Homework 1: Searching and Sorting

Due: September 18, 2025

This homework must be typed in LaTeX and submitted via Gradescope.

Please ensure that your solutions are complete, concise, and communicated clearly. Use full sentences and plan your presentation before your write. Except where indicated, consider every problem as asking for a proof.

Problem 1. The *Fibonacci numbers* is a sequence of numbers starting with $f_1 = f_2 = 1$ defined by the recurrence:

$$f_{n+2} = f_n + f_{n+1}$$

for $n \geq 1$. Use induction to prove the following formula for $n \geq 1$:

$$P(n) := \sum_{i=1}^{n} f_i^2 = f_n f_{n+1}$$

Solution.
Base Case: In the case where $n = 1$ we have that

$$\sum_{i=1}^{1} f_i^2 = 1 = 1 \cdot 1 = f_1 f_2$$

Induction Hypothesis: Suppose that the statement holds inductively for $n = k$, or in other words

$$\sum_{i=1}^{k} f_i^2 = \underbrace{1^2 + 1^2 + \cdots + f_k^2}_{k \text{ terms}} = f_k f_{k+1}$$

Induction Step: We want to show that the statement holds for the case when $n = k + 1$. Notice that

$$\sum_{i=1}^{k+1} f_i^2 = \underbrace{1^2 + 1^2 + \cdots + f_k^2}_{P(k) = f_k f_{k+1}} + f_{k+1}^2 = f_k f_{k+1} + f_{k+1}^2 = f_{k+1}(f_k + f_{k+1})$$

By the definition of the Fibonacci numbers, we have that

$$P(k+1) = \sum_{i=1}^{k+1} f_i^2 = f_{k+1}(f_k + f_{k+1}) = f_{k+1} f_{k+2}$$

which closes the induction. □

Problem 2. A *geometric sequence* with common ratio r is a sequence of numbers given by:

$$a_1,\ a_1 r,\ a_1 r^2,\ a_2 r^3,\ \cdots$$

For example, the following is a geometric sequence with common ratio 2.

$$1,\ 2,\ 4,\ 8,\ 16,\ \cdots$$

Describe an algorithm to find the value of a deleted term of a geometric sequence of length n with common ratio r in $O(\log n)$ time. For example, the sequence

$$1,\ \frac{1}{3},\ \frac{1}{9},\ \frac{1}{81},\ \frac{1}{243}$$

is missing the term $\frac{1}{27}$.

Solution. The idea here is to modify binary search by comparing an element at a given index i to the expected value of the element of the sequence at index i. If the element in the array is not the expected value, the missing element must be in the left sub-array, otherwise the missing element is in the right sub-array.

Algorithm 1 Missing Element in Sequence

Input: A geometric sequence, arr, with common ratio r with a term missing
Output: The value of the missing term

```
1: function ModifiedBinarySearch(arr, r)
2:     leftIndex, rightIndex ← 0, length of arr - 1
3:     while leftIndex ≤ rightIndex do
4:         middleIndex ← integer average of leftIndex and rightIndex
5:         if arr[0]r^middleIndex = arr[middleIndex] then
6:             leftIndex = middleIndex + 1
7:         else
8:             rightIndex = middleIndex - 1
9:     return arr[0]r^leftIndex
```

We can compute the ith term (0-indexed) of a geometric series using the formula

$$a_i = a_0 r^i$$

Therefore, if the element at index i is not equal to what we expect, this means that the missing element must be in the left sub-array since the the recurrence relationship does not hold. If the element at index i is equal to what we expect, then the missing element must be in the right sub-array since we are guaranteed that there is one missing element.

The runtime of this algorithm is $O(\log n)$ since this is a slight modification of binary search. □

Problem 3. Let $X = [a_1, \cdots, a_n]$ and $Y = [y_1, \cdots, y_n]$ be two sorted arrays (in non-decreasing order). For simplicity, assume n is a power of 2.

(a) Describe an algorithm to find the median of all $2n$ elements in the arrays X and Y in $O(\log n)$ time.

(b) Provide a succinct proof of the correctness of the algorithm.

(c) Provide an analysis of the running time (asymptotic analysis is correct) and memory utilization of the algorithm.

Hint: Note that the given arrays are already sorted and of the **same size**! You may want to use binary search to exploit this fact. :)

Solution. **Algorithm:**

- If $n = 1$ or $n = 2$ then merge the arrays X and Y manually and find the median of the new array.
- If $n > 2$, then we'll first compute $Xw_{mid} = \lfloor \frac{n}{2} \rfloor$ and $Y_{mid} = n - 1 - X_{mid}$.
    1. (condition 1) If $(X[X_{mid}] \leq Y[Y_{mid}+1])$ and $(Y[Y_{mid}] \leq X[X_{mid}+1])$, then identify the 2 largest numbers from the list $[Y[Y_{mid}-1], Y[Y_{mid}], X[X_{mid}], X[X_{mid}-1]$ and return their average.
    2. (condition 2) If $X[X_{mid}] > Y[Y_{mid}+1]$, then recur on $X[0 : X_{mid}]$ and $Y[Y_{mid}+1 :]$, and return its result.
    3. (condition 3) Otherwise (ie. $Y[Y_{mid}] > X[X_{mid}+1]$), then recur on $X[X_{mid}+1]$ and $Y[0 : Y_{mid}]$, and return its result.

