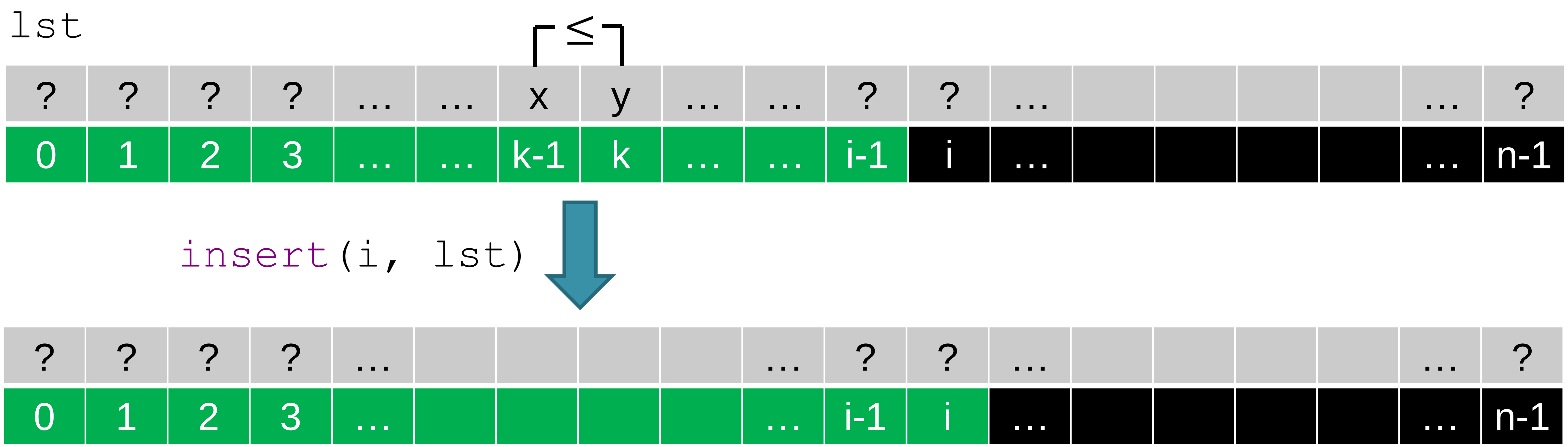
Let's assume first assertion is true

lst

?	?	?	?	…					…	?	?	…					…	?
0	1	2	3	…					…	i-1	i	…					…	n-1

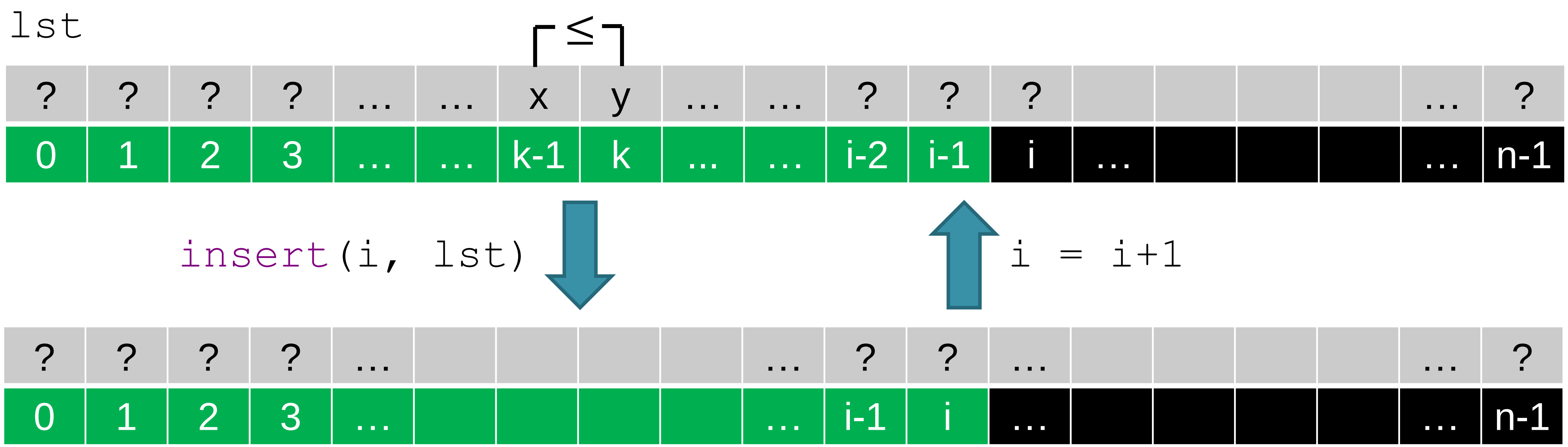
```
def insertion_sort(lst):
    """accepts: list lst of length n of comp. elements
       postcon: lst has same elements as on call but
                is sorted                                  """
    for i in range(1, len(lst)):
        # lst[:i] sorted
        insert(i, lst)
        # lst[:i+1] sorted
```

Then loop body ensures second assertion



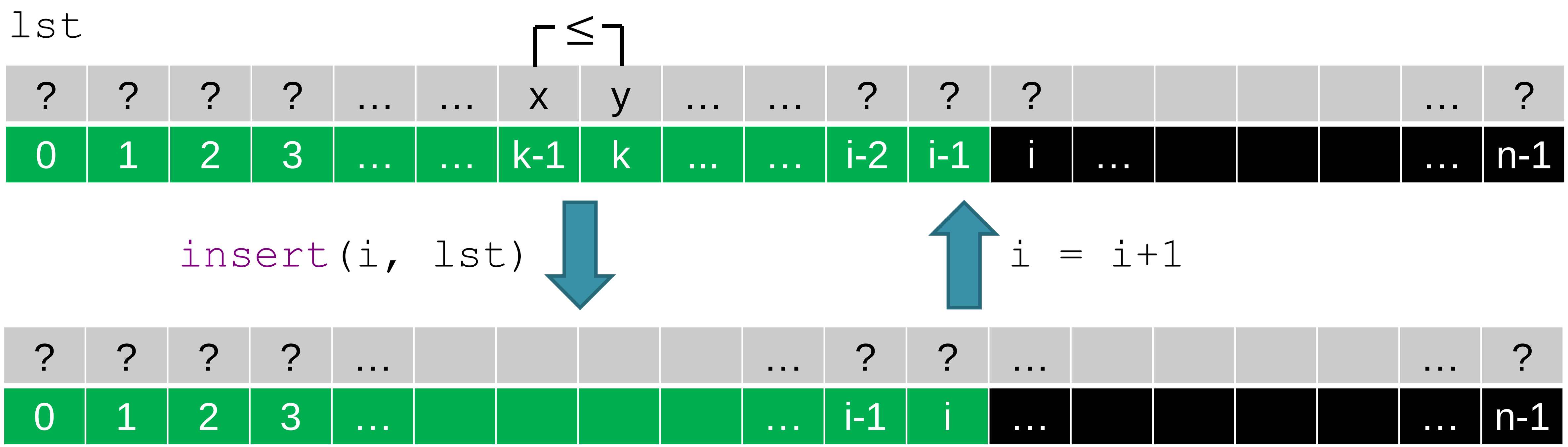

```
def insertion_sort(lst):
    """accepts: list lst of length n of comp. elements
       postcon: lst has same elements as on call but
                is sorted                                """
    for i in range(1, len(lst)):
        # lst[:i] sorted
        insert(i, lst)
        # lst[:i+1] sorted
```

Which in turn implies first assertion in next iteration!



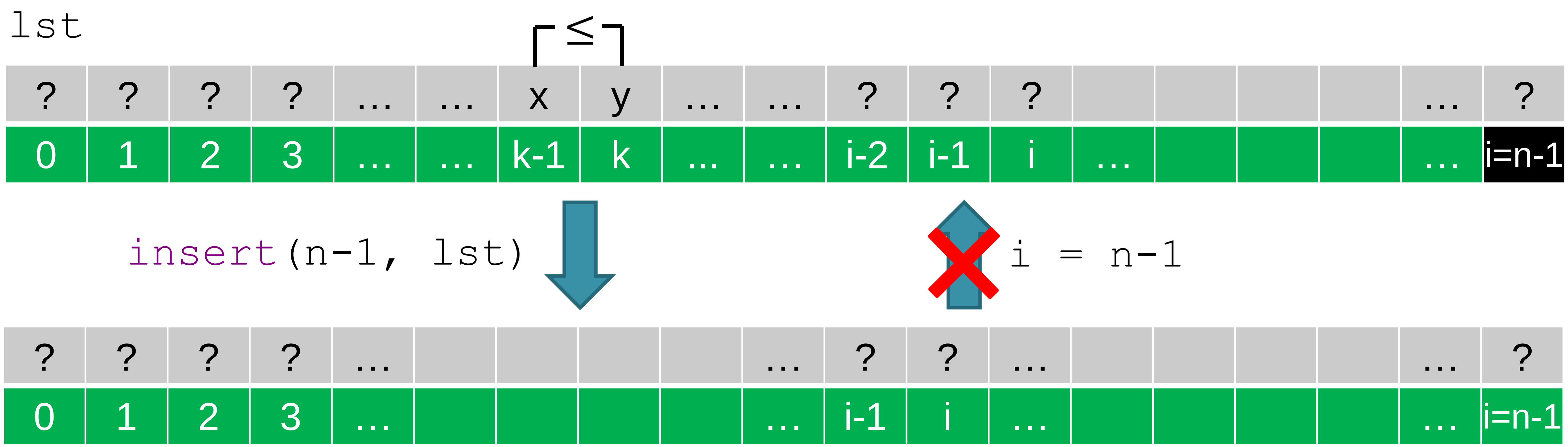

```
def insertion_sort(lst):
    """accepts: list lst of length n of comp. elements
       postcon: lst has same elements as on call but
                is sorted                                """
    for i in range(1, len(lst)):
        # lst[:i] sorted
        insert(i, lst)
        # lst[:i+1] sorted
```

Thus these assertions are loop invariants!



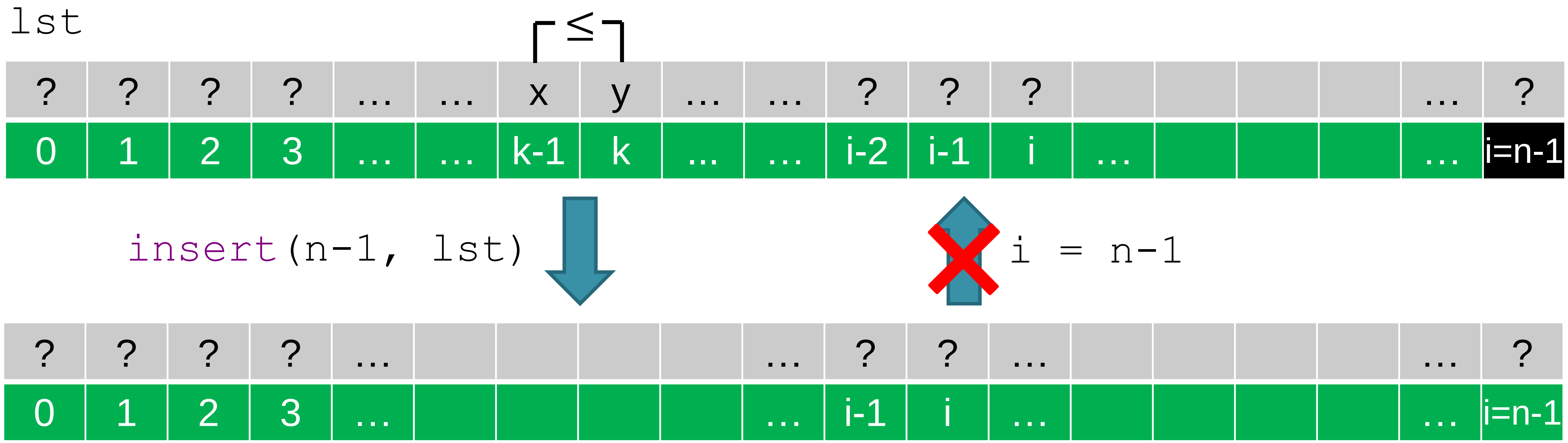

```
def insertion_sort(lst):
    """accepts: list lst of length n of comp. elements
       postcon: lst has same elements as on call but
                is sorted                           """
    for i in range(1, len(lst)):
        #I: lst[:i] sorted
        insert(i, lst)
        #I': lst[:i+1] sorted
```

What happens at end of loop?




```
def insertion_sort(lst):
    """accepts: list lst of length n of comp. elements
       postcon: lst has same elements as on call but
                is sorted                                """
    for i in range(1, len(lst)):
        #I: lst[i] sorted
        insert(i, lst)
        #I': lst[i+1] sorted
    #EXC: i = n-1
```

Loop exit condition and invariant imply desired post condition




```
def insertion_sort(lst):
    """accepts: list lst of length n of comp. elements
       postcon: lst has same elements as on call but
                is sorted                                       """
    for i in range(1, len(lst)):
        #I: lst[i] sorted
        insert(i, lst)
        #I': lst[i+1] sorted
    #EXC: i = n-1
    #POC: lst[:n] sorted
```

Outline

- Assertions and invariants
- Analysing Insertion Sort
- Analysing Min Index Selection
- Analysing Prim's Algorithm

Recap: what is min index trying to do (formally)?

```
def min_index(lst):
    """accepts: list of length n>0 of comp. elements
       returns: ?
                                                    """
    k = 0
    for i in range(1, len(lst)):
        if lst[i] < lst[k]:
            k = i
    return k
```

Quiz time (https://flux.qa)

