Topic 2b, Divide & Conquer: Solutions

---

1. Correctness of Binary Search

Consider the Binary Search algorithm shown below (a Divide & Conquer algorithm). The inputs are a ***sorted*** array *A* of integers. You will use a loop invariant to show that the algorithm finds *x* correctly if it is in the range of *A* to be searched.

```
Binary-Search(x, A, min, max)
1       low = min
2       high = max
3       while low <= high
4               mid =⌊(low + high) / 2⌋
5               if x == A[mid]
6                       return mid
7               else if x < A[mid]
8                       high = mid - 1
9               else low = mid + 1
10      return "NOT FOUND"
```

a. State the loop invariant for the while loop. *Hint: Think about what the algorithm does and why **you** think it is correct.*

> If x is in the range of the array A[min,max] to be searched it is in the subarray A[low,high].

b. (**Initialization**) Referencing the code, show the invariant is true at loop initialization:

> Trivially true because low=min and high=max in lines 1 and 2.

c. (Maintenance) Referencing the code, show that if the invariant is true at the beginning of a loop pass it is true at the end of the loop pass (there will be three cases):

> **Case 1**: If line 5 is true, then invariant is true because low <= mid <= high and x == A[mid]
>
> **Case 2**: If line 7 is true, then because the array is sorted all items in A[mid,high] are larger than x and may be discarded from consideration.
>
> **Case 3**: If line 9 is true, then we have x > A[mid] by elimination and because the array is sorted all items in A[low,mid] must be smaller than x, and this range may

be discarded.

d. (Termination) Use the truth of the invariant at exit to show that the algorithm is correct.

If the loop exits prematurely due to line 6, x has been found and the index at which it is found is returned correctly. (Note we haven't used the loop invariant for this part.)

If the loop exits because low > high, then the range A[low,high] has gone empty. Since the invariant was (b) true at the start and (c) has been maintained, it remains true. Thus, if x is not in this subarray, it cannot be in the range of the full array (if $P \rightarrow Q$ is true and $\neg Q$ then we can conclude $\neg P$).

e. Which part of your proof fails if the array is not sorted?

Cases 2 and 3 of the Maintenance condition fail ("because the array is sorted").

2. Tradoffs in Search and Sorting

We know from the previous class work that Linear Search is $\Theta(n)$, so one might think it is always better to use the $\Theta(\lg n)$ BinarySearch. However, in order to apply Binary Search we have to sort the data, which (we will soon learn) requires $\Theta(n \lg n)$ time for the best known algorithms (e.g., MergeSort) in the general case. This question explores when it is worth paying the extra cost of sorting the data in order to apply fast Binary Search.

Suppose you will be **searching a list of *n* items *m* times**. When is *m* big enough relative to *n* to make it worth sorting and using binary search rather than just using linear search? You will answer this question below.

We have two alternatives. Simply using Linear Search *m* times on *n* items has an expected (average) cost of $\Theta(mn)$. The second alternative is (a) below, and (b)-(d) explore the tradeoffs.

a. What is the expected cost to apply Merge Sort once to sort *n* items, and then apply Binary Search *m* times to *n* items?

Theta(n log n + m log n)

b. Suppose $m = 1$: which strategy is better, and why? Use the above expressions to justify your answer mathematically.

Linear Search time: mn = n = O(n)
Hybrid Search Time: n lg n + lg n = O(n lg n)

The Winner: LinearSearch

c. Suppose *m* = *n*: which strategy is better, and why? Use the above expressions to justify your answer mathematically.

Linear Search time: $mn = n^2$
Hybrid Search Time: n lg n + n lg n = O(n lg n)
The Winner: Hybrid

d. What is the cutoff point in terms of *m* expressed as a function of *n* between when it is faster to just apply the linear search and when it is faster to apply Merge Sort and Binary Search? Set up an equation and solve for m.

Solve mn = n lg n + m lg n for m:
mn - m lg n = n lg n
m(n - lg n) = n lg n
m = n lg n / (n - lg n)

e. Using your formula, and assuming a data size of n=1024 items, for what value of m does it become worth sorting the data?

m = n lg n / (n - lg n) so
m = 1024 * 10 / (1024 - 10)
= 10240 / 1014
= 10.0986
It is worth sorting for above 10 searches (11 or higher).