Homework 2: Searching and Sorting

Due: February 4, 2026 at 11:59 p.m.

This homework must be typed in LaTeX. Please use the code released as part of the homework, remove these instructions and fill in the solutions.

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

Your solutions must be submitted using Gradescope. When doing so, please correctly mark the pages with the solutions to each problem. a

Problem 1. A *geometric sequence* with common ratio r is a sequence of numbers given by:

$$a_1,\ a_1r,\ a_1r^2,\ a_2r^3,\ \cdots$$

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

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

(a) 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 $\frac{1}{27}$.

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

(c) Prove an upper bound on the running time of your algorithm (asymptotic analysis is sufficient).

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.
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 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. □

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
function ModifiedBinarySearch(arr, $r$)
    leftIndex, rightIndex $\leftarrow$ 0, length of arr - 1
    while leftIndex $\leq$ rightIndex do
        middleIndex $\leftarrow$ integer average of leftIndex and rightIndex
        if arr[0]$r^{\texttt{middleIndex}}$ = arr[middleIndex] then
            leftIndex = middleIndex + 1
        else
            rightIndex = middleIndex - 1
    return arr[0]$r^{\texttt{leftIndex}}$
```

Problem 2. 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:**

```
median(xs, ys):
    if |xs| = 1:
        let [x] = xs
            [y] = ys
        return (x + y) / 2

    let mid      = floor(|xs| / 2)
        small_xs = xs[:mid]
        big_xs   = xs[mid:]
        small_ys = ys[:mid]
        big_ys   = ys[mid:]

    if small_ys[-1] <= small_xs[-1] and big_xs[0] <= big_ys[0]:
        return (small_xs[-1] + big_xs[0]) / 2

    if small_xs[-1] <= small_ys[-1] and big_ys[0] <= big_xs[0]:
        return (small_ys[-1] + big_ys[0]) / 2
    if big_xs[0] <= big_ys[0]:
        return median(big_xs, small_ys)

    return median(small_xs, big_ys)
```

Correctness: First, a lemma: let x and y respectively be the nth- and $(n+1)$th-smallest elements in the size-n inputs (total $2n$ elements). Removing c elements less than or equal to x and c greater than or equal to y, as long as x and y are not removed, preserves the median. □ (Proof is trivial)

We prove correctness by weak induction on (the base-2 logarithm of) the power-of-two size n of the inputs. In the base case, this algorithm is clearly correct for $n = 2^0$.

Suppose the algorithm is correct on all inputs of size 2^k for some $k \geq 0$. We want to prove that it is correct on all inputs of size $n = 2^{k+1}$.

If the line 13 condition holds, then each small element is less than or equal to each big element, small_xs[-1] is greatest among small elements, and big_xs[0] is least among big elements. That means small_xs[-1] and big_xs[0] are the middle two elements and their average is the median of the inputs. Similarly, if the line 16 condition holds, the median is the average of small_ys[-1] and big_ys[0].

Otherwise, if big_xs[0] $\leq$ big_ys[0], we know small_xs[-1] $<$ small_ys[-1]. There are at least $n+1$ elements (small_ys[-1] and all big elements) greater than or equal to small_xs[-1] and similarly at least $n+1$ less than or equal to big_ys[0]. The lemma above tells us that removing all of small_xs and big_ys from the inputs preserves the median, and our inductive hypothesis says that the recursive call computes this median correctly.

Otherwise, big_xs[0] $>$ big_ys[0], and by the same argument the recursive call gives us the correct median. □

Time Complexity: At each step of the algorithm, we exactly halve 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: Our approach uses a constant number of constant-size local variables (the array slices can be represented as pairs of bounds instead of copies of the elements). It makes up to one recursive call, but that is a tail call. Our overall memory complexity is $O(1)$.

□

Problem 3. Let A be an array of n *distinct* integers. An *inversion* in A is a pair of indices i and j such that $i < j$, but $A_i > A_j$. For example, the following sequence has three *inversions*:

$$\{1,\ 5,\ 2,\ 8,\ 4\}$$
$$(5,2),\ (5,4),\ (8,4)$$

(a) Provide a succinct (but clear) description of an algorithm running in $O(n \log n)$ time to determine the number of *inversions* in A. You may provide a pseudocode.

(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.

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 length of left Arr
            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 as it is constant time comparison. 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. □

Problem 4. Through this problem, assume you are working only with comparison-based sorting algorithms. All following questions refer to the following standard insertion sort:

```
for j = 2 to n:
    key = A[j]
    i = j - 1
    while i >= 1 and A[i] > key:
        A[i+1] = A[i]
        i = i - 1
    A[i+1] = key
```

You may assume constant-time overhead per loop iteration.

Part I: Inversions and Insertion Sort

As previously defined, an *inversion* in array A is a pair (i, j) with $i < j$ and $A[i] > A[j]$. Let $\text{inv}(A)$ denote the number of inversions.

1. Show that each execution of the inner `while` loop removes exactly one inversion.
2. Deduce that insertion sort runs in
$$\Theta(n + \text{inv}(A)).$$

Part II: k Elements in Correct Final Positions (Known)

Exactly $k < n$ elements are in their final sorted positions, and the algorithm **is told which elements these are**.

1. Analyze the running time of **standard insertion sort** under this assumption. (Do not modify the algorithm.)
2. Prove a comparison-based lower bound under this assumption.
3. Design a comparison-based sorting algorithm that exploits this knowledge to run in
$$O((n - k) \log(n - k)).$$

Part III: k Elements in Correct Final Positions (Unknown)

Exactly $k < n$ elements are in their final positions, but the algorithm **does not know which elements**. Access is only via comparisons.

1. Give an asymptotic upper bound on $\text{inv}(A)$ in terms of n and k.
2. Using Part I, analyze the running time of insertion sort.
3. Compare this running time with that obtained under the assumption of Part II. What do you observe?
4. Prove a comparison-based lower bound under this assumption.
5. Would the standard Merge Sort algorithm's running time be affected by this assumption? An informal explanation (without proof) will be sufficient.

Part IV: k Elements in Correct Relative Order

The array contains a strictly increasing subsequence of length k, but these elements are not necessarily in their final positions.

1. Give an asymptotic upper bound on $\text{inv}(A)$ in terms of n and k.
2. Using Part I, analyze the running time of insertion sort.
3. Prove a comparison-