Lab2: Black-box (2023)

Mimmi Lindgren, Katrina Liang

February 2023

1 Condensation Graph

A condensation graph for the binary search algorithm:

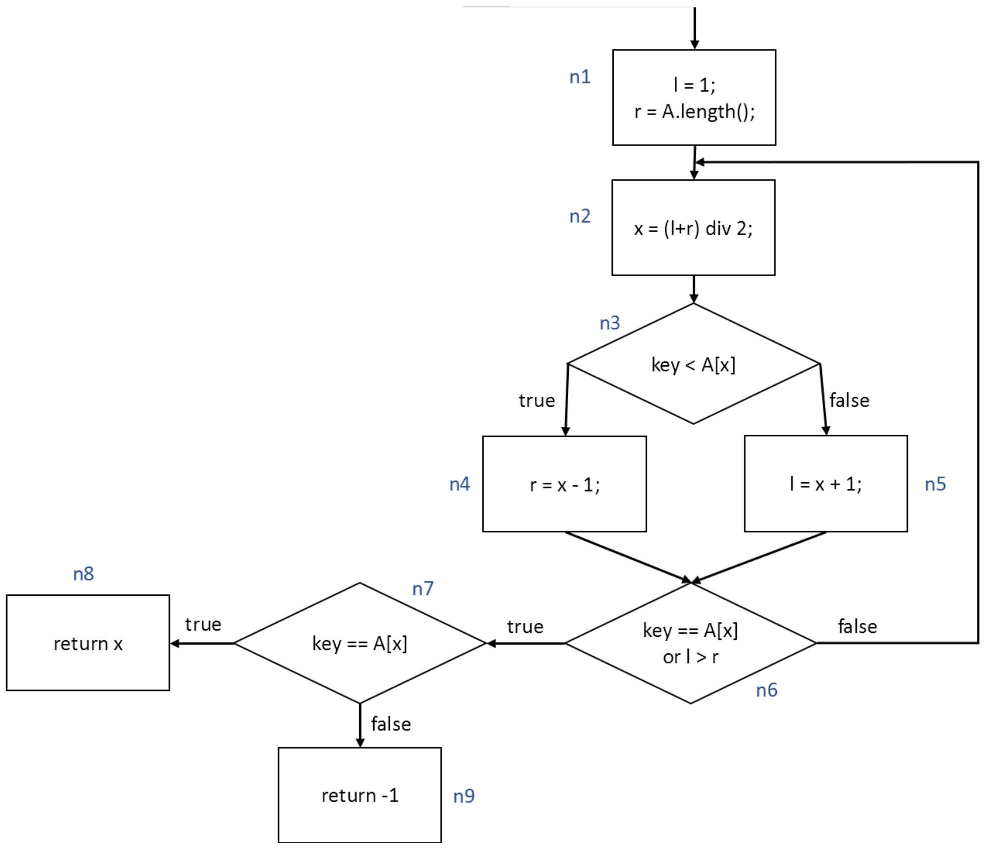

Figure 1: Condensation graph for Algorithm 2.

2 JML

2.1 Sorting

```
//@ requires A != null && 0 < A.length;
//@ ensures \forall int i; 0 <= i < A.length-1; A[i+1] => A[i];
//@ ensures A.length == \old(A).length;
```

If we want to be able to use an empty array. (A weaker precondition.)
Added a stronger post-condition. Karl's version:

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

2.2 Searching

```
//@ requires A != null && 0 < A.length && key != null;
//@ old boolean containsKey (\exists int i; 0 <= i < A.length; A[i] == key);
//@ ensures containsKey <==> 0 <= \result < A.length;
//@ ensures !containsKey <==> \result == -1;
```

Weaker precondition with empty array as a possibility. Karl's version:

```
//@ requires A != null;
//@ ensures (\exists int i; 0 <= i < \old(A).length; key == \old(A)[i])
// => \old(A)[\result] == key
// & A.lnegth == 0 | (!\exists int i; 0 <= i < \old(A).length; key == \old(A)[i])
// => \result == -1;
```

2.3 Membership

```
//@ requires A != null && 0 < A.length && key != null;
//@ old boolean containsKey (\exists int i; 0 <= i < A.length; A[i] == key);
//@ ensures containsKey <==> \result == 1;
//@ ensures !containsKey <==> \result == 0;
```

Weaker precondition, and stronger post-condition. Karl's version:

```
//@ requires A != null;
//@ ensures (\exists int i; 0 <= i < \old(A).length; key == \old(A)[i])
// => \result == 1
// & A.length == 0 | (!\exists int i; 0 <= i < \old(A).length; key == \old(A)[i])
// => \result == 0;
```