Clayton: AXXULH

Malaysia: LWERDE

Recap: what is min index trying to do (formally)?

```
def min_index(lst):
    """accepts: list of length n>0 of comp. elements
       returns: index k in range(n) such that
                for all j in range(n), lst[k]<=lst[j] """
    k = 0
    for i in range(1, len(lst)):
        if lst[i] < lst[k]:
            k = i
    return k
```

lst

≥ ≤

?	?	?	…	y’	…	x	…	y	…			…					…	?
0	1	2	…	j’	…	k	…	j	…			…					…	n-1

Does min index function always yield index of minimum value?

```
def min_index(lst):
    """accepts: list of length n>0 of comp. elements
       returns: index k in range(n) such that
                for all j in range(n), lst[k]<=lst[j]"""
    k = 0
    for i in range(1, len(lst)):
        if lst[i] < lst[k]:
            k = i
    return k
```

lst

?	?	?	…														…	?
0	1	2	…														…	n-1

Situation before reaching loop statement

```
def min_index(lst):
    “””accepts: list of length n>0 of comp. elements
       returns: index k in range(n) such that
                for all j in range(n), lst[k]<=lst[j]”””
    k = 0
    for i in range(1, len(lst)):
        if lst[i] < lst[k]:
            k = i
    return k
```

lst

?	?	?	…														…	?
k=0	1	2	…														…	n-1

First iteration of loop

```
def min_index(lst):
    """accepts: list of length n>0 of comp. elements
       returns: index k in range(n) such that
                for all j in range(n), lst[k]<=lst[j]"""
    k = 0
    for i in range(1, len(lst)):
        if lst[i] < lst[k]:
            k = i
    return k
```

lst

?	?	?	…														…	?
k=0	i=1	2	…														…	n-1

What is true at this point?

```
def min_index(lst):
    """accepts: list of length n>0 of comp. elements
       returns: index k in range(n) such that
                for all j in range(n), lst[k]<=lst[j]"""
    k = 0
    for i in range(1, len(lst)):
        # ?
        if lst[i] < lst[k]:
            k = i
    return k
```

lst

?	?	?	…														…	?
k=0	i=1	2	…														…	n-1

Effect of conditional statement

```
def min_index(lst):
    """accepts: list of length n>0 of comp. elements
       returns: index k in range(n) such that
                for all j in range(n), lst[k]<=lst[j]"""
    k = 0
    for i in range(1, len(lst)):
        # ?
        if lst[i] < lst[k]:
            k = i
    return k
```

lst

?	?	?	…														…	?
k=0	i=1	2	…														…	n-1

```
if lst[1] < lst[k]:
    k = 1
```

I st scenario

?	?	?	…														…	?
k=0	i=1	2	…														…	n-1

Effect of conditional statement

```
def min_index(lst):
    """accepts: list of length n>0 of comp. elements
       returns: index k in range(n) such that
                for all j in range(n), lst[k]<=lst[j]"""
    k = 0
    for i in range(1, len(lst)):
        # ?
        if lst[i] < lst[k]:
            k = i
    return k
```

lst

?	?	?	…														…	?
k=0	i=1	2	…														…	n-1

```
if lst[1] < lst[k]:
    k = 1
```

2nd scenario

?	?	?	…														…	?
0	k=i=1	2	…														…	n-1

In both cases: k is min index among the small index set {0, 1}

```
def min_index(lst):
    """accepts: list of length n>0 of comp. elements
       returns: index k in range(n) such that
                for all j in range(n), lst[k]<=lst[j]"""
    k = 0
    for i in range(1, len(lst)):
        # ?
        if lst[i] < lst[k]:
            k = i
        # lst[k] <= lst[0] and lst[k]<=lst[1]
    return k
```

lst

?	?	?	…														…	?
k=0	i=1	2	…														…	n-1

```
if lst[1] < lst[k]:
    k = 1
```

2nd scenario

?	?	?	…														…	?
0	k=i=1	2	…														…	n-1

This suggests general pattern

```
def min_index(lst):
    """accepts: list of length n>0 of comp. elements
       returns: index k in range(n) such that
                for all j in range(n), lst[k]<=lst[j]"""
    k = 0
    for i in range(1, len(lst)):
        # lst[k] <= lst[0]
        if lst[i] < lst[k]:
            k = i
        # lst[k] <= lst[0] and lst[k]<=lst[1]
    return k
```

lst

?	?	?	…														…	?
k=0	i=1	2	…														…	n-1

```
if lst[1] < lst[k]:
    k = 1
```

2nd scenario

?	?	?	…														…	?
0	k=i=1	2	…														…	n-1

This suggests general pattern

```
def min_index(lst):
    """accepts: list of length n>0 of comp. elements
       returns: index k in range(n) such that
                for all j in range(n), lst[k]<=lst[j]"""
    k = 0
    for i in range(1, len(lst)):
        # for all j in range(i): lst[k]<=lst[j]
        if lst[i] < lst[k]:
            k = i
        # for all j in range(i+1): lst[k]<=lst[j]
    return k
```

lst

?	?	?	…														…	?
k=0	i=1	2	…														…	n-1

```
if lst[1] < lst[k]:
    k = 1
```

2nd scenario

?	?	?	…														…	?
0	k=i=1	2	…														…	n-1

Let's consider general loop iteration

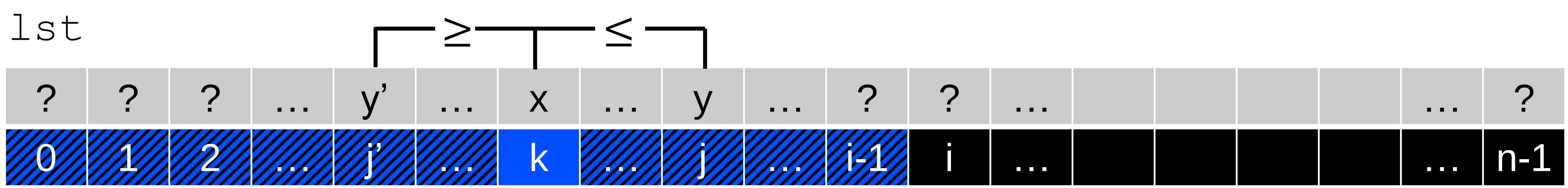
```
def min_index(lst):
    """accepts: list of length n>0 of comp. elements
       returns: index k in range(n) such that
                for all j in range(n), lst[k]<=lst[j]"""
    k = 0
    for i in range(1, len(lst)):
        # for all j in range(i): lst[k]<=lst[j]
        if lst[i] < lst[k]:
            k = i
        # for all j in range(i+1): lst[k]<=lst[j]
    return k
```

`lst`

?	?	?	…							…	?	…					…	?
0	1	2	…							…	i	…					…	n-1

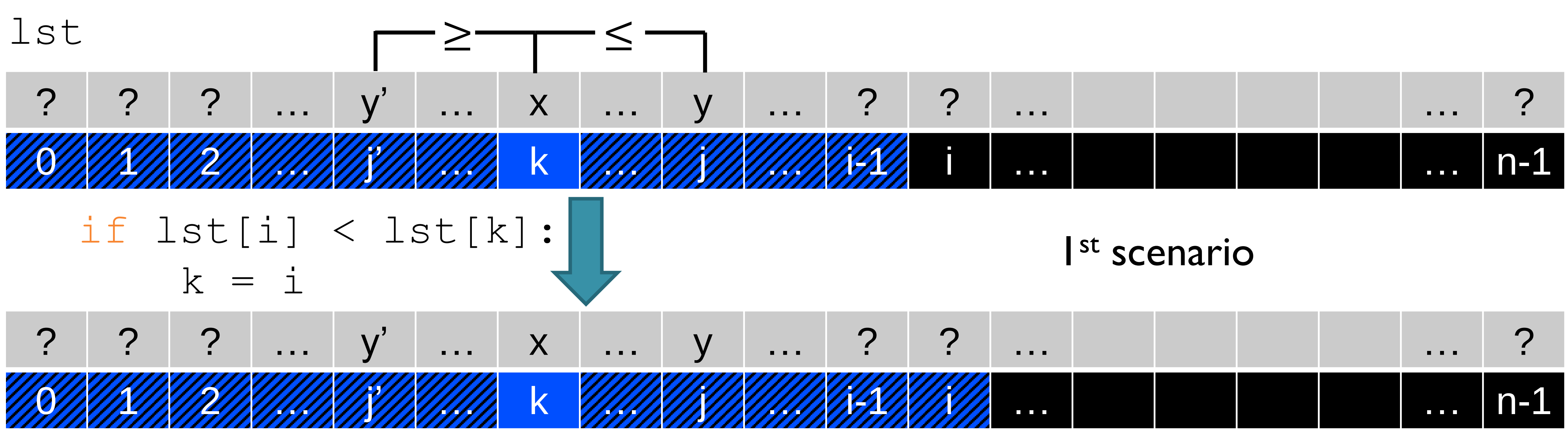
Assume first assertion is true

```
def min_index(lst):
    """accepts: list of length n>0 of comp. elements
       returns: index k in range(n) such that
                for all j in range(n), lst[k]<=lst[j]"""
    k = 0
    for i in range(1, len(lst)):
        # for all j in range(i): lst[k]<=lst[j]
        if lst[i] < lst[k]:
            k = i
        # for all j in range(i+1): lst[k]<=lst[j]
    return k
```



Effect of conditional statement

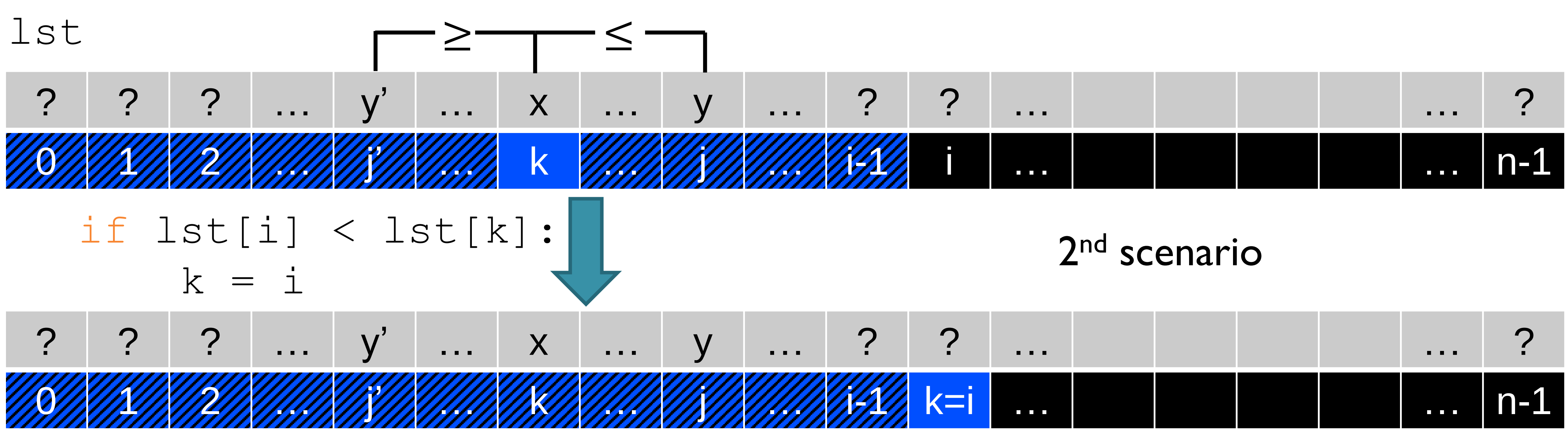
```
def min_index(lst):
    """accepts: list of length n>0 of comp. elements
       returns: index k in range(n) such that
                for all j in range(n), lst[k]<=lst[j]"""
    k = 0
    for i in range(1, len(lst)):
        # for all j in range(i): lst[k]<=lst[j]
        if lst[i] < lst[k]: k = i
        # for all j in range(i+1): lst[k]<=lst[j]
    return k
```



1st scenario

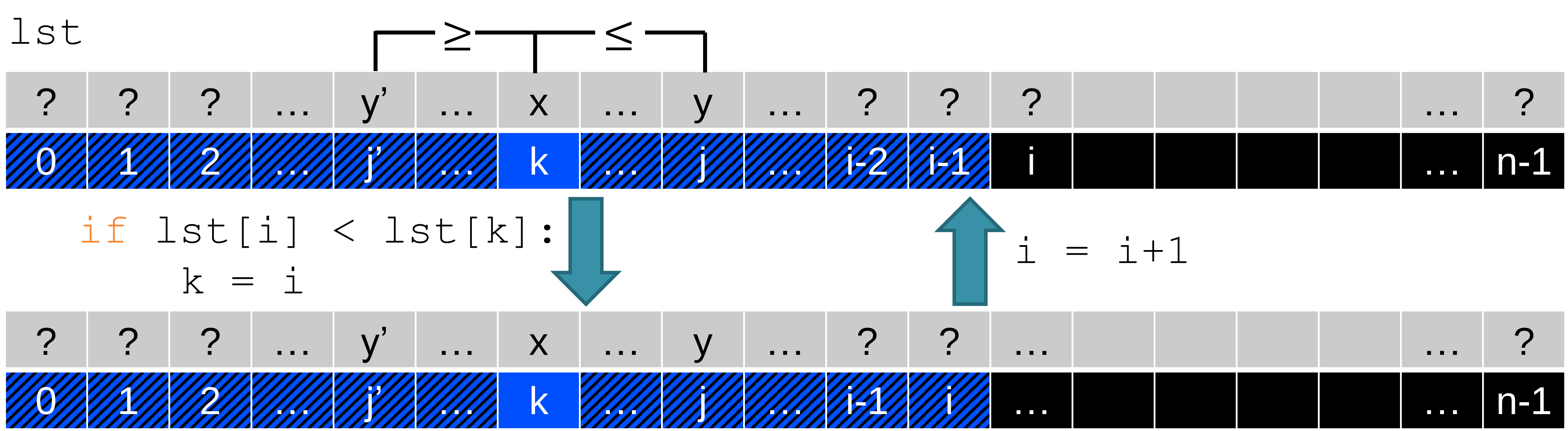
Conditional statement ensures second assertion

```python
def min_index(lst):
    """accepts: list of length n>0 of comp. elements
       returns: index k in range(n) such that
                for all j in range(n), lst[k]<=lst[j]"""
    k = 0
    for i in range(1, len(lst)):
        # for all j in range(i): lst[k]<=lst[j]
        if lst[i] < lst[k]: k = i
        # for all j in range(i+1): lst[k]<=lst[j]
    return k
```



