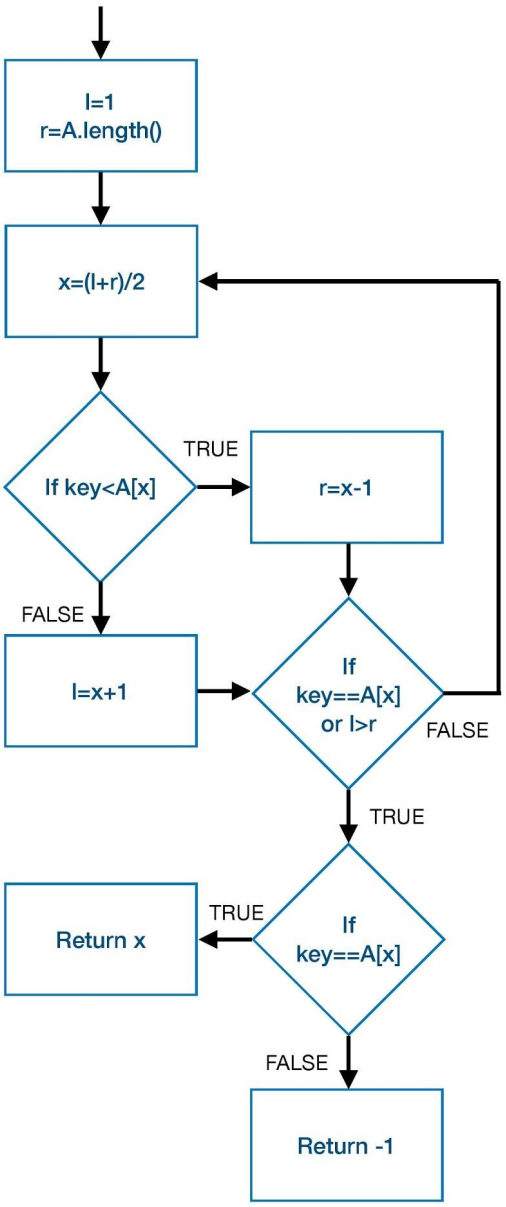
Lab 2: Black-box and Requirements-Based Testing: Sorting and Searching

Sofie Madsen Franzén and Daniel Saber Tehrani

Exercise 1. Condensation Graph

Draw a condensation graph for Algorithm 2.



Exercise 2. JML

Write JML pre- and postconditions for a. ***sorting****, b.* ***searching****, c.* ***membership****, and d.* ***binary searching****.*

a. requires A != null
ensures (\forall int i; 0 <= i & i <= A.length-1; A[i] <= A[1+i]) &
(\old(A.length) == A.length) &
(\forall int j; 0 <= j & j < A.length; \num_of int i; 0 <=i & i <= A.length; A[i] ==
\old(A[j]) == \num_of int i; 0 <= i & i < A.length; \old(A[i]) == \old(A[j])

b. requires A != null
ensures(\exists int i; 0 <= i & i <A.length; (key == A[i])) ⇒ (\result >= 0 &
\result < A.length & A[\result] == key)

c. requires A != null
ensures \result == (\exists int i; 0 <= i < A.length; A[i] == key)

d. requires A != null
requires (\forall int i; 0 <= i & i <= A.length-1; A[i] <= A[1+i]) &
(\old(A.length) == A.length) &
(\forall int j; 0 <= j & j < A.length; \num_of int i; 0 <=i & i <= A.length; A[i] ==
\old(A[j]) == \num_of int i; 0 <= i & i < A.length; \old(A[i]) == \old(A[j])
ensures A[\result] == key | \result == -1

Exercise 3. Implementation

Implement sorting of integer arrays of arbitrary length, membership queries on sorted arrays of arbitrary length using binary search, and membership queries on unsorted arrays of arbitrary length by combining the two previous implementations.

i)

```
def sortering(array):
    i = 1
    while i < (len(array)):
        int1 = i
        while (int1 > 0) and (array[int1] < array[int1-1]):
            temporar = array[int1]
            array[int1] = array[int1-1]
            array[int1-1] = temporar
            int1 -= 1
        i += 1
    return array
```

ii)

```python
def isMember(key, array):
    L=0
    R=len(array)-1
    X = (L+R)//2
    while array[X] != key and L <= R:
        if array[X] < key:
            L=X+1
        else:
            R=X-1
        X = (L+R)//2
    return (array[X] == key)
```

iii)

```python
def binarysortsearchmember(key,array):
    sortedArray = sortering(array)
    print(sortedArray)
    return(isMember(key,sortedArray))
```

Exercise 4. Testing

Build a random and a pairwise testing framework for the combined membership query implementation.

