# CITS5501 Software Testing and Quality Assurance Semester 1, 2020

# Workshop 3 (week 5) – sample solutions and discussion

## Exercise 1

a. Discuss how you would go about creating tests using Input Space Partitioning. What steps are involved? What is the input domain? And what characteristics and partitions would you use?

The steps are:

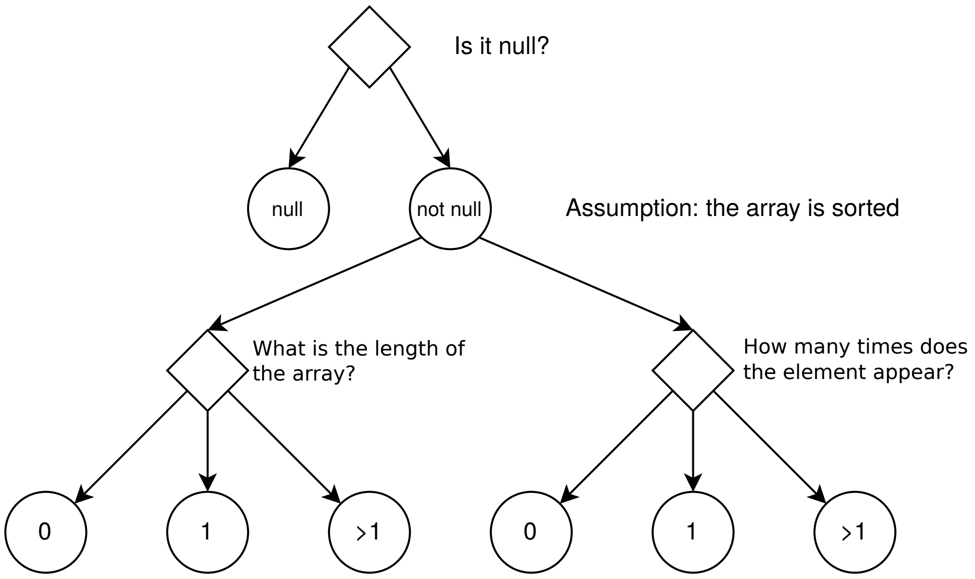
1. Identify testable functions
2. Identify all *parameters* to the functions
3. Model the input domain in terms of *characteristics*, each of which can be partitioned.
4. Choose particular partitions, and values from within those partitions
5. Refine into test values
6. Review

In this case, the "function" to be tested is the static method `static int binarySearch(char[] array, int startIndex, int endIndex, char value)`.
The function is static, so there is no "receiver object" it acts on. Since no global variables are mentioned in the specification, the parameters are simply:

- `char[] array`
- `int startIndex`
- `int endIndex`
- `char value`

The input domain for the function consists of all possible values of these parameters.

For the parameter `char[] array`, we might come up with the following possible characteristics and partitions:


Is it null?
null
not null
Assumption: the array is sorted
What is the length of the array?
How many times does the element appear?
0
1
>1
0
1
>1

When working out what characteristics to use, it can be handy to sketch these out in a diagram, like the one shown above.

Some things to note:

- If the array is *non-null*, then in our tests we will always *assume* that it is sorted. Why? Because if it is *not sorted*, the result of the method is *undefined* – there is literally nothing we can test for (not even an exception). So we must make the assumption that it is sorted (and take care that, in our tests, we always pass sorted arrays.)
- If the array *is* null, we might still have some tests involving it, based on characteristics we'll consider shortly. For instance: suppose `startIndex > endIndex`, *and* `array` is null – what exception will be thrown? The documentation says that the method should throw an `IllegalArgumentException`, so we might want to check that it indeed does so, even when `array` is null.
- Some combinations of partitions are impossible – e.g., if the length of the array is 0, then the element being looked for necessarily can't appear in it – but we don't worry about that at this stage.

For the `startIndex` parameter, one characteristic might be:

- Is `startIndex < 0`?