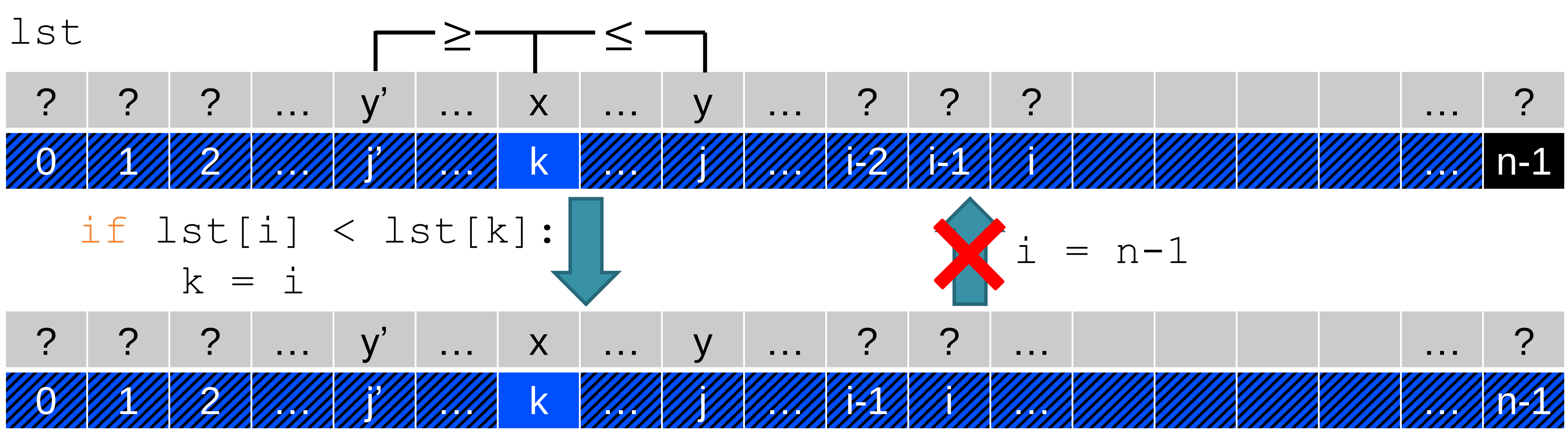
Which in turn assures first assertion in next iteration

```
def min_index(lst):
    """accepts: list of length n>0 of comp. elements
       returns: index k in range(n) such that
                for all j in range(n), lst[k]<=lst[j]"""
    k = 0
    for i in range(1, len(lst)):
        #I: for all j in range(i): lst[k]<=lst[j]
        if lst[i] < lst[k]: k = i
        #I': for all j in range(i+1): lst[k]<=lst[j]
    return k
```



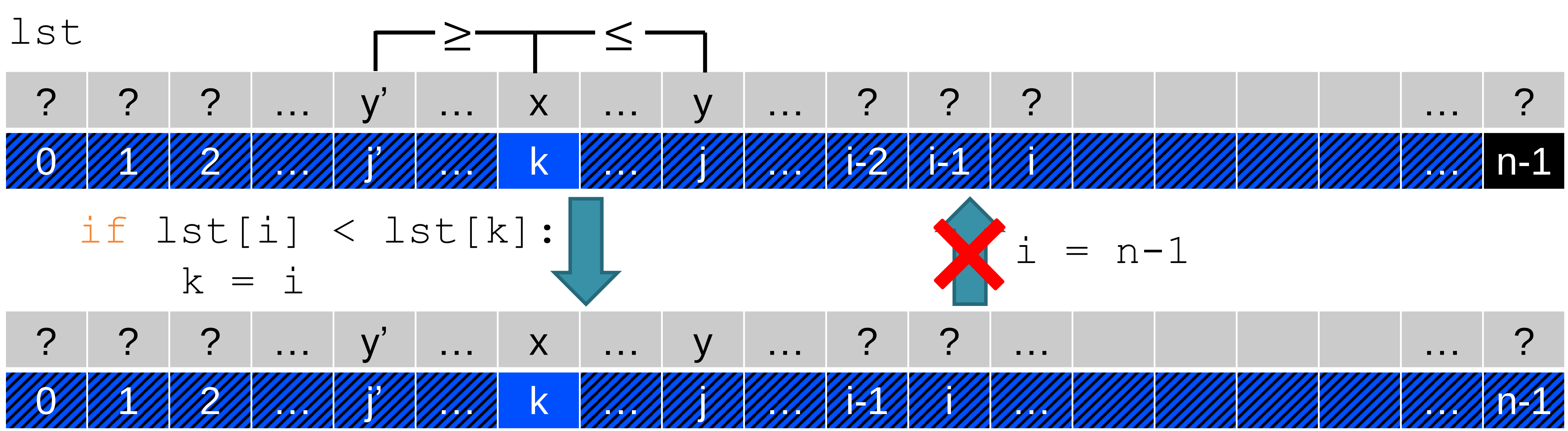
What happens at end of loop?

```
def min_index(lst):
    """accepts: list of length n>0 of comp. elements
       returns: index k in range(n) such that
                for all j in range(n), lst[k]<=lst[j]"""
    k = 0
    for i in range(1, len(lst)):
        #I: for all j in range(i): lst[k]<=lst[j]
        if lst[i] < lst[k]: k = i
        #I': for all j in range(i+1): lst[k]<=lst[j]
    #EXC: i = n-1
    return k
```



Again loop exit condition and invariants imply desired post cond.

```
def min_index(lst):
    """accepts: list of length n>0 of comp. elements
       returns: index k in range(n) such that
                for all j in range(n), lst[k]<=lst[j]"""
    k = 0
    for i in range(1, len(lst)):
        #I: for all j in range(i): lst[k]<=lst[j]
        if lst[i] < lst[k]: k = i
        #I': for all j in range(i+1): lst[k]<=lst[j]
    #EXC: i = n-1, #POC: for j in range(n): lst[k]<=lst[j]
    return k
```



Outline

- Assertions and invariants
- Analysing Insertion Sort
- Analysing Min Index Selection
- Analysing Prim's Algorithm

Prim's algorithm: does it always produce spanning tree?

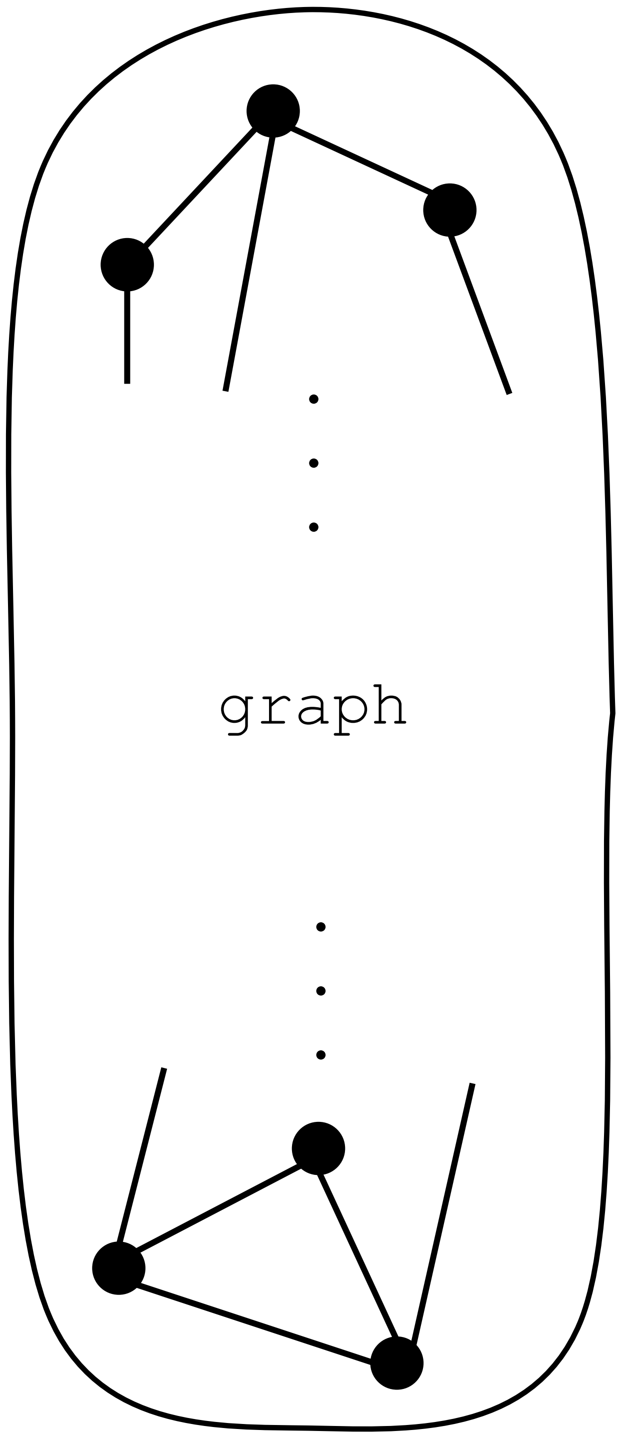
```
def extension(con, g):
    """input: vertices con connected in g
       output: edge (i,j) of g with i in
               con and j not in con"""
    for i in con:
        for j in range(len(g)):
            if j not in con and g[i][j]:
                return i, j
```

```
def spanning_tree(graph):
    """input: graph given as adj. matrix
       output: spanning tree of graph"""
    tree = empty_graph(len(graph))
    con = {0}
    while len(con) < len(graph):
        i, j = extension(con, graph)
        tree[i][j], tree[j][i] = 1, 1
        con.add{j}
return tree
```

Let us visualise generic input

```
def extension(con, g):
    """input: vertices con connected in g
       output: edge (i,j) of g with i in
               con and j not in con"""
                     …
```

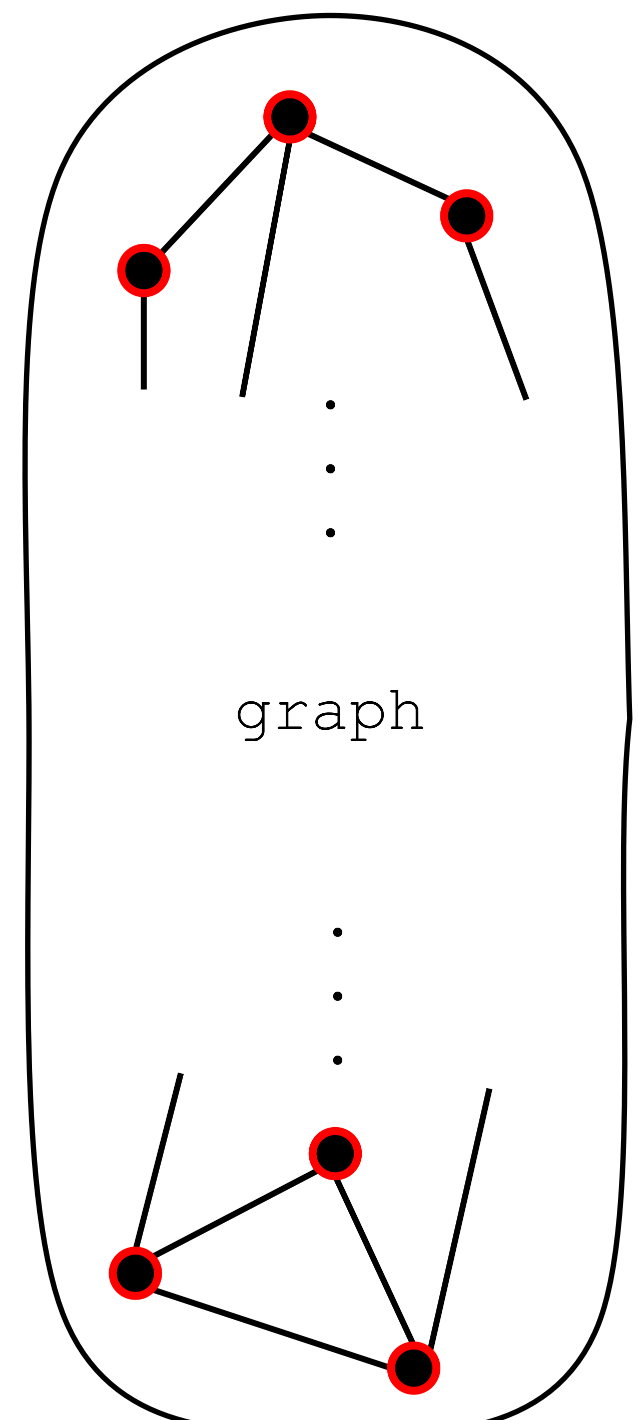
```
def spanning_tree(graph):
    """input: graph given as adj. matrix
       output: spanning tree of graph"""
    tree = empty_graph(len(graph))
    con = {0}
    while len(con) < len(graph):
        i, j = extension(con, graph)
        tree[i][j], tree[j][i] = 1, 1
        con.add{j}
return tree
```



…and analyse what happens during computation

```
def extension(con, g):
    “””input: vertices con connected in g
       output: edge (i,j) of g with i in
               con and j not in con“””
                     …
```

