luigi cussigh [2023148006]

Data Structures Completion Points Assignment 1

Implementation 1.

```
public static int search(int n, int[] a, int x) {
    if (n == 0)
        return -1;
    int left = 0;
    int right = n - 1;
    while (left < right) {
        int mid = (left + right) / 2;
        if (x > a[mid])
            right = mid - 1;
        else
            left = mid;
    }
    if (a[left] == x)
        return left;
    else
        return -1;
}
```

This implementation is incorrect. We can make sure that is the case by doing a dry run with these values:

```
int n = 2;
int[] arr = new int[] { 1, 0 };
int x = 0;
```

During the first iteration of the while loop we end up setting mid to 0. x is less than a[mid] so the else block is executed and left is set to 0 (unchanged). Thus, the program will never terminate, because no variable was changed during the first iteration.

Implementation 2.

```java
public static int search(int n, int[] a, int x) {
    int left = 0;
    int right = n;
    while (left + 1 < right) {
        int mid = (left + right) / 2;
        if (a[mid] >= x)
            left = mid;
        else
            right = mid;
    }
    if (left + 1 == right && a[left] == x)
        return left;
    else
        return -1;
}
```

Essentially, we changed the inclusivity of the midpoint with this implementation. When we change the inclusivity of the midpoint in a binary search algorithm, we need to modify the midpoint calculation process too. Instead of rounding down to an integer, we need to round up. That way, when we reach the subarray of length, we don't get stuck in an infinite loop.

And in this implementation, we did change the rounding to round up. But instead of changing the midpoint calculation process, we changed how we handle the variable right. In this implementation essentially right is made bigger by one than its actual value, that way when we calculate the midpoint we get a round-up value.

But this messes up other parts of the code, like the while loop condition or the right bound setting/initialization process.

So, whoever wrote this code had to change the code in such a way that we always pretend that right is smaller than it actually is: for example, the condition left + 1 < right can be rewritten as: left < right + 1, essentially compensating the increased R value.

This also affected the line right = mid; a more conventional implementation of the algorithm would have this line of code replaced by right = mid – 1. Once again, we see that right in this implementation needs to be bigger by one than its supposed value, resulting in this peculiar code to be written.

In conclusion, even though the code is weird, it works.

Implementation 3.

```java
public static int search(int n, int[] a, int x) {
    if (n == 0)
        return -1;
    int left = 0;
    int right = n - 1;
    while (right - left > 0) {
        int mid = (left + right) / 2;
        if (a[mid] == x) {
            left = mid;
            break;
        }
        if (a[mid] > x)
            left = mid + 1;
        else
            right = mid - 1;
    }
    if (a[left] == x)
        return left;
    else
        return -1;
}
```

This implementation is correct. It is basically the same as the most commonly used binary search algorithm, but with a slight optimization, checking whether the value on the midpoint is equal to the value we are looking for. The only change made to the rest of the code is the setting of the right bound. Since a[mid] cannot be equal to x by the time we get to the point we have to set the right bound (if it was, we would’ve broken out of the loop), we don’t need to include the midpoint in the checking interval, thus we set the right pointer to mid – 1 instead of mid + 1.

Implementation 4.

```
public static int search(int n, int[] a, int x) {
    if (n == 0)
        return -1;
    int left = 0;
    int right = n - 1;
    while (right - left > 1) {
        int mid = (left + right + 1) / 2;
        if (a[mid] == x) {
            left = mid;
            right = mid;
            break;
        }
        if (a[mid] > x)
            left = mid + 1;
        else
            right = mid - 1;
    }
    if (a[right] == x)
        return right;
    else
        return -1;
}
```

This implementation is incorrect. It differs slightly differs from the previous implementation: it rounds up when calculating the midpoint and also, the loop condition is modified, to break out when the subarray becomes of size two or less. This results in the program not working correctly when initial arrays of length 2 are given as input:

```
int n = 2;
int[] arr = new int[] { 1, 0 };
int x = 1;
```

These inputs, for example, demonstrate the program working incorrectly. Since left is set to 0 and right is set to 1 initially, we never enter the while loop and jump straight to the final if statement, where we check whether the element which is pointed by the right pointer is equal to x, which is not the case with these inputs. The program doesn’t check the first element in the array and comes to wrong conclusion that the value doesn’t exist in the array, which is clearly not true.