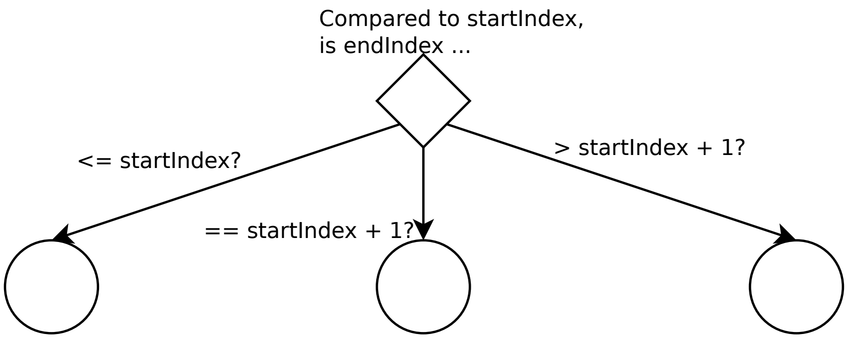
But I would probably consider `startIndex` and `array` together, and use the following characteristic with 3 partitions:

- Is `startIndex < 0`, or `>= array.length`, or between the two?

And then we could do the same for the `endIndex` parameter:

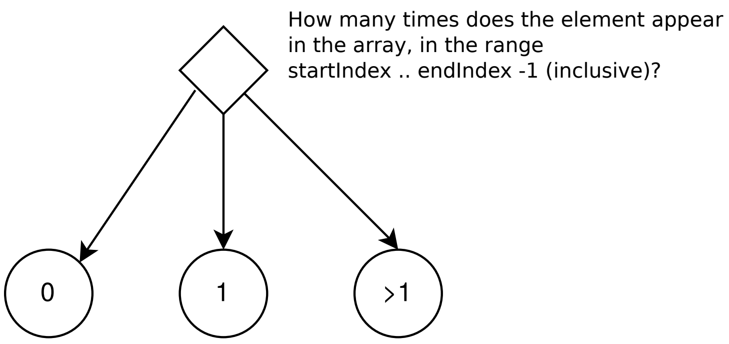
- Is `endIndex < 0`, or `>= array.length`, or between the two?

Since `startIndex` and `endIndex` represent bounds of a range (and since when `startIndex > endIndex` is a case mentioned by the documentation), we might consider those two together:



Why do we pick these partitions? Because they represent the cases where the range being searched is invalid, of size 0, and of size greater than 0, respectively. (We could also add in a partition for the case when the range is of size 1, if we liked.)

In addition to just how many times `value` appears in the *array*, we might want to consider how many times it appears in the cells with indexes from `startIndex` to `endIndex-1` (inclusive), since that's the portion of the array that will be searched. (This is a characteristic involving *four* parameters – `startIndex`, `endIndex`, `value` and `array`.)


How many times does the element appear in the array, in the range startIndex .. endIndex -1 (inclusive)?
0
1
>1

There are more characteristics we might consider (for instance: supposing the element being looked for is outside the bounds of `startIndex` and `endIndex`, is it to the left or the right?), but this will do for the moment.

b. List three different tests derived using this method.

To re-cap, our characteristics were:

- For `array`:
  - Is array null?
  - If not null,
    - Is its length: 0, 1 or >1?
- For `array` and `value`:
  - How many times does `value` appear in array: 0, 1 or >1?
- For `startIndex` and `array`:
  - Is it `<0`, `>= array.length`, or between the two?
- For `endIndex` and `array`:
  - Is it `<0`, `>= array.length`, or between the two?
- For `startIndex` and `endIndex`:
  - Is `endIndex <(startIndex+1)`, `==(startIndex+1)`, or `>(startIndex+1)`?
- For `startIndex`, `endIndex`, `array` and `value`:
  - How many times does the element appear in the range `startIndex ..`

`endIndex-1` (inclusive): 0, 1, or >1?

(This suggests if we were to write tests for all combinations of partitions we'd end up with $4 \times 3^5$ or 972 tests, which is clearly unrealistic. Some of those would be ruled out by not being feasible – e.g. if the element doesn't appear in the array at all, then it can't appear in the search range 1 time, for instance – but it's still likely to be far more than we would want.)

Assuming our characteristics do indeed define partitions, then every test case will involve some combination of those partitions.