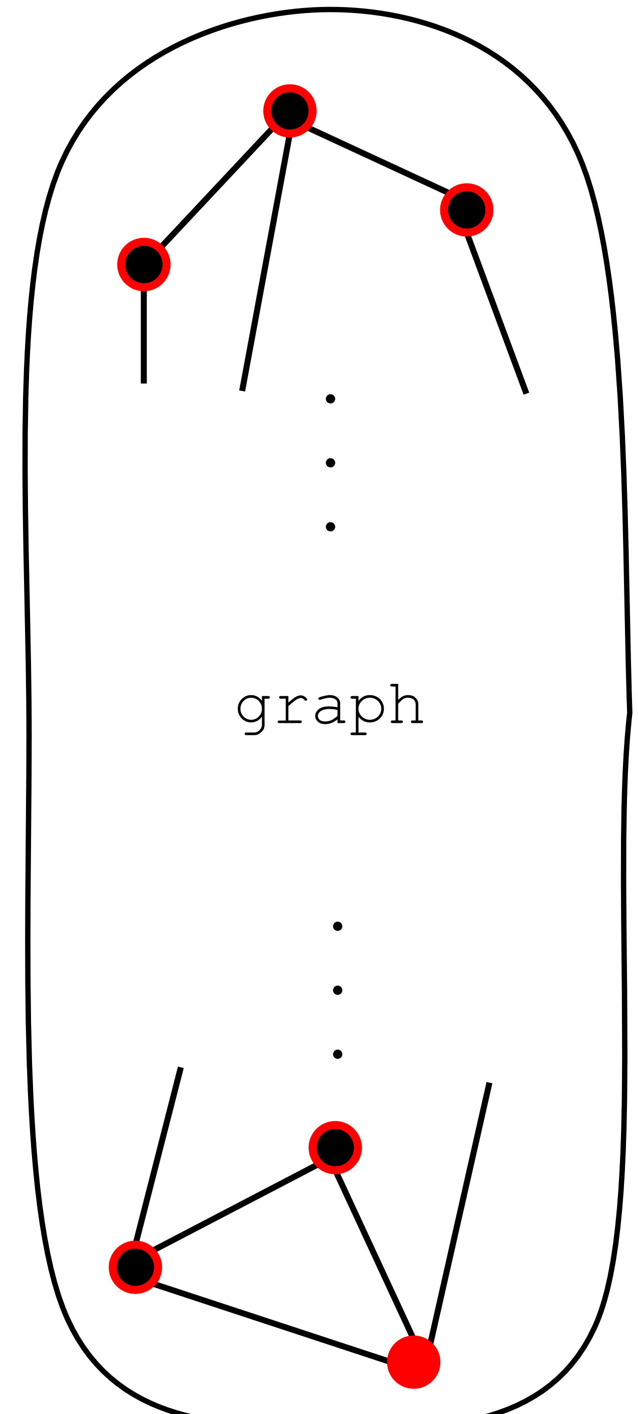
```
def spanning_tree(graph):
    “””input: graph given as adj. matrix
       output: spanning tree of graph“””
    tree = empty_graph(len(graph))
    con = {0}
    while len(con) < len(graph):
        i, j = extension(con, graph)
        tree[i][j], tree[j][i] = 1, 1
        con.add{j}
return tree
```



…and analyse what happens during computation

```
def extension(con, g):
    """input: vertices con connected in g
       output: edge (i,j) of g with i in
               con and j not in con"""
                    …
```

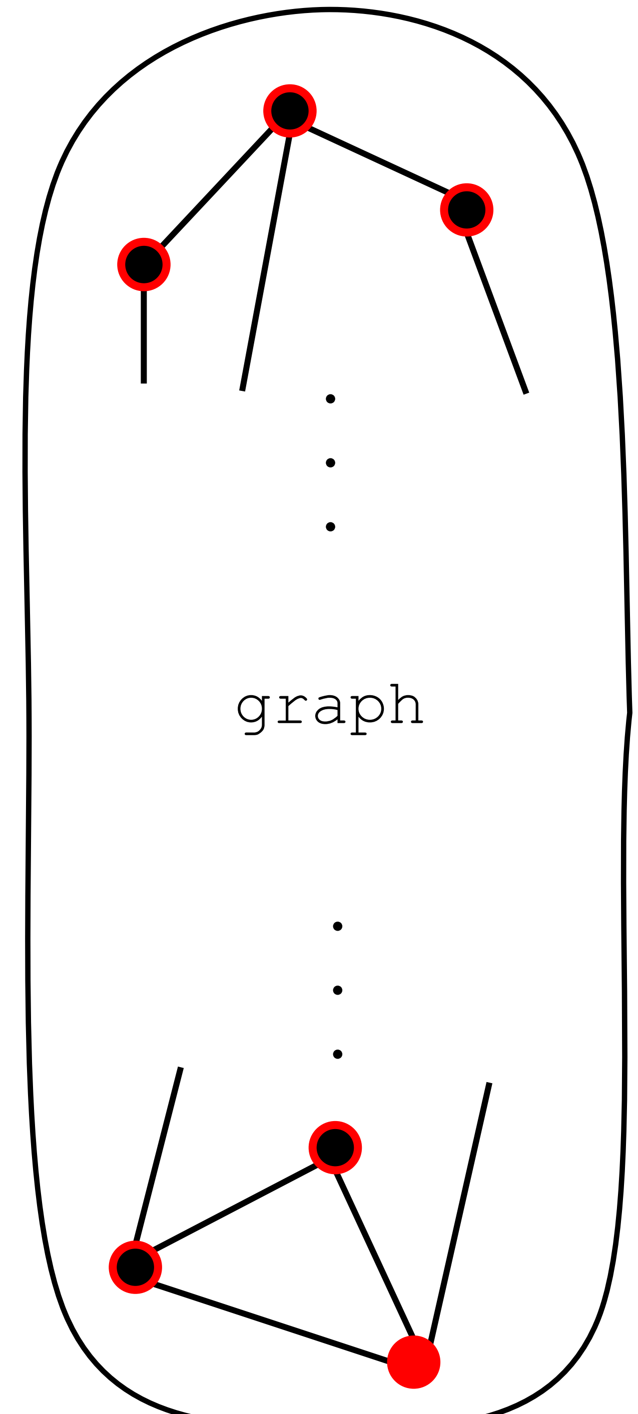
```
def spanning_tree(graph):
    """input: graph given as adj. matrix
       output: spanning tree of graph"""
    tree = empty_graph(len(graph))
    con = {0}
    while len(con) < len(graph):
        i, j = extension(con, graph)
        tree[i][j], tree[j][i] = 1, 1
        con.add{j}
return tree
```



…and analyse what happens during computation

```
def extension(con, g):
    """input: vertices con connected in g
       output: edge (i,j) of g with i in
               con and j not in con"""
                      …
```

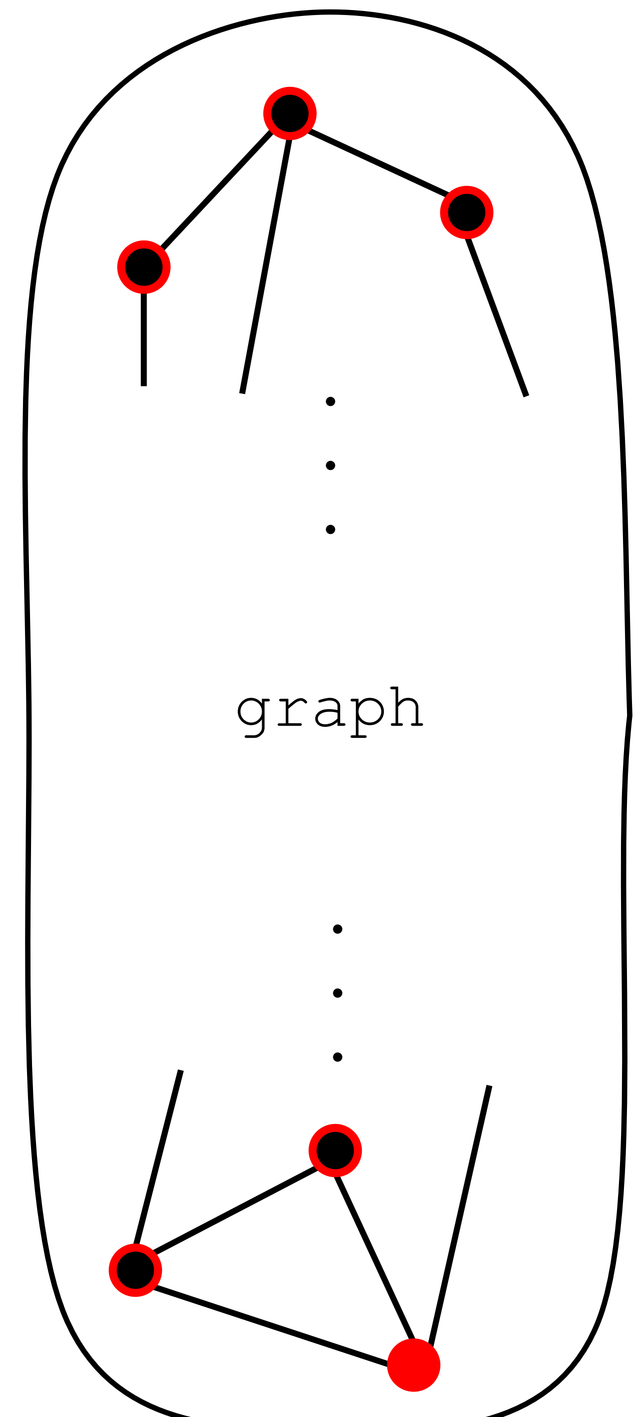
```
def spanning_tree(graph):
    """input: graph given as adj. matrix
       output: spanning tree of graph"""
    tree = empty_graph(len(graph))
    con = {0}
    while len(con) < len(graph):
        i, j = extension(con, graph)
        tree[i][j], tree[j][i] = 1, 1
        con.add{j}
return tree
```



…and analyse what happens during computation

```
def extension(con, g):
    """input: vertices con connected in g
       output: edge (i,j) of g with i in
               con and j not in con"""
                    …
```

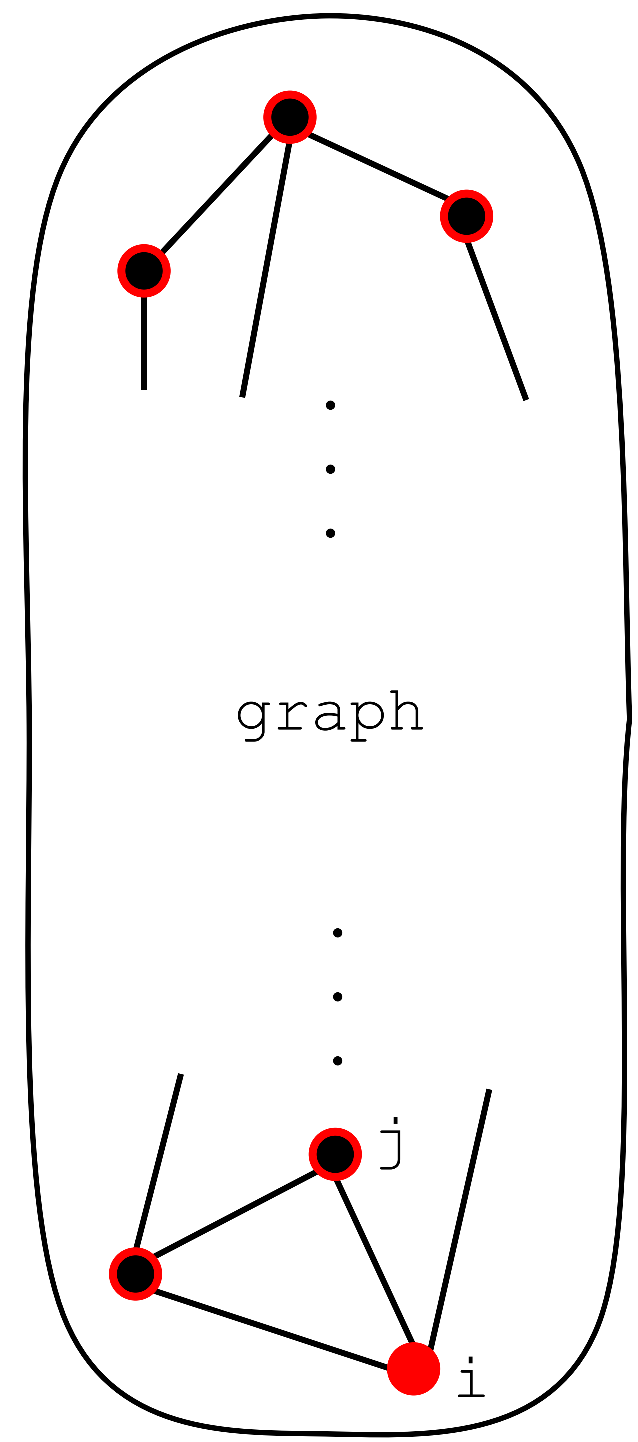
```
def spanning_tree(graph):
    """input: graph given as adj. matrix
       output: spanning tree of graph"""
    tree = empty_graph(len(graph))
    con = {0}
    while len(con) < len(graph):
        # con is connected in tree
        i, j = extension(con, graph)
        tree[i][j], tree[j][i] = 1, 1
        con.add{j}
return tree
```



…and analyse what happens during computation

```
def extension(con, g):
    """input: vertices con connected in g
       output: edge (i,j) of g with i in
               con and j not in con"""
                    …
```