First of all we have the code for the randomvise tester and underneath we can see the test code where you run the original binary search and compare it with a mutation and tha system will get a result when the results don't match.

```
def randomTest(arraySize=8, start=0, stop=100):
    array=[]
    for i in range(0,arraySize,1):
        array.append(random.randint(start, stop))
    return array
```

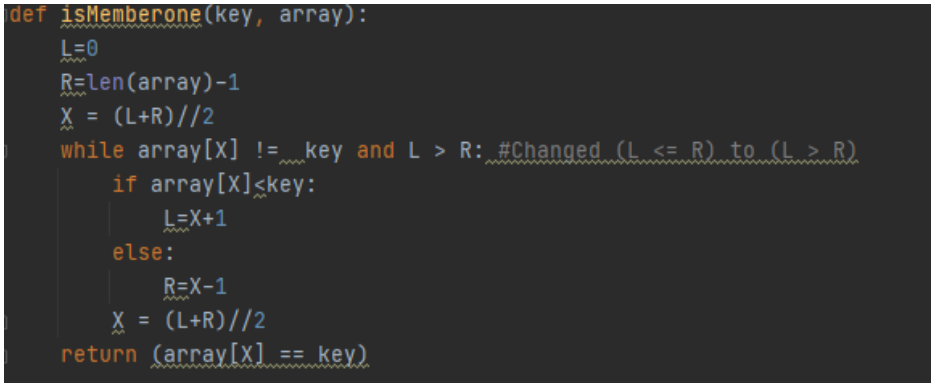
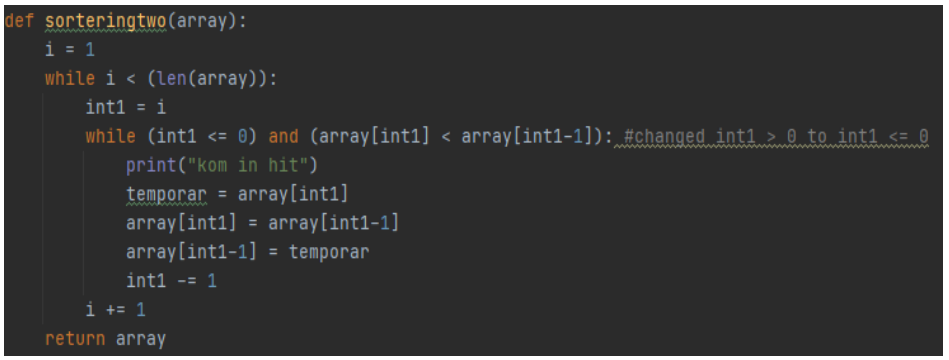
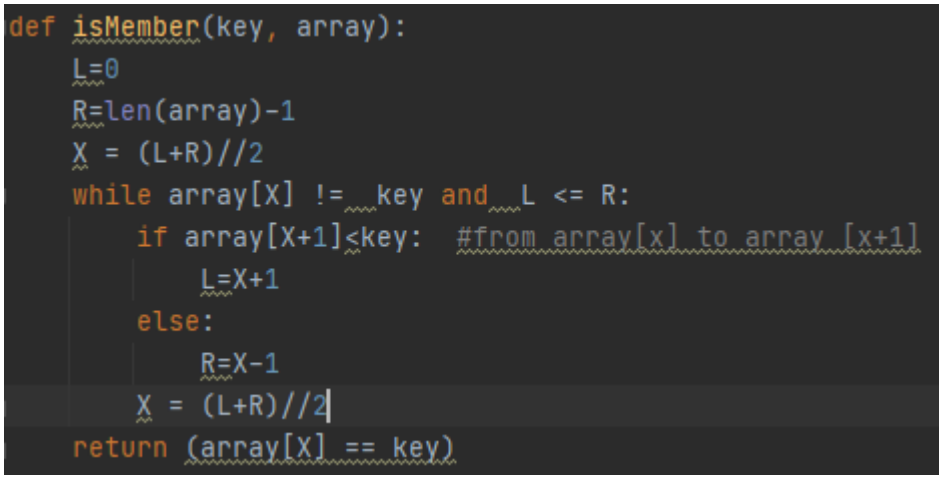
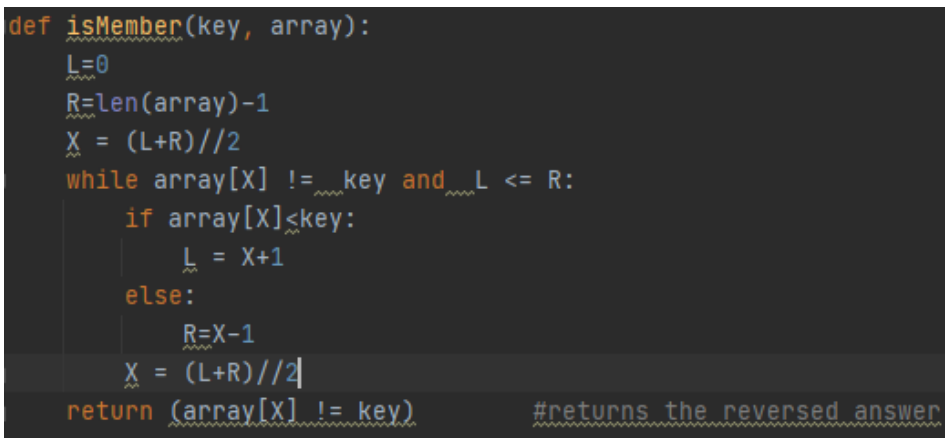
```
def test():
    key = 0
    LappCounter = 1
    k = True
    while k == True:
        randomarray = randomTest()
        sakjagforsoker = []
        for testings in randomarray:
            sakjagforsoker.append(testings)
        if key == 0:
            nummer = random.randint(0,7)
            key = randomarray[nummer]
            print("The key is " + str(key))
        CorrectAnswer = binarysortsearchmember(key,randomarray)
        MutationAnswer = binarysortsearchmemberone(key,sakjagforsoker)
        if CorrectAnswer == MutationAnswer:
            LappCounter += 1
        else:
            print("Result should give "+ str(CorrectAnswer) + " but gives " + str(MutationAnswer))
            print("Mutation is found on lapp " + str(LappCounter))
            break
```