We are not asked to tackle our test selection in any particular way, though, so we can just list three different test cases using test values from whatever partitions we like. The worksheet also doesn't specify any particular format with which to describe our tests, but we know at minimum we must give inputs and expected results. And it doesn't hurt to give a description.

- **Description:** Behaviour of `binarySearch` when `endIndex < startIndex`, and array is null.
  **Inputs:** array = null, startIndex = 4, endIndex = 3, value = 'a'
  **Expected result:** need to clarify, see below.

  You may have noticed the spec is contradictory in this case, since it says we should throw an `IllegalArgumentException` (since `endIndex < startIndex`), but also an `ArrayIndexOutOfBoundsException` (since both are outside the bounds of the array). We would need to go back to the designer of binarySearch and clarify what the intended behaviour is.

- **Description:** Behaviour of `binarySearch` when array is of size 1, the element does not appear in it, and `endIndex==(startIndex+1)`.
  **Inputs:** array = `{'a'}`, startIndex = 0, endIndex = 1, value = 'b'
  **Expected result:** -2 (since the new element would be inserted at position 1, and -1 – 1 = -2).

- **Description:** Behaviour of `binarySearch` when array is of size >1, the element appears in it twice, `endIndex>(startIndex+1)`, and the element appears in that range once.
  **Inputs:** array = `{'a', 'b', 'c', 'c'}`, startIndex = 0, endIndex = 3, value = 'c'
  **Expected result:** 2

c. Discuss how you would assess whether a set of tests have *base choice* coverage. What would you use for base choices?

To do this, we need to decide, for each characteristic, what we're going to use as our *base choice*, and we should document the reason we chose it.

The lecture slides tell us that we might specify a particular base choice because it is:

- most likely
- simplest
- smallest
- first in some order

We might settle on the following base choices:

- Is array null or non-null?
  - We choose non-null, as being most likely, from an end-use point of view.
- Is the array of length 0, 1 or >1?
  - We choose >1, again as being most likely.
- How many times does the value appear in the array: 0, 1 or >1?
  - We choose >1, as being a likely case – many times, lists contain repeated elements.
- Is `startIndex<0`, or `>=array.length`, or between the two?
  - We choose "between the two", as being most likely.
- Is `endIndex<0`, or `>=array.length`, or between the two?
  - We choose "between the two", as being most likely.
- Is `endIndex <(startIndex+1)`, or `==(startIndex+1)`, or `>(startIndex+1)`?
  - We choose `>(startIndex+1)`, as being most likely.
- How many times does the element appear in the range `startIndex .. endIndex-1` (inclusive): 0, 1, or >1?
  - We choose ">1", as being most likely.

Note that if we're concerned this coverage criterion might not be stringent enough, and might let some edge cases slip through the crack, we're always at liberty to use *Multiple* Base Choice. For instance, for array length, we might select 1 *and* >1 as being base choices, or even all three partitions.

If we use Base Choice as a coverage criterion, then it will be satisfied if we have:

- A test in which the base choice is selected, for all characteristics
- For each characteristic *c*: a test in which all *other* characteristics are held constant, and *c*, the characteristic we're looking at, is varied.

A quick count suggests the number of tests will be 1 (the base choice for all) plus

- 1 (for null array)
- 2 (array of length 0 and array of length 1)
- 2 (values appears 0 times and 1 time)
- 2 (`startIndex < 0` and `startIndex >= array.length`)
- 2 (`endIndex < 0` and `endIndex >= array.length`)
- 2 (`endIndex <(startIndex+1)` and `==(startIndex+1)`)
- 2 (element appears in the searched range 0 times, and 1 time)

for a total of 14 tests – much more tractable than 972 tests. (Again, some combinations can be removed because they're not feasible – but the number of tests will certainly be no *more* than 14.)