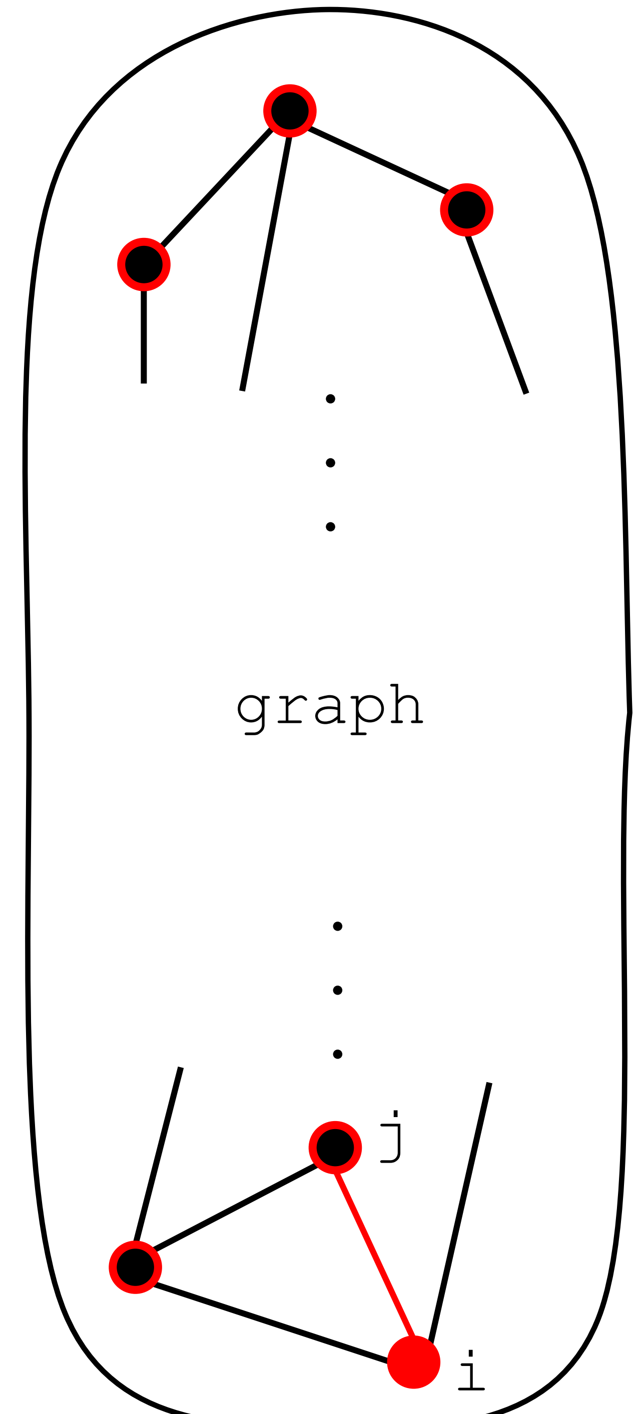
```
def spanning_tree(graph):
    """input: graph given as adj. matrix
       output: spanning tree of graph"""
    tree = empty_graph(len(graph))
    con = {0}
    while len(con) < len(graph):
        # con is connected in tree
        i, j = extension(con, graph)
        tree[i][j], tree[j][i] = 1, 1
        con.add{j}
return tree
```



…and analyse what happens during computation

```
def extension(con, g):
    """input: vertices con connected in g
       output: edge (i,j) of g with i in
               con and j not in con"""
                       …
```

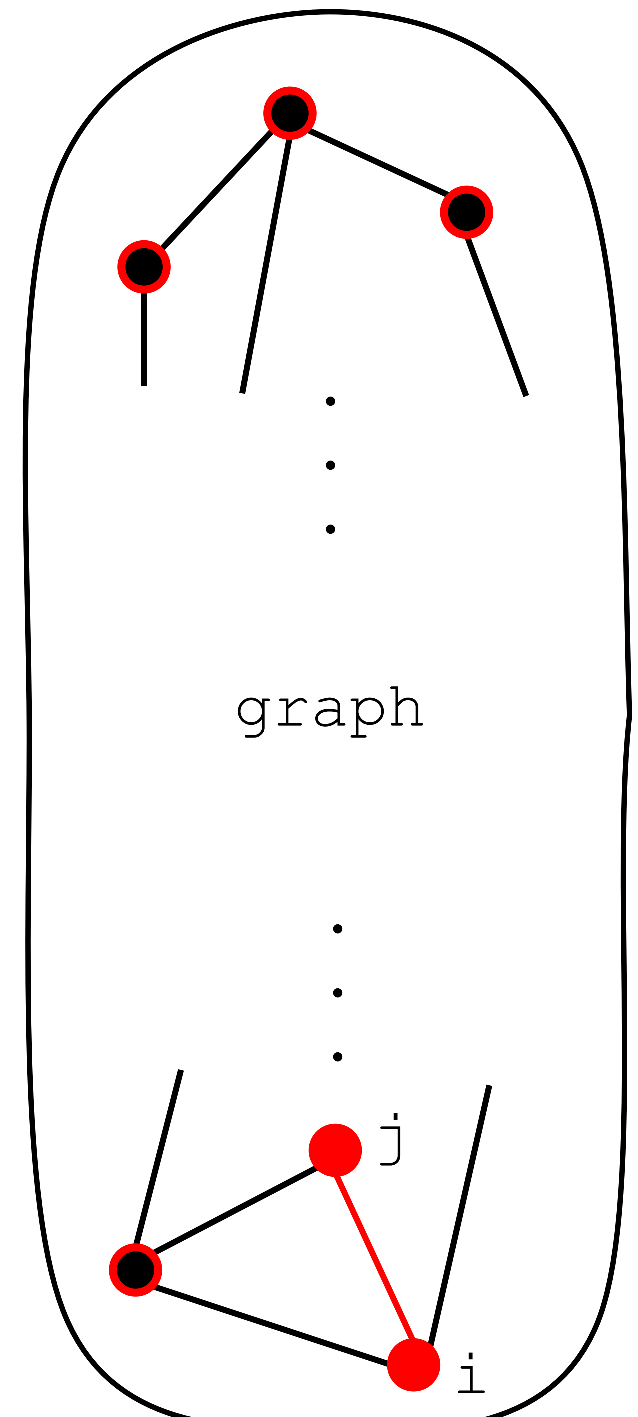
```
def spanning_tree(graph):
    """input: graph given as adj. matrix
       output: spanning tree of graph"""
    tree = empty_graph(len(graph))
    con = {0}
    while len(con) < len(graph):
        # con is connected in tree
        i, j = extension(con, graph)
        tree[i][j], tree[j][i] = 1, 1
        con.add{j}
return tree
```



…and analyse what happens during computation

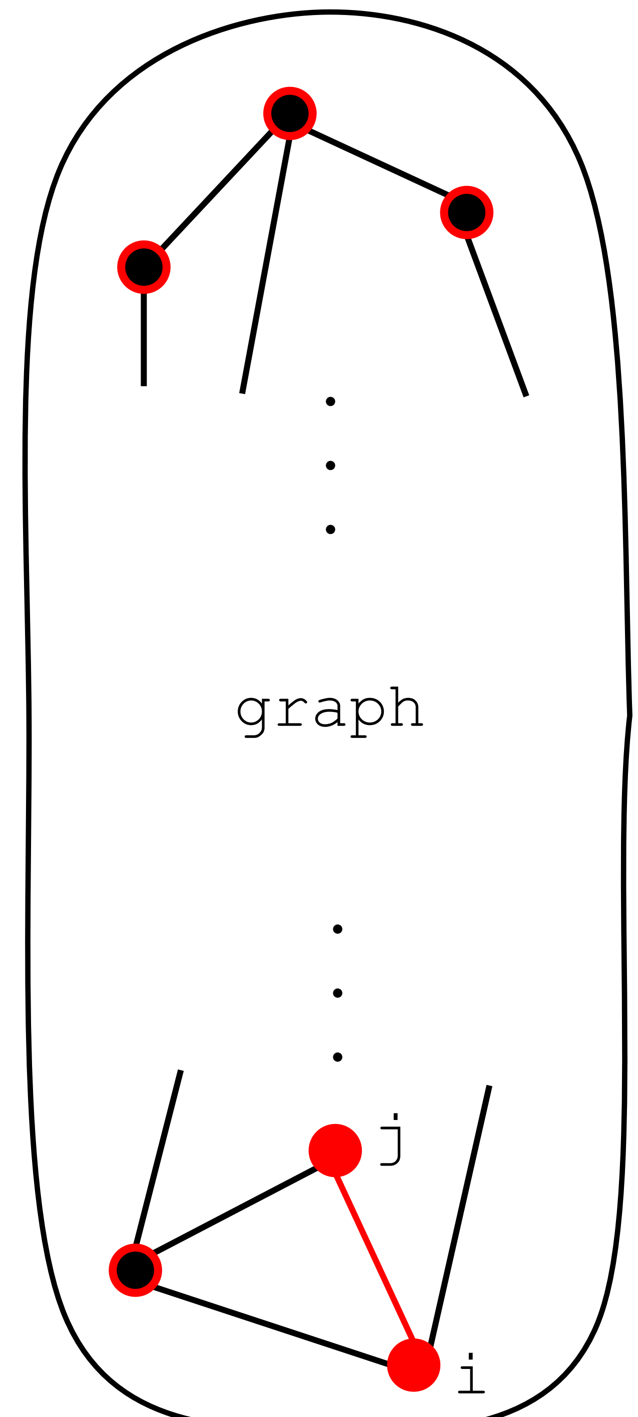
```
def extension(con, g):
    """input: vertices con connected in g
       output: edge (i,j) of g with i in
               con and j not in con"""
                    …
```

```
def spanning_tree(graph):
    """input: graph given as adj. matrix
       output: spanning tree of graph"""
    tree = empty_graph(len(graph))
    con = {0}
    while len(con) < len(graph):
        # con is connected in tree
        i, j = extension(con, graph)
        tree[i][j], tree[j][i] = 1, 1
        con.add{j}
return tree
```



…and analyse what happens during computation

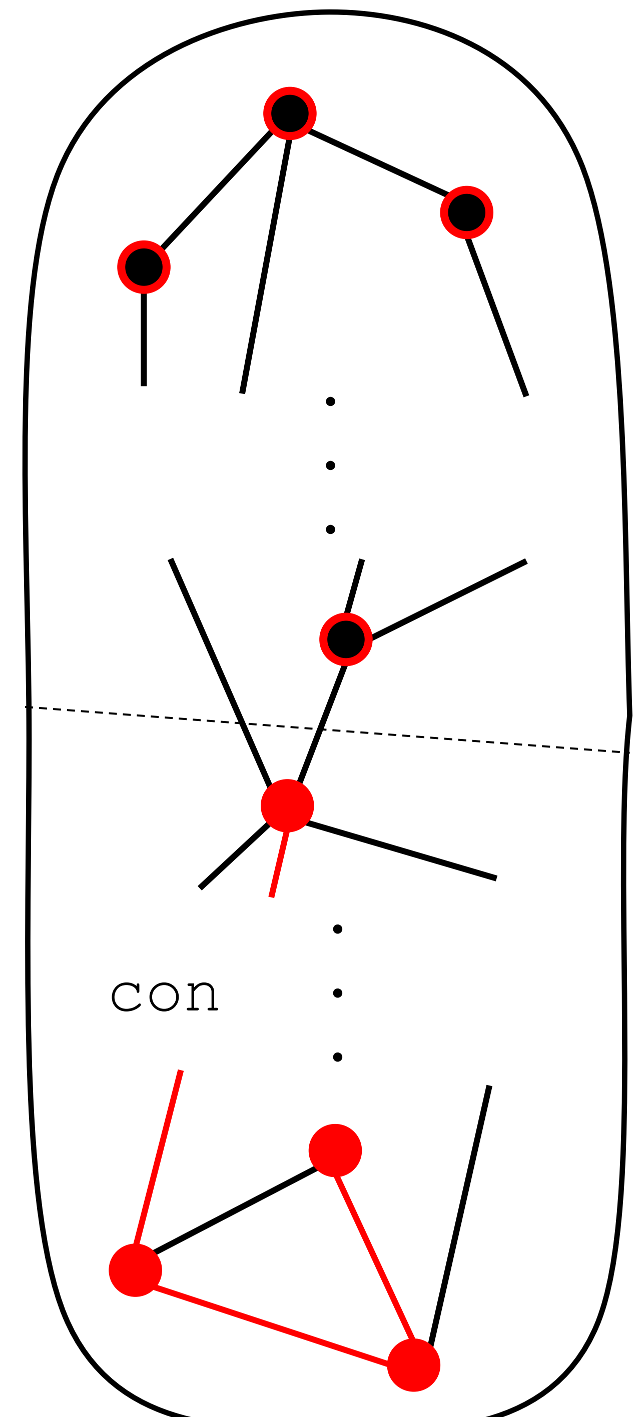
```
def extension(con, g):
    """input: vertices con connected in g
       output: edge (i,j) of g with i in
               con and j not in con"""
                     …

def spanning_tree(graph):
    """input: graph given as adj. matrix
       output: spanning tree of graph"""
    tree = empty_graph(len(graph))
    con = {0}
    while len(con) < len(graph):
        # con is connected in tree
        i, j = extension(con, graph)
        tree[i][j], tree[j][i] = 1, 1
        con.add{j}
        # con is connected in tree
return tree
```



Assume con is connected at start of arbitrary loop iteration

```
def extension(con, g):
    """input: vertices con connected in g
       output: edge (i,j) of g with i in
               con and j not in con"""
                    …
```

