CS 1110: Introduction to Computer Science

Assignment 5: Sorting and Searching

Author: Vaani Goenka
Date: 5th June 2025

Problem 1

Specifications

The given problem asks us to determine whether a specified element (referred to as the target `t`) exists within a provided list of elements. Naively, we would examine each element, comparing it with `t`. However, to enhance efficiency, we can utilize the fact that if the array is sorted, we can compare the size of elements with `t`. At each stage, we can ascertain whether `t` is greater than or less than the current element being examined. This allows us to decide whether to search in the left or right portion of the array, effectively eliminating many elements from consideration with each check. This is the essence of the binary search algorithm.

Steps

1. We want to pick an element to begin the comparison. The question arises: What should this number be? But we don't know what numbers are there in the list, so we ask a different question: where should this number be? Consider that we take it in some index towards the beginning of the array, and the element - let's call it `e` from now - is lesser than `t`. That means, `t` can definitely not be among the elements to the left of `e` because that would mean the sorted order is wrong! So `t` has to be in the right part. Now consider that we took `e` towards the left of the array, so we still have at least more than half elements to look at. So this kind of am 'imbalanced' divide does not guarantee that we always have the smallest possible division of the array. To fix this, we always take `e` to be the middle element of the array. Let's define:

```
mid = (first index + last index)/2
```

2. So now we have a comparison element to begin with. Depending on whether `arr[mid]` is less than or greater than `t`, we want to either look at the left elements of `mid`, or its right elements. In other words:

```
if (t > arr[mid]) -> consider right elements
 if (t < arr[mid]) -> consider left elements
```

3. But, `t` can also be equal to `mid`. In fact, if that happens, we have indeed found the element in the array! So let's add that cas:

```
if (t = arr[mid]) -> found element!
```

4. Now, we also have to consider what we do when we do find the element - the question says we want

the location of the element. So, we output the return. For a function, the notion of the 'output' is given by the return value, so in other words, we write:

```
if (t = mid) -> return mid
```

5. But it could also so happen that the element we want is not in the array. We have to decide when to stop searching - when can we conclude that the element `t` is definitely not in the array? It is precisely when we know that none of the elements can be equal to the array (maybe they are all lesser or all greater - we have only these two other options). So, when, after successive comparisons, we reach a point where we look at only one element (`first index = last index`), then we have no left or right to go to, in other words, after this updating, `first index > last index`.

Code

We begin by simply declaring a function. Now we must decide what our function 'operates' on, or in other words, what is the input to the function. We know that we need the list of elements and the target element (obviously), but we also want to know what elements we are 'consdering' so let's keep a start and end to know our array under consideration.

```
def binary_search (l : int list)(t : int)(start : int)(end : int) : int =
```

Now, before trying to compute the middle element, we need to be sure that such an element exists in the first place! From step 5,

```
match start > end with:
| true -> -1 | false -> (* We can proceed! *)
```

Now we need to know what the middle element is. From step 1 and using the helper function (given at the end) we get the required `mid` element.

```
let midIndex = (start + end)/2 in
let mid = l.getElem(midIndex) in
```

Let's perform the checks. We are essentially concerned with the truth value of whether `mid > t`, so we match on those - because this is exactly the 'result' that defines what next step we must take. From steps 2, 3 & 4

```
match mid > t with
| true -> binary_search l t start (midIndex-1)
| false where mid = t -> midIndex
| false -> binary_search l t (midIndex+1) end
```

Here is the complete code:

```
def binary_search (l : int list)(t : int)(start : int)(end : int) : int
=
    match start > end with:
    | true -> -1
    | false ->
        let midIndex = (start + end)/2 in
        let mid = l.getElem(midIndex) in
            match mid > t with
            | true -> binary_search l t start (midIndex-1)
            | false where mid = t -> midIndex
            | false -> binary_search l t (midIndex+1) end
```

> Proof
>
> To prove that the function works correctly, we use induction on the size of the search space, (end-start).

Begin Induction

> **Base Case**
>
> l = []
> start = 0
> end = -1
> t = any element
> By line 3 the function returns -1, which is correct since the list is empty and the target cannot be found.
>
> **Induction Hypothesis**
>
> Assume that the function works for all search spaces of size `k` or less.
>
> **Induction Step**
>
> 1. By line 9, If `l[mid] = t`, the target is found, and its index mid is returned. This is a base case of the recursion.
> 2. line 8 If `t < l[mid]`, the sorted property of the array implies that `t` can only exist in the left sub-array. The algorithm recurses on the strictly smaller search space `[start, mid - 1]`.
> 3. By line 10 If `t > l[mid]`, the target `t` can only exist in the right sub-array. The algorithm recurses on the strictly smaller search space `[mid + 1, end]`.
>
> **By the inductive hypothesis, the recursive calls on these sub-problems, which are of a size less than k, will return a correct result. Therefore, the function is correct for a search space of size k.**

Conclude Induction