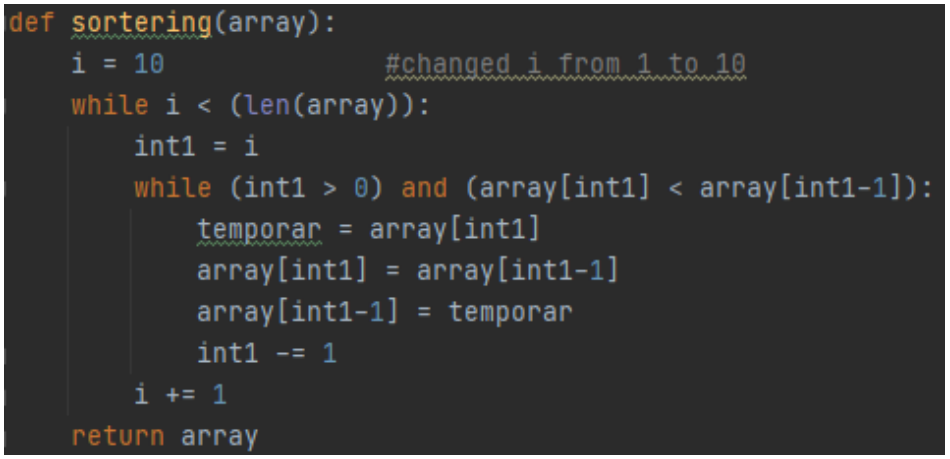
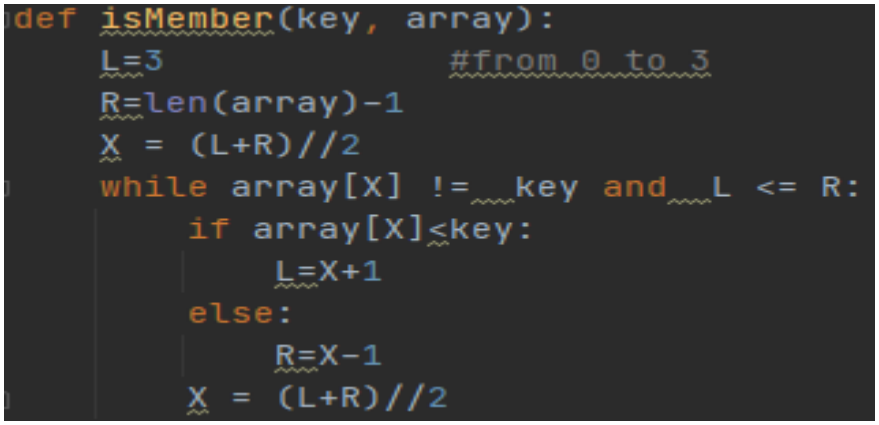
Here we see the pairwise code which also has a test code underneath that works in the same way as the one for randomwise.

```
def pairwisegenerator(Elements=8, highest=1000, lowest=1):
        defaults = []
        typicals = []
        testCases = []

        for NumberOfElements in range(Elements):
            defaults.append(randint(lowest, highest))
        while len(typicals) < len(defaults):
            NewTypical = randint(lowest, highest)
            if NewTypical != defaults[len(typicals)]:
                typicals.append(NewTypical)
        """0-wise"""
        testCases.append(defaults)
        """1-wise"""
        for element in range(Elements):
            newTestCase = defaults[0:element] + typicals[element:element + 1] + defaults[element + 1:]
            testCases.append(newTestCase)
        """2-wise"""
        for element1 in range(Elements):
            for element2 in range(element1 + 1, Elements):
                newTestCase = defaults[0:element1] + typicals[element1:element1 + 1] + \
                              defaults[element1 + 1:element2] + typicals[element2:element2 + 1]\
                              + defaults[element2 + 1:]
                testCases.append(newTestCase)
        return testCases
```

```
def test():
    LappCounter = 1
    k = True
    while k == True:
        gigaarray = pairwisegenerator()
        key = 0
        for elementarray in gigaarray:
            sakjagforsoker = []
            for testings in elementarray:
                sakjagforsoker.append(testings)
            if key == 0:
                nummer = random.randint(0,7)
                key = elementarray[nummer]
                print("The key is " + str(key))
            CorrectAnswer = binarysortsearchmember(key,elementarray)
            MutationAnswer = binarysortsearchmemberone(key,sakjagforsoker)
            if CorrectAnswer == MutationAnswer:
                LappCounter += 1
            else:
                print("Result should give "+ str(CorrectAnswer) + " but gives " + str(MutationAnswer))
                print("Mutation is found on lapp " + str(LappCounter))
                k = False
                break
```