## Exercise 2

```java
public static int binarySearch(char[] array, int startIndex, int endIndex, char value) {
    if (startIndex > endIndex) {
        throw new IllegalArgumentException();
    }
    if (startIndex < 0 || endIndex > array.length) {
        throw new ArrayIndexOutOfBoundsException();
    }

    int lo = startIndex;
    int hi = endIndex - 1;
    while (lo <= hi) {
        int mid = (lo + hi) / 2;
        char midVal = array[mid];
        if (midVal < value) {
            lo = mid + 1;
        } else if (midVal > value) {
            hi = mid - 1;
        } else {
            return mid;  // value found
        }
    }
    return lo * -1;  // value not present
}
```

a. Discuss how you would construct a control-flow graph for the method. Try drawing the graph, stating any simplifying assumptions you need to make.

We would construct the control-flow graph by converting each basic control-flow structure (if, while and so on) into a graph as shown in the lecture slides.

To simplify the graph – given that when an exception is thrown, we don't know *where* execution will resume – we will assume that it goes to a fictional location, which we'll call "ExHandler".

We should end up with a diagram like the one following:

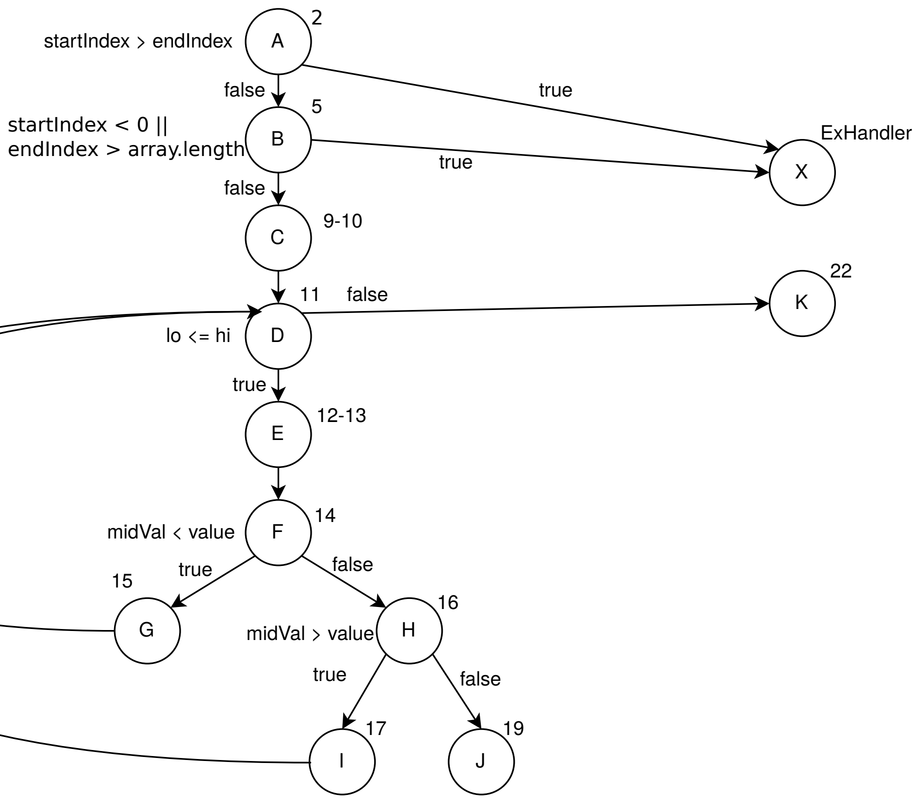


Figure 1: Control-flow graph for `binarySearch`

b. See if you can identify prime paths in which the loop is executed:

- zero times
- once
- more than once

For reference, the prime paths are:

| | | |
|---|---|---|
| AX | DEFHID | GDEFG |
| ABX | EFGDE | GDEFHI |
| ABCDK | EFGDK | GDEFHJ |
| ABCDEFG | EFHIDE | HIDEFG |
| ABCDEFHI | EFHIDK | HIDEFH |
| ABCDEFHJ | FGDEF | IDEFHI |
| DEFGD | FHIDEF | IDEFHJ |

Zero times: the paths AX, ABX and ABCDK are all prime paths, and all execute the loop body exactly zero times.

One time: the paths ABCDEFHJ, EFGDK and EFHIDK are all prime paths, and all traverse the loop body only once, so they meet our criterion.