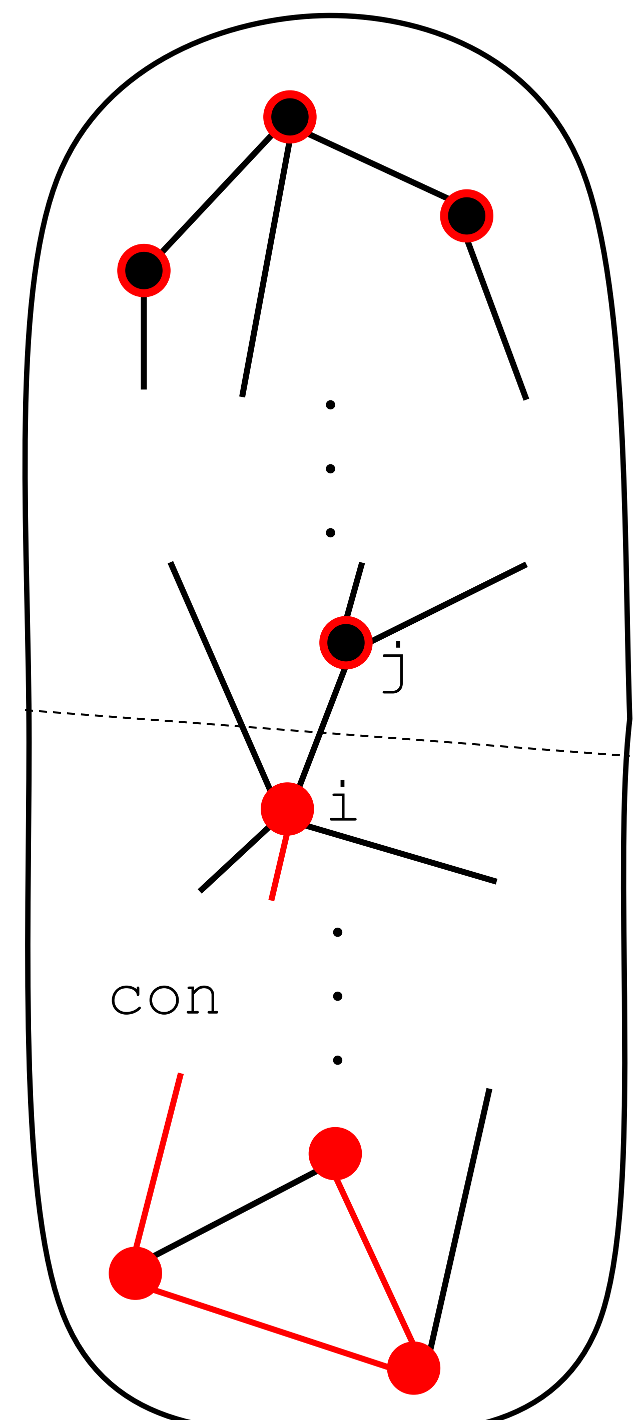
```
def spanning_tree(graph):
    """input: graph given as adj. matrix
       output: spanning tree of graph"""
    tree = empty_graph(len(graph))
    con = {0}
    while len(con) < len(graph):
        # con is connected in tree
        i, j = extension(con, graph)
        tree[i][j], tree[j][i] = 1, 1
        con.add{j}
        # con is connected in tree
return tree
```



Extension edge bridges connected to not yet connected

```
def extension(con, g):
    """input: vertices con connected in g
       output: edge (i,j) of g with i in
               con and j not in con"""
                      …
```

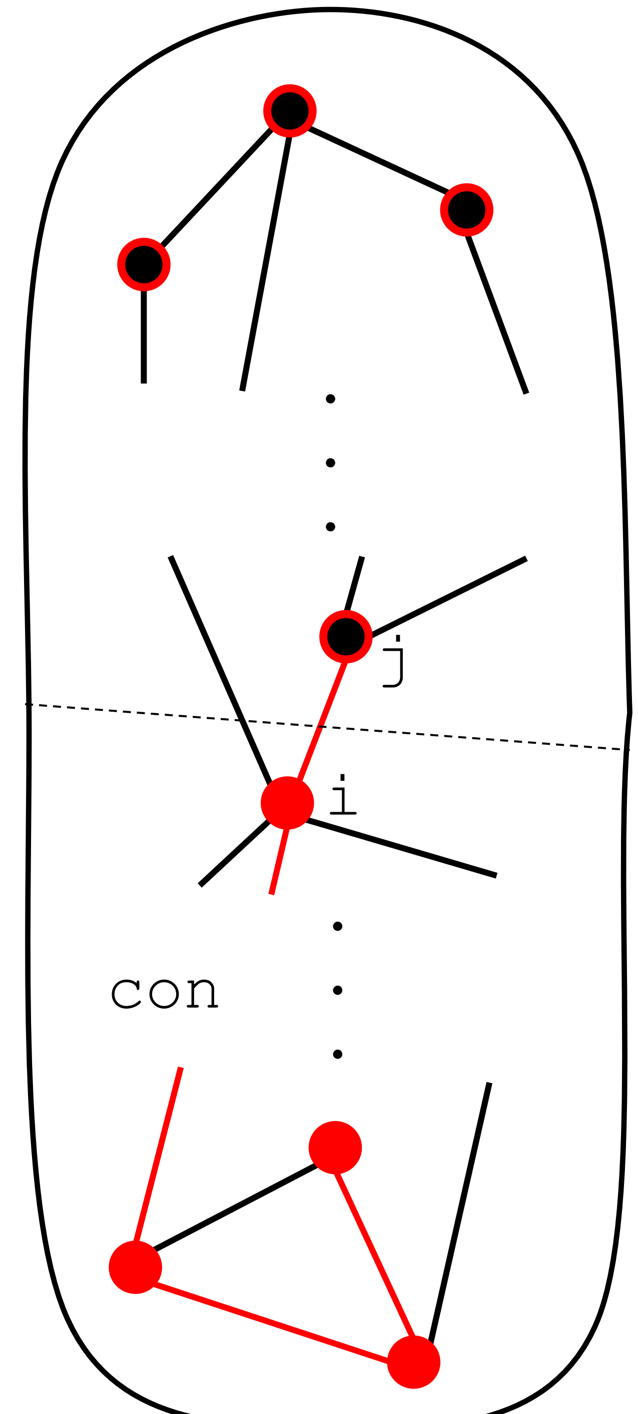
```
def spanning_tree(graph):
    """input: graph given as adj. matrix
       output: spanning tree of graph"""
    tree = empty_graph(len(graph))
    con = {0}
    while len(con) < len(graph):
        # con is connected in tree
        i, j = extension(con, graph)
        tree[i][j], tree[j][i] = 1, 1
        con.add{j}
        # con is connected in tree
return tree
```



Extension edge bridges connected to not yet connected

```
def extension(con, g):
    """input: vertices con connected in g
       output: edge (i,j) of g with i in
               con and j not in con"""
                    …
```

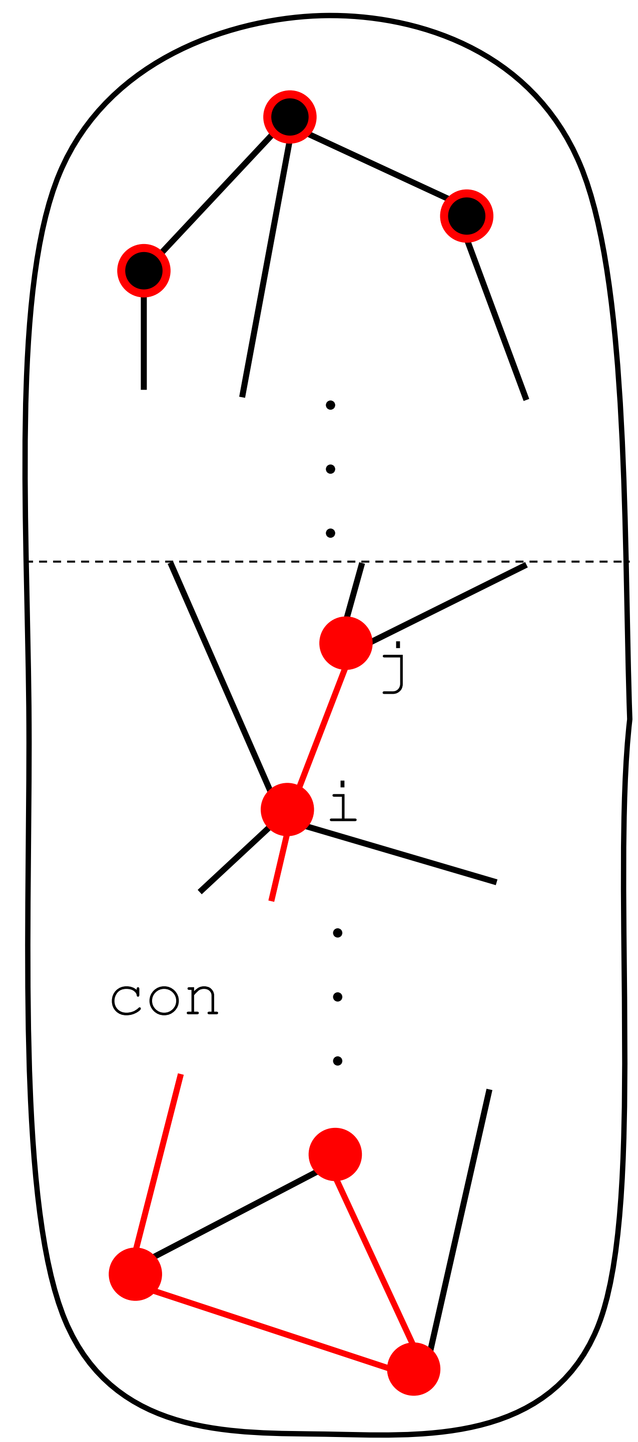
```
def spanning_tree(graph):
    """input: graph given as adj. matrix
       output: spanning tree of graph"""
    tree = empty_graph(len(graph))
    con = {0}
    while len(con) < len(graph):
        # con is connected in tree
        i, j = extension(con, graph)
        tree[i][j], tree[j][i] = 1, 1
        con.add{j}
        # con is connected in tree
return tree
```



After adding extension edge to tree j is also connected

```
def extension(con, g):
    “””input: vertices con connected in g
       output: edge (i,j) of g with i in
               con and j not in con”””
                    …
```

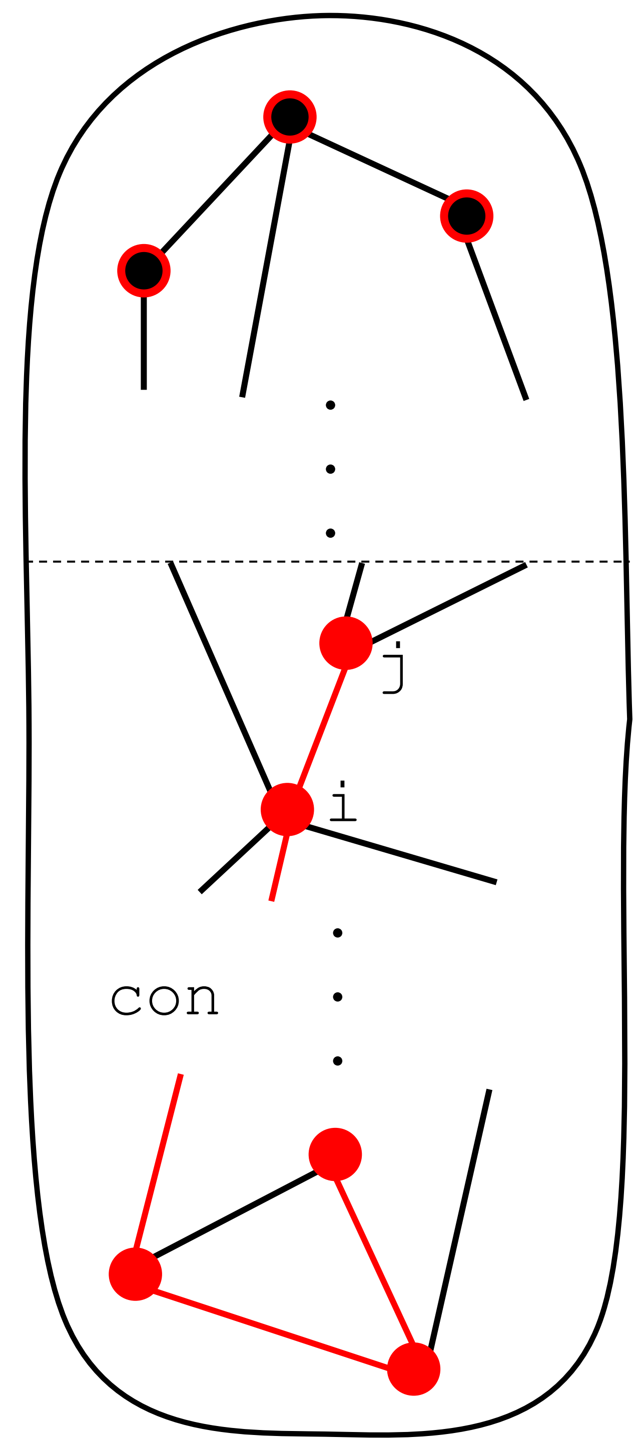
```
def spanning_tree(graph):
    “””input: graph given as adj. matrix
       output: spanning tree of graph”””
    tree = empty_graph(len(graph))
    con = {0}
    while len(con) < len(graph):
        # con is connected in tree
        i, j = extension(con, graph)
        tree[i][j], tree[j][i] = 1, 1
        con.add{j}
        # con is connected in tree
return tree
```



Invariant: con is connected

```
def extension(con, g):
    """input: vertices con connected in g
       output: edge (i,j) of g with i in
               con and j not in con"""
                      …
```

