Example Case Study: Binary Search in a Sorted Array

Introduction

This lab provides a complete example of a simple case study. Use this as a template and reference for your own case study assignment. Note that this case study is at a smaller scale than what is expected for your case study.

Case study: Binary search in an array

Part 1. Informal Specification

Problem Description

Implement a binary search algorithm that finds the index of a target element in a sorted array of integers.

Input

- `arr`: An array of integers sorted in ascending order
- `target`: An integer value to search for

Output

- An index `i` where `arr[i] == target`, if the target exists in the array
- `-1` if the target does not exist in the array

Informal Constraints

- The input array must be sorted in ascending order
- The array may be empty
- The array may contain duplicate values (in which case, any valid index is acceptable)

Informal Desired Properties

1. **Correctness (when the value is found)**: If the algorithm returns an index `i ≥ 0`, then `arr[i]` must equal the target
2. **Correctness (when the value is not present)**: If the algorithm returns `-1`, then the target does not exist anywhere in the array
3. **Completeness**: If the target exists in the array, the algorithm must find it (not return `-1`)
4. **Efficiency**: The algorithm should run in O(log n) time

5. **Safety**: The algorithm should never access array indices out of bounds

---

Part 2. Formal Specification

Algorithm description

The binary search algorithm maintains a search window defined by indices `low` and `high`, repeatedly narrowing this window by comparing the middle element with the target.

Preconditions

```
isSorted(arr)
```

Where `isSorted` is defined as:

```
forall i, j :: 0 <= i < j < arr.Length ==> arr[i] <= arr[j]
```

Explanation: The array must be sorted in ascending order. This is essential for binary search correctness.

Postconditions

The `target` is the information to search in the array `arr`. The following should be true:

```
1. 0 <= result < arr.Length ==> arr[result] == target
2. result == -1 ==> forall k :: 0 <= k < arr.Length ==> arr[k] != target
```

Explanation: - First postcondition: If we return a valid index, that index contains the `target` (Correctness - when the value is found) - Second postcondition: If we return -1, the target doesn’t exist in the array (Correctness - when the value is not present)

> Exercise: formulate the rest of the postconditions (3,4, and 5) from above. Are they covered by the above formal properties or more are needed?

Loop Invariants

Assuming that the algorithm maintains a search window defined by indices `low` and `high`, repeatedly narrowing this window by comparing the middle element with the target, the main challenge in verifying binary search is proving the loop invariants. Here is a proposal of some invariants that are meant to remain valid through this repetitive process:

```
invariant 0 <= low <= high + 1 <= arr.Length
invariant forall k :: 0 <= k < low ==> arr[k] < target
invariant forall k :: high < k < arr.Length ==> arr[k] > target
```

Explanation: 1. **Bounds invariant**: Ensures `low` and `high` stay within valid bounds and maintain the proper relationship 2. **Left exclusion invariant**: All elements before `low` are strictly less than the target - we’ve eliminated them from consideration 3. **Right exclusion invariant**: All elements after `high` are strictly greater than the target - we’ve eliminated them too

These invariants together ensure that if the target exists, it must be in the range `[low, high]`.

Correspondence of Formal Properties to Informal Properties

Informal Property	Formal Specification
Correctness (Found)	`ensures 0 <= result < arr.Length ==> arr[result] == target`
Correctness (Not found)	`ensures result == -1 ==> forall k :: 0 <= k < arr.Length ==> arr[k] != target`
Completeness	Combination of both postconditions (if target exists, can’t return -1)
Safety	Loop invariant `0 <= low <= high + 1 <= arr.Length` + bounds checks

Exercise: how about **Efficiency**. Can we do that?

Part 3. Implementation and Verification in Dafny

Exercise: provide a full implementation and verification of the proposed algorithm above.

```
TBA
```

Part 4. Discussion

What Was Easy?

1. **Defining the informal specification**: Binary search is a well-known algorithm, so describing the inputs, outputs, and desired behavior was straightforward.

2. **Basic postconditions**: The postconditions directly capture the obvious correctness properties - if we return an index, it should contain the target.

3. **Implementation structure**: The algorithm itself is simple and classic, with clear logic for narrowing the search window.

What Was Difficult?

1. **Loop invariants**: The most challenging aspect was identifying and proving the loop invariants. The key insight was recognizing that we maintain the property that all elements outside the `[low, high]` range have been eliminated from consideration.

2. **Handling edge cases**: Ensuring the invariants hold for empty arrays and single-element arrays required careful attention to boundary conditions. The invariant `0 <= low <= high + 1` (rather than just `low <= high`) was crucial for handling the case when the loop exits.

3. **Initial attempts with strict inequality**: Initially, I tried using `arr[k] <= target` and `arr[k] >= target` in the invariants, but this didn’t work because the target might not exist. Using strict inequalities (`<` and `>`) was the key to making the proof work.

What We Didn’t Accomplish

1. **Formal verification of time complexity**: While we know binary search is O(log n), Dafny doesn’t directly verify time complexity. We used a `decreases` clause to prove termination, but not the logarithmic time bound.

2. **Proving we return the first/last occurrence**: In case of duplicates, our specification doesn't guarantee which occurrence we return. We could strengthen the specification to require the first or last occurrence, but this would complicate the invariants significantly.

What We Had to Change

These below are just examples, to illustrate what could be inserted in this section.

1. **Overflow handling**: The naive `mid := (low + high) / 2` could overflow. We changed to `mid := low + (high - low) / 2` to prevent this issue (though Dafny's arbitrary-precision integers make this less critical than in languages like C).
2. **Invariant refinement**: Through several iterations, we refined the invariants to their current form. Early versions had weaker invariants that weren't sufficient to prove the postconditions.

Key Findings and Insights

1. **Invariants are the contract with the loop**: The loop invariants serve as the "contract" that each iteration must maintain. They bridge the gap between what we know before the loop and what we need to prove after it.
2. **Exclusion properties are powerful**: Rather than trying to maintain "the target is in `[low, high]`," it's easier to maintain "the target is NOT outside `[low, high]`" using strict inequalities.
3. **Preconditions enable simpler invariants**: The `isSorted` precondition is essential. Without it, we couldn't prove the exclusion invariants, since moving `low` or `high` wouldn't guarantee we're not skipping over the target.
4. **Dafny's automation is impressive**: Once the correct invariants were specified, Dafny automatically verified the implementation without additional assertions or lemmas.

Conclusion

This case study demonstrates the complete process of formal verification: starting with an informal understanding, formalizing the properties, implementing with verification, and reflecting on the process. The key lesson is that formal verification forces us to think precisely about our assumptions (preconditions) and guarantees (postconditions and invariants), often revealing subtle aspects of algorithms we thought we understood completely.