Mutations	Randomwise	Pairwise
	5	1
	43	5
	38	2
	1	1

	115	38
	75	75

Here we can see that the pairwise tester sees the mutations faster in most of the cases. But because both of the testers have randomised array.

Mutation 1:

Here we changed the while loop so that it can never actually enter it. Therefore we should also always find the mutation in the first loop. This is the case for both of the methods.

Mutation 2:

Here we change so that we can't get into the loop which sorts the array. This doesn't give an immediate mutation that can always be seen. Since the arrays and keys are randomly selected, it might then take different amounts of loops to find the mutation. Here we can see that the pairwise found it faster than random wise.

Mutation 3:

Here we made the binary search move one step more than what it should. This isn't too big of a change, and we can see that the random wise had more problems finding the mutation then the pairwise one.

Mutation 4:

Here we changed the return of the membership code to invert the result. Both of these testers should find this mutation in the first run, and to no surprise we see that both of them do find it.

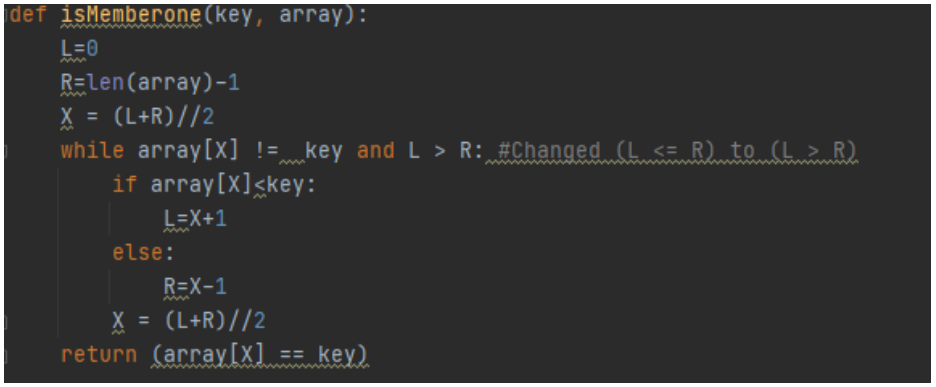
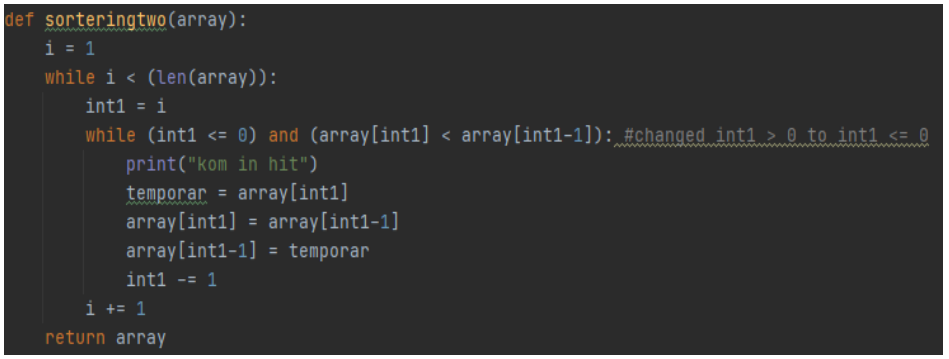
Mutation 5:

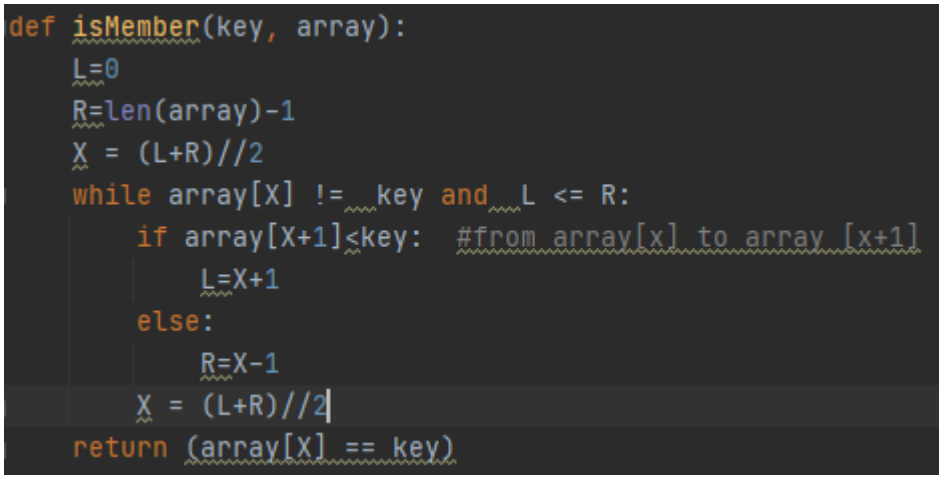
Here we change the incrementer to start at 10, this basically does so that we don't sort the last 10 elements. This should be easy to find when the array is short but a lot harder when we have a big array. Here we can see That both of the tests had a hard time finding the mutation.