More than once: paths which definitely execute the loop body more than once are:

| | |
|---|---|
| EFGDE | GDEFHJ |
| EFHIDE | HIDEFG |
| FGDEF | HIDEFH |
| FHIDEF | IDEFHI |
| GDEFG | IDEFHJ |
| GDEFHI | |

For the remaining prime paths: it’s not clear whether we can say which of these categories they fall into. (The paths are ABCDEFG, ABCDEFHI, DEFGD, and DEFHID.) They traverse the nodes making up the loop body once, but if we actually executed the code so as to traverse these paths, we might end up executing the loop body another time – it would depend on the value of `lo <= hi` once the program got to node D for a second time.

c. See if you can construct test cases for these prime paths.

Let’s suppose we selected the following prime paths for part (b):

- AX (zero times)
- EFGDK (once)
- EFGDE (more than once)

For a test case which traverses the path AX, we just need `startIndex > endIndex`.

So a possible test case might be:

- **Description:** Test case to execute path AX in Figure 1
  **Inputs:** array = `{’a’, ’b’}`, startIndex = 1, endIndex = 0, value = ‘b’
  **Expected result:** Throws `IllegalArgumentException`

For a test case which traverses the path EFGDK, we need to come up with inputs such that `lo <= hi` is true the first time round, but will become false after. If you experiment with the code, you will see that this happens when `hi = lo + 1`, and the searched-for value is not found.

So a test case for this path could be:

- **Description:** Test case to execute path EFGDK in Figure 1
  **Inputs:** array = `{’a’}`, startIndex = 0, endIndex = 1, value = ‘b’
  **Expected result:** Returns -1 (since considering the “slice” of the array from position 0 to position 0 inclusive: if the value “b” is not found there, then the place to insert it is at position 0, and (-1 × 0) – 1 = -1.)

(Try executing the code “by hand”. Before entering the loop, you should get that `lo` and `hi` both equal 0.
Then `mid = (0 + 0) / 2` (which is 0), `midVal = 'a'`, and the the new values of `lo` and `hi` are 1 and 0, respectively.)

For a test case which traverses the path EFGDE, we need to come up with inputs that get us to node E (so `lo <= hi` needs to be true) *and* then goes through the loop again (so `lo <= hi` the second time). If you experiment with the code, you will see that we can make this happen when `hi = lo + 2`, and the searched-for value is not found the first time through the loop.

So here is a possible test case:

- **Description:** Test case to execute path EFGDE in Figure 1
  **Inputs:** array = `{'a', 'b'}`, startIndex = 0, endIndex = 2, value = 'c'
  **Expected result:** Returns -3 (since considering the "slice" of the array from position 0 to position 1 inclusive: if the value "c" is not found there, then the place to insert it would be at position 2, and $(-1 \times 2) - 1 =$ -3.)

Some further notes on the `binarySearch` method:

- If you try executing the above tests on the implementation that has been given, you'll see that it doesn't give correct results. Can you work out what the fault is?
- Binary search is notoriously difficult to code correctly, despite the fact that a basic implementation (without exception throwing) takes only a dozen lines or so.

  In 1986, Jon Bentley in his book *Programming Pearls* wrote that (based on his experience teaching courses to professional programmers) 90% of programmers implement binary search incorrectly, even when given several hours to do it, and he presents a version which he then uses program verification techniques to "prove" correct.

  But in fact, there is a subtle bug in the code Bentley presents. Google software engineer Joshua Bloch, who wrote the Java implementation of binary search, and used Bentley's code as a basis for it, wrote about the bug's 2006 discovery (see his blog post, at googleblog.com): the bug had gone unnoticed for 20 years.

  In our code listing, the bug appears in line 12 (the line with the statement `int mid = (lo + hi) / 2`). The `binarySearch` method will return an incorrect result when `lo + hi` is greater than $2^{31}$ (more details are given by blogger Mohit Chawla here). As Bloch says in his blog post, "This bug can manifest itself for arrays whose length (in elements) is $2^{30}$ or greater (roughly a billion elements). This was inconceivable back in the '80s, when *Programming Pearls* was written, but it is common these days at Google and other places."

  So although I have said that *most* of us need not often be concerned with the upper and lower boundaries of data types like `int`, it is certainly not always the case that they can be ignored.