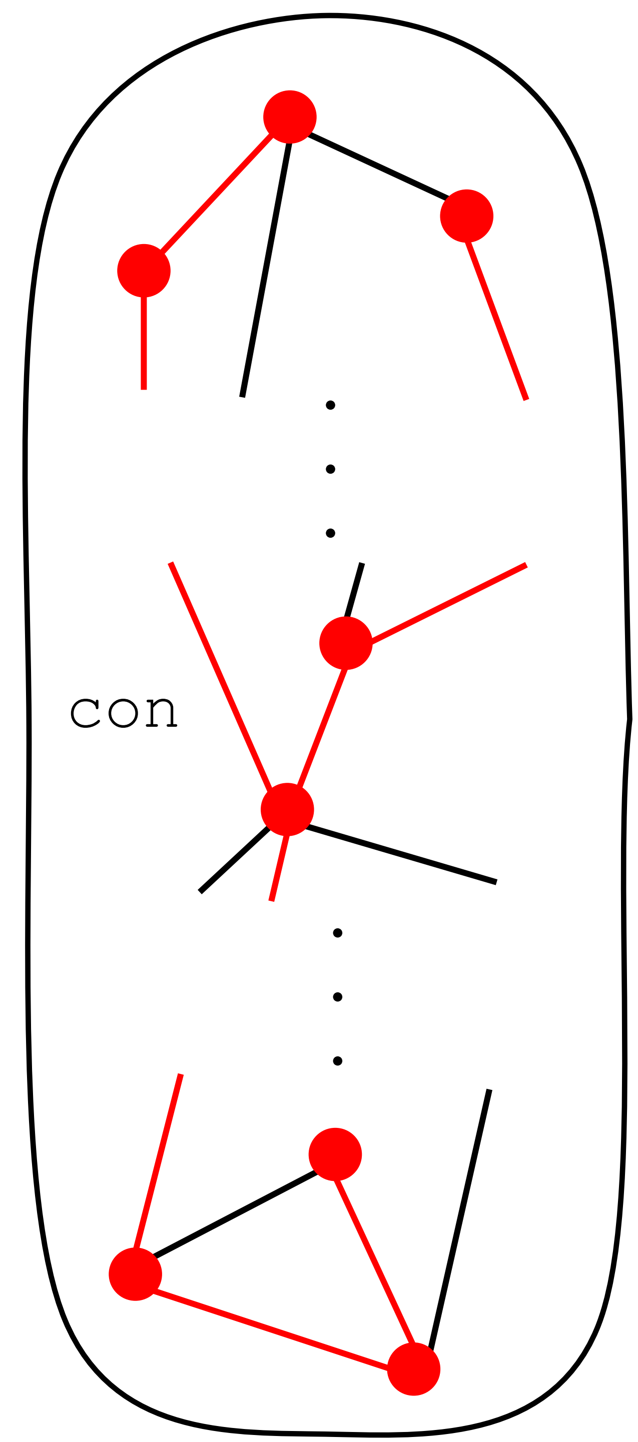
```
def spanning_tree(graph):
    """input: graph given as adj. matrix
       output: spanning tree of graph"""
    tree = empty_graph(len(graph))
    con = {0}
    while len(con) < len(graph):
        #I: con is connected in tree
        i, j = extension(con, graph)
        tree[i][j], tree[j][i] = 1, 1
        con.add{j}
        #I: con is connected in tree
return tree
```



Is this enough to conclude desired post condition?

```
def extension(con, g):
    """input: vertices con connected in g
       output: edge (i,j) of g with i in
               con and j not in con"""
                    …
```

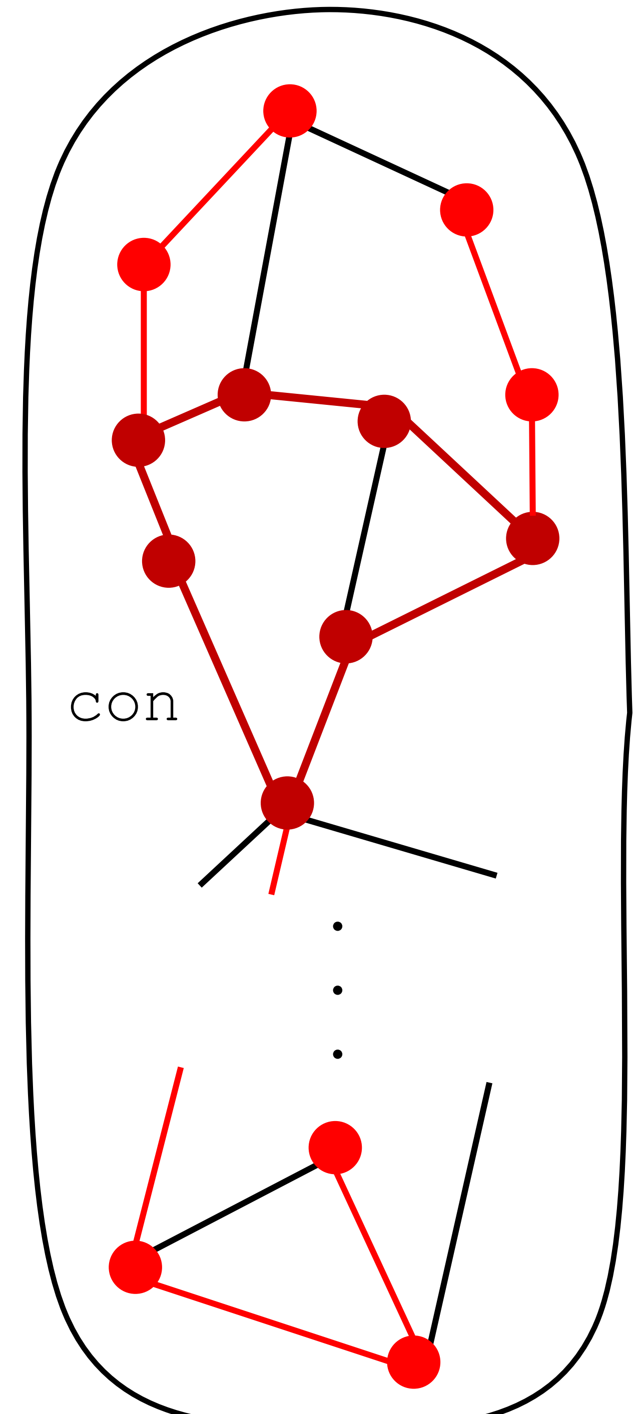
```
def spanning_tree(graph):
    """input: graph given as adj. matrix
       output: spanning tree of graph"""
    tree = empty_graph(len(graph))
    con = {0}
    while len(con) < len(graph):
        i, j = extension(con, graph)
        tree[i][j], tree[j][i] = 1, 1
        con.add{j}
        #I: con is connected in tree
    #EXC: len(con) == len(graph)
return tree
```



No. Can conclude that tree connected, but could contain cycle

```
def extension(con, g):
    """input: vertices con connected in g
       output: edge (i,j) of g with i in
               con and j not in con"""
                     …
```

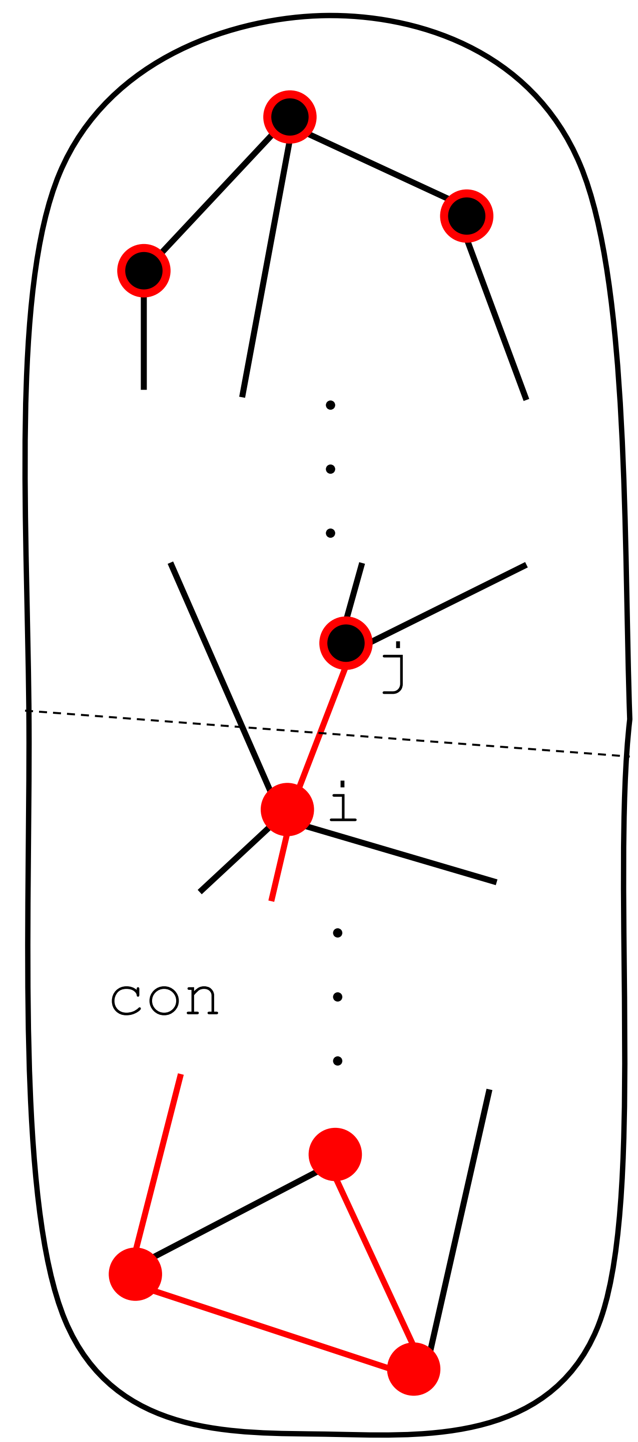
```
def spanning_tree(graph):
    """input: graph given as adj. matrix
       output: spanning tree of graph"""
    tree = empty_graph(len(graph))
    con = {0}
    while len(con) < len(graph):
        i, j = extension(con, graph)
        tree[i][j], tree[j][i] = 1, 1
        con.add{j}
        #I: con is connected in tree
    #EXC: len(con) == len(graph)
    #POC: tree connected
return tree
```



What should or second invariant be?

```
def extension(con, g):
    """input: vertices con connected in g
       output: edge (i,j) of g with i in
               con and j not in con"""
                         …
```

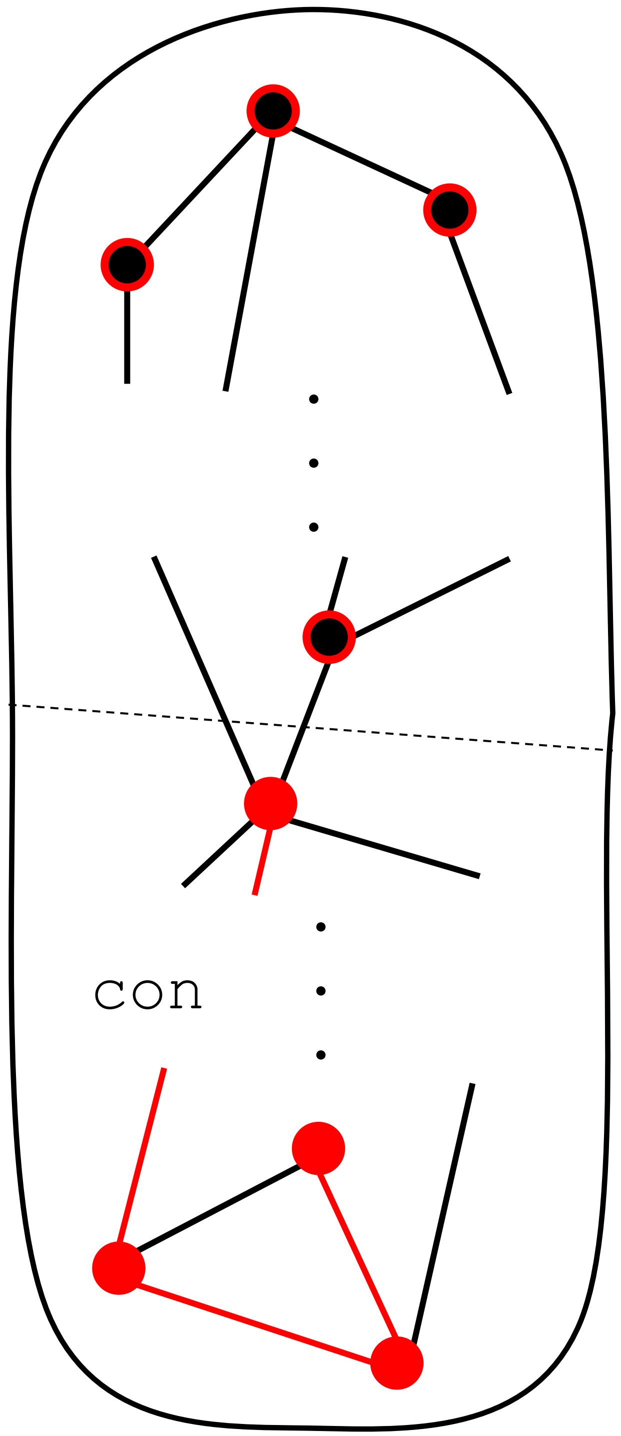
```
def spanning_tree(graph):
    """input: graph given as adj. matrix
       output: spanning tree of graph"""
    tree = empty_graph(len(graph))
    con = {0}
    while len(con) < len(graph):
        i, j = extension(con, graph)
        tree[i][j], tree[j][i] = 1, 1
        con.add{j}
        #I1: con is connected in tree
        #I2: ?
return tree
```



Need to guarantee that we never add a cycle to tree

```
def extension(con, g):
    """input: vertices con connected in g
       output: edge (i,j) of g with i in
               con and j not in con"""
                    …
```

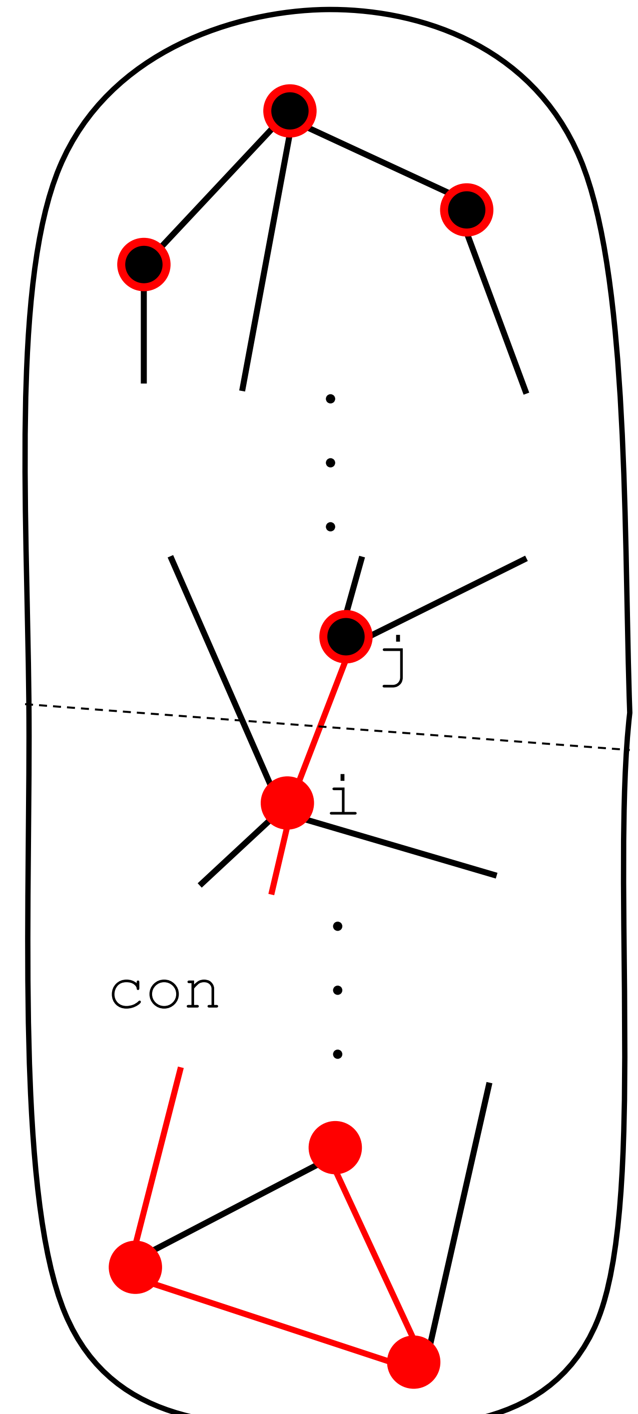
```
def spanning_tree(graph):
    """input: graph given as adj. matrix
       output: spanning tree of graph"""
    tree = empty_graph(len(graph))
    con = {0}
    while len(con) < len(graph):
        i, j = extension(con, graph)
        tree[i][j], tree[j][i] = 1, 1
        con.add{j}
        #I: con is connected in tree
        # tree does not contain cycle
return tree
```



Observation: extension edge never creates cycle with edges in tree

```
def extension(con, g):
    """input: vertices con connected in g
       output: edge (i,j) of g with i in
               con and j not in con"""
                     …
```

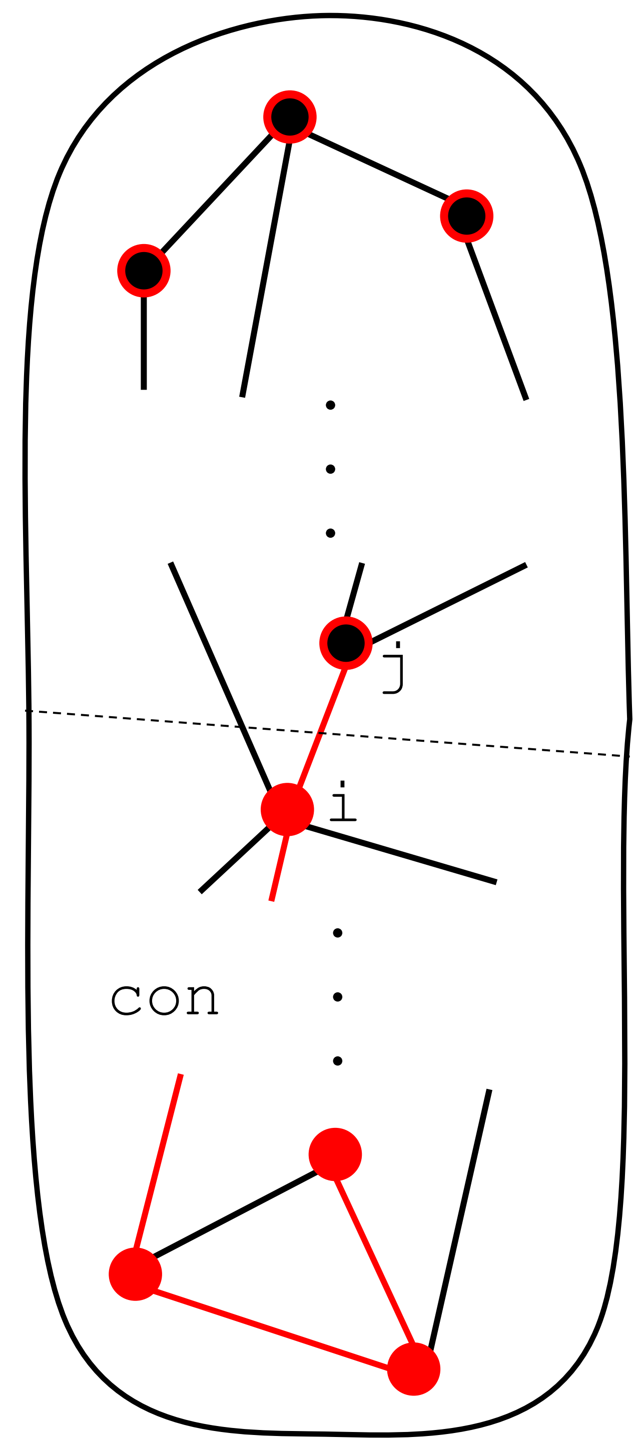
```
def spanning_tree(graph):
    """input: graph given as adj. matrix
       output: spanning tree of graph"""
    tree = empty_graph(len(graph))
    con = {0}
    while len(con) < len(graph):
        # tree does not contain a cycle
        i, j = extension(con, graph)
        tree[i][j], tree[j][i] = 1, 1
        con.add{j}
        #I1: con is connected in tree
        # tree does not contain a cycle
return tree
```



Invariant 2: tree does not contain a cycle

```
def extension(con, g):
    """input: vertices con connected in g
       output: edge (i,j) of g with i in
               con and j not in con"""
                    …
```

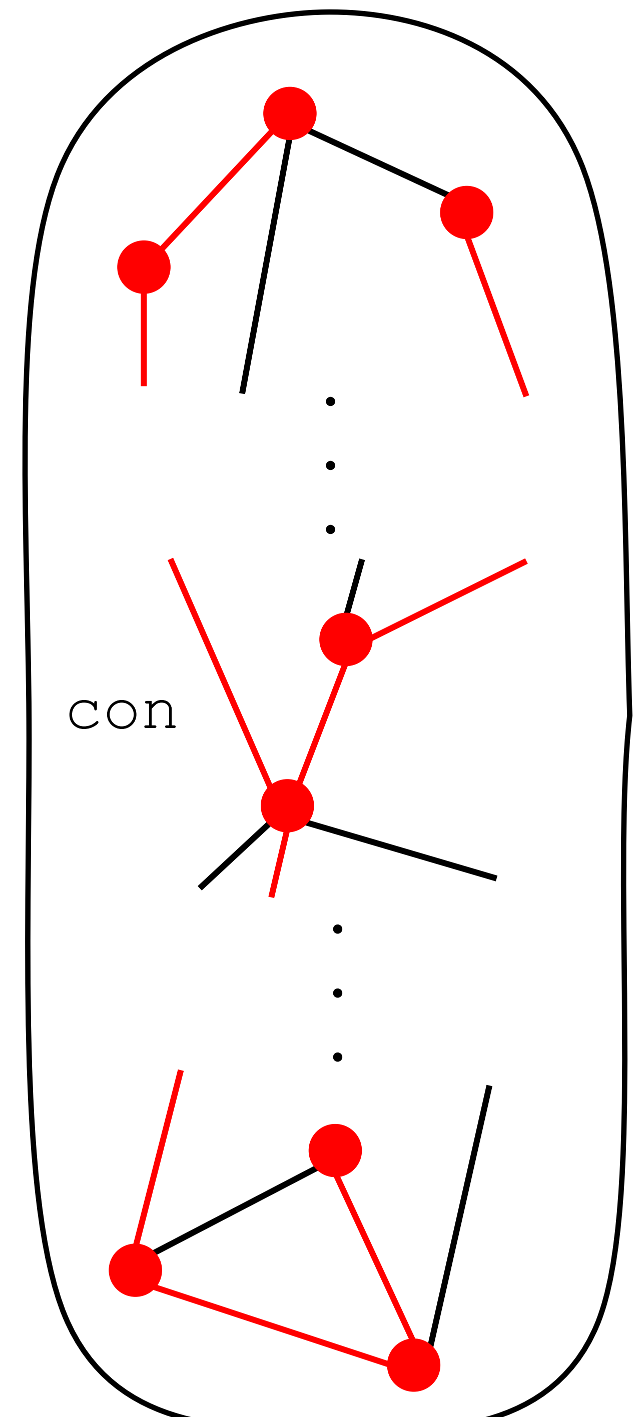
```
def spanning_tree(graph):
    """input: graph given as adj. matrix
       output: spanning tree of graph"""
    tree = empty_graph(len(graph))
    con = {0}
    while len(con) < len(graph):
        i, j = extension(con, graph)
        tree[i][j], tree[j][i] = 1, 1
        con.add{j}
        #I1: con is connected in tree
        #I2: tree does not contain cycle
return tree
```



Now we know tree is connected and without cycle at loop exit

```
def extension(con, g):
    """input: vertices con connected in g
       output: edge (i,j) of g with i in
               con and j not in con"""
                    …
```

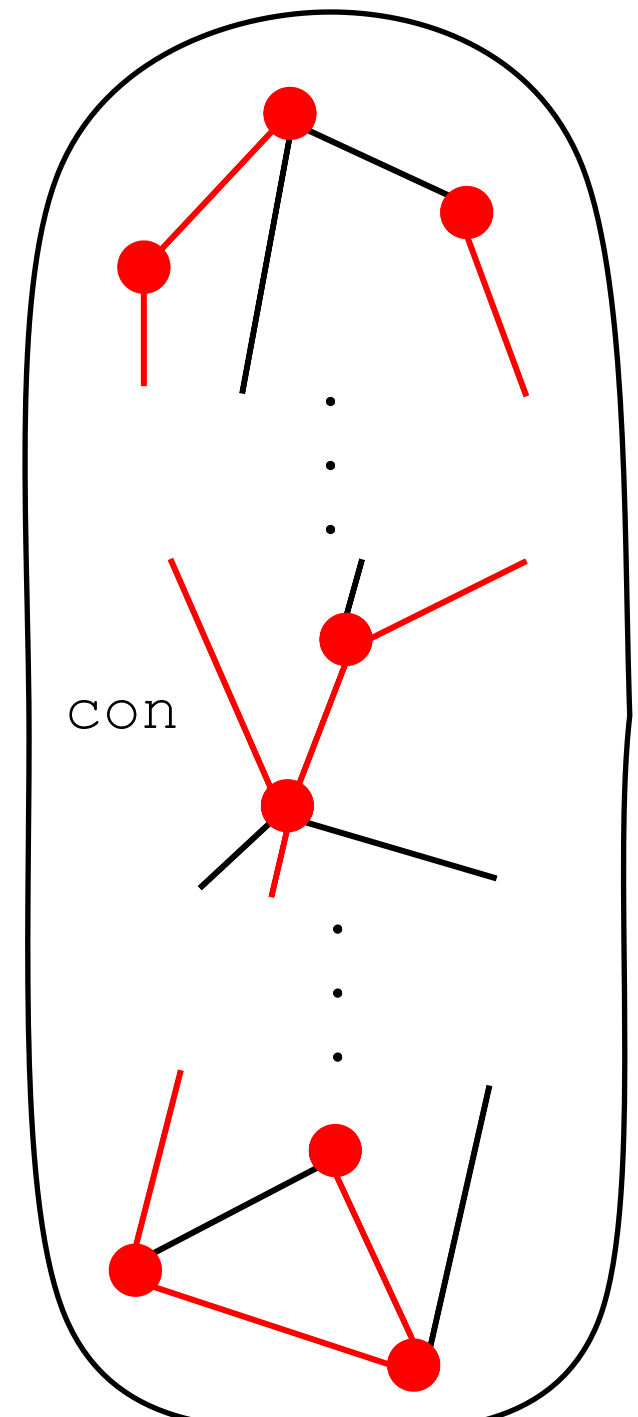
```
def spanning_tree(graph):
    """input: graph given as adj. matrix
       output: spanning tree of graph"""
    tree = empty_graph(len(graph))
    con = {0}
    while len(con) < len(graph):
        i, j = extension(con, graph)
        tree[i][j], tree[j][i] = 1, 1
        con.add{j}
        #I1: con is connected in tree
        #I2: tree does not contain cycle
    #EXC:  len(con)==len(graph)
return tree
```



…in other words: tree must be a spanning tree!!

```
def extension(con, g):
    """input: vertices con connected in g
       output: edge (i,j) of g with i in
               con and j not in con"""
                   …
```

```
def spanning_tree(graph):
    """input: graph given as adj. matrix
       output: spanning tree of graph"""
    tree = empty_graph(len(graph))
    con = {0}
    while len(con) < len(graph):
        i, j = extension(con, graph)
        tree[i][j], tree[j][i] = 1, 1
        con.add{j}
        #I1: con is connected in tree
        #I2: tree does not contain cycle
    #EXC: len(con)==len(graph)
    #POC: tree is spanning tree of graph
return tree
```



What have we learnt?

- Use assertions about execution state to reason about programs
- Loop invariants can be used to analyse behaviour of loopy control flows
- Look for invariants that turn into desired post-condition when loop exit condition is true

Coming Up

- Computational complexity
- Search
- More invariants