Correctness: To prove that our algorithm works for $k \geq 2$, we use strong induction over the predicate $P(n)$: our algorithm correctly computes the median of two sorted arrays of lengths n.

Base Cases: $P(1)$ and $P(2)$ hold since we are manually computing the median after merging the two arrays.

Inductive Hypothesis: Suppose that for $k = \{3, \ldots, n\}$, $P(k)$ holds. We'll prove that $P(n+1)$ holds.

If condition 1 holds, all of the elements in $X[0 : X_{mid}+1] \bigcup Y[0 : Y_{mid}+1]$ must be less than or equal to all of the elements from $X[X_{mid}+1 :] \bigcup Y[Y_{mid}+1 :]$. Then, the median will simply be the average of the two largest numbers from $X[0 : X_{mid}+1] \bigcup Y[0 : Y_{mid}+1]$.

If condition 2 holds, X_{mid} should be at a lower index in X (inversely, Y_{mid} should be at a greater index in Y). The maximum length of the sub-arrays we recur on are at least of length 1 and at most length n. Recurring on these inputs should yield the correct solution.

If condition 3 holds, X_{mid} should be at a greater index in X. As previously explained, recurring on the new sub-arrays should yield the correct solution.

Conclusion: Since our algorithm yields a correct solution in all cases of the if-then selection, $P(n+1)$ must hold. By strong induction, $P(k)$ holds for all $k \geq 1$.

Time Complexity: At each step of the algorithm, we approximately halve (ie. +/- 1) the length of the input arrays and perform a fixed series of constant time computations, C. Thus, the runtime of our algorithm is $O(C \log_2 n) = O(\log(n))$.

Memory Utilization: In practice, our approach uses a left and right pointer to keep track of the input sub-arrays. We also temporarily allocate space for X_{mid} and Y_{mid}, so our overall memory complexity is $O(1)$.

□

Problem 4. Let S be an array of n *distinct* integers. An inversion in S is a pair of indices i and j such that $i < j$, but $S_i > S_j$. For example, following sequence has six inversions:

$$\{8,\ 6,\ 4,\ 1\}$$
$$(8,6),\ (8,4),\ (8,1),\ (6,4),\ (6,1),\ (4,1)$$

Describe an algorithm running in $O(n \log n)$ time to determine the number of inversions in S.

Solution. Consider the following modification of MergeSort.

Algorithm 2 Counting Inversions

```
Input: Two sorted arrays to merge, leftArr and rightArr
Output: A pair containing the merged sorted array and the number of inversions
function ModifiedMerge(leftArr, rightArr)
    outputArr ← []
    inversionCounter ← 0

    while leftArr and rightArr are not empty do
        if rightArr is empty or leftArr[0] ≤ rightArr[0] then
            remove the first element of leftArr and append it to outputArr
        else if leftArr is empty or rightArr[0] < leftArr[0] then
            increment inversionCounter by the length of leftArr
            remove the first element of rightArr and append it to outputArr

    return (outputArr, inversionCounter )

Input: An array to sort and optionally the total number of inversions so far
Output: The sorted array and the number of inversions required to sort the array
function ModifiedMergeSort(arr, numberOfInversions = 0)
    if the arr has length ≤ 1 then
        return (arr, 0)

    leftArr, leftInversions ← ModifiedMergeSort(left half of arr)
    rightArr, rightInversions ← ModifiedMergeSort(right half of arr)
    sortedArr, mergeInversions ← ModifiedMerge(leftArr, rightArr)

    return (sortedArr, leftInversions + rightInversions + mergeInversions )
```

This algorithm correctly counts the number of inversions in an input array. The order of the input numbers in MergeSort are changed during the merge step. Since the arrays to ModifiedMerge are sorted, if the leftmost element of the right array is smaller than the leftmost element of the left array, we know that the leftmost element of the right array is smaller than all of the elements in the left array. These are exactly the inversions in the original array. Therefore, the total number of inversions in the array are given by the total number of inversions required to sort each sub-array along with the total number of inversions required to merge the sub-arrays.

The modifications required to MergeSort to count inversions does not affect the overall big-O runtime class. Incrementing a counter in ModifiedMerge will affect the runtime of ModifiedMerge by a constant. Therefore, the overall runtime of ModifiedMergeSort is in the same asymptotic class as MergeSort which runs in $O(n \log n)$ time. □