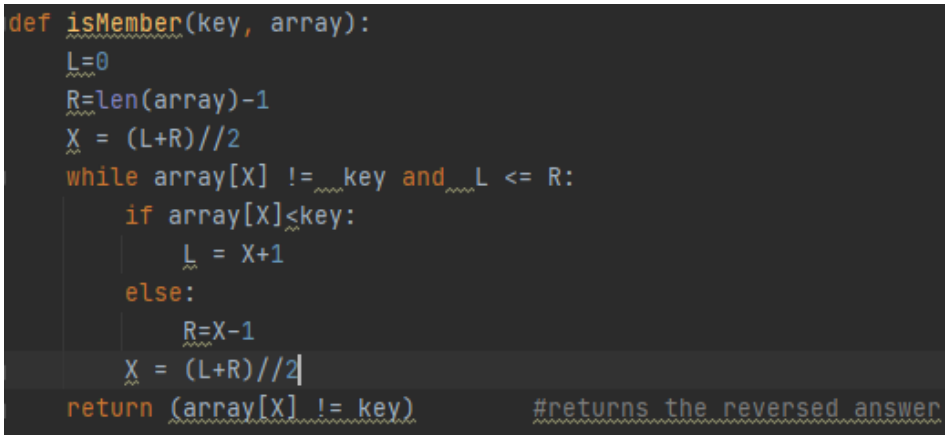
Mutation 6:

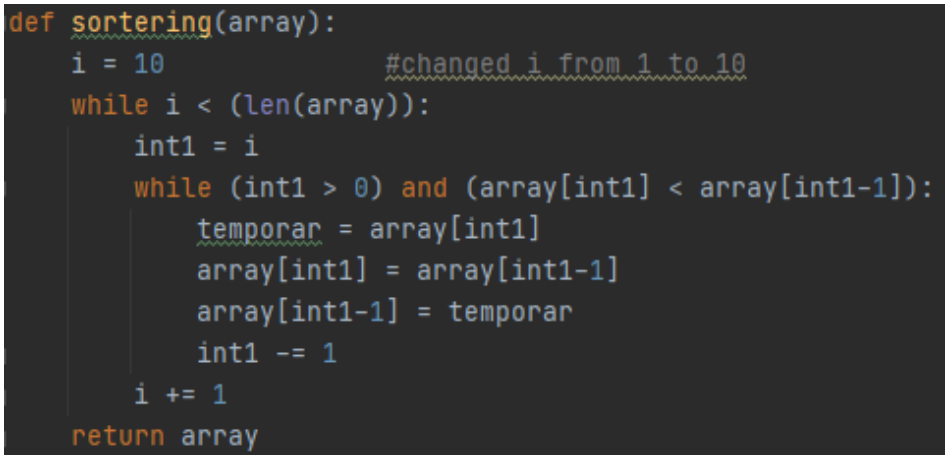
Here we make sure that you can't find the three first elements in the sorted array in the binary tree. This is a little like mutation 5 but in a different location in the code.

Now we increase the array size from 8 to 100 elements to see how each long it will take to find each mutation.

Mutations	Randomwise	Pairwise
	1	1
	2	1

 def isMember(key, array):
L=0
R=len(array)-1
X = (L+R)//2
while array[X] != key and L <= R:
if array[X+1]<key: #from array[x] to array [x+1]
L=X+1
else:
R=X-1
X = (L+R)//2
return (array[X] == key)
	6	3

 def isMember(key, array):
L=0
R=len(array)-1
X = (L+R)//2
while array[X] != key and L <= R:
if array[X]<key:
L = X+1
else:
R=X-1
X = (L+R)//2
return (array[X] != key) #returns the reversed answer
	1	1

 def sortering(array):
i = 10 #changed i from 1 to 10
while i < (len(array)):
int1 = i
while (int1 > 0) and (array[int1] < array[int1-1]):
temporar = array[int1]
array[int1] = array[int1-1]
array[int1-1] = temporar
int1 -= 1
i += 1
return array
	7	1

```
def isMember(key, array):
    L=3                 #from 0 to 3
    R=len(array)-1
    X = (L+R)//2
    while array[X] != key and L <= R:
        if array[X]<key:
            L=X+1
        else:
            R=X-1
        X = (L+R)//2
```

	------	------

When repeating the process with an array that is 100 long we see that both the pairwise and randomwise find the mutation after fewer loops. But with the increase in array size we also get a problem when trying to find the last mutation which didn't seem possible after letting the script run for several minutes without finding the error.

But there are also surprises where we thought that it would be alot harder to find mutation 5, but it seems that wasn't the case but that it didn't find mutation 6 on either of the tests after 30000 laps/runs.