2.4 Binary search

```
//@ requires A != null && 0 < A.length && key != null;
//@ requires \forall int i; 0 <= i < A.length-1; A[i+1] > A[i];
//@ old boolean containsKey (\exists int i; 0 <= i < A.length; A[i] == key);
//@ ensures containsKey <==> 0 <= \result < A.length;
//@ ensures !containsKey <==> \result == -1;
```

Weaker precondition, after the seminar:

```
//@ requires A != null;
//@ requires \forall int i; 0 <= i < A.length-1; A[i+1] > A[i];
//@ old boolean containsKey (\exists int i; 0 <= i < A.length; A[i] == key);
//@ ensures containsKey <==> 0 <= \result < A.length;
//@ ensures !containsKey <==> \result == -1;
```

3 Implementation in Python

```python
def merge_sort(arr):
    if len(arr) > 1:
        mid = len(arr) // 2
        l_arr = arr[:mid]
        r_arr = arr[mid:]
        merge_sort(l_arr)
        merge_sort(r_arr)

        i = j = k = 0

        while i < len(l_arr) and j < len(r_arr):
            if l_arr[i] < r_arr[j]:
                arr[k] = l_arr[i]
                i += 1
            else:
                arr[k] = r_arr[j]
                j += 1
            k += 1

        while i < len(l_arr):
            arr[k] = l_arr[i]
            i += 1
            k += 1

        while j < len(r_arr):
            arr[k] = r_arr[j]
            j += 1
            k += 1

    return arr
```

Listing 1: mergeSort to sort an integer array of arbitrary length.

```python
def binary_search(arr, key):
    l = 0
    r = len(arr) - 1

    while True:
        mid = (r + l) // 2
        if arr[mid] < key:
            l = mid + 1
        elif arr[mid] > key:
            r = mid - 1
        if l > r or (arr[mid] == key):
            break

    if key == arr[mid]:
        return mid
    else:
        return -1
```

Listing 2: Membership queries on sorted arrays of arbitrary length using binary search.

```python
from sorting import merge_sort
from binary_search import binary_search


def check_membership(arr, key):
    sorted_array = merge_sort(arr)
    result = binary_search(sorted_array, key)
    if result == -1:
        return 0
    else:
        return 1
```

Listing 3: Membership queries on unsorted arrays of arbitrary length using merge sort and binary search.

4 Random and Pairwise Testing

We consider the array as N numbers of input data, to be able to make more test cases in pairwise testing. Otherwise, we will only get one test case with the array as one input and the key as the other. (This could be tested by random testing.) The array size was 7 throughout the testing. The randomness implemented in this experiment had a boundary [-100, 100] for array values, and [-200, 200] for the key. By having a larger boundary for the key, we could guarantee that there will be negative test cases where the key is not a member of the array.

For pairwise testing, we choose the default value of each index in the input array to be [7, 6, 5, 4, 3, 2, 1], and the default key 1.

4.1

Mutation	Random (average of 100 runs)	Pairwise
> to <	98.04	1 (default values)
”and” to ”or”	1 (exception)	1
< to >	65.0	1
+= 1 to += 2	1683.53	Error not found (all 37 passed)
+= 1 to = 1	1.06 (exception)	1 (exception)
+= 1 to -= 1	1.03 (exception)	1 (exception)

Table 1: Mutation injection of error in merge sort listing: 1

4.2

6 errors were injected to program 1, merge sort.

1. Change > to < in line 2
2. Change ”and” to ”or” in line 11
3. Change < to > in line 12
4. Change += 1 to += 2 in line 14
5. Change += 1 to = 1 in line 17
6. Change += 1 to -= 1 in line 22

4.3

Because we had set the default values in the pairwise testing to [7, 6, 5, 4, 3, 2, 1], and the key to 1, the first test found all the errors except for mutation 4 on line 14, where it was not able to detect an error at all. When we checked why, the only one of the tests that could have failed got the key sorted to the middle and therefore the binary search could find it right away.

In some cases (mutation 1 och 3), pairwise testing could find an error or wrong answer with fewer tests than random testing. For mutation 4, if we have done an assertion for each method, we could have found an earlier false result, even though the final result happens to be right coincidentally. With random testing, a wrong answer was found. but it took over one thousand tests on average. Such a large number of tests would not be feasible if the execution time is much longer.

4.4

The above experiment was repeated with an array size of 100.

For random testing, the number of test cases it took to find an error was much smaller than in the previous experiment when the array size was 7. In pairwise testing, the error in mutation 4 could now be found, since the number of test cases increased.

Mutation	Random (average of 100 runs)	Pairwise
> to <	5.49	1 (default values)
”and” to ”or”	1 (exception)	1 (exception)
< to >	5.99	1
+= 1 to += 2	13.39	4907
+= 1 to = 1	1 (exception)	1 (exception)
+= 1 to -= 1	1 (exception)	1 (exception)

Table 2: Mutation injection of error in merge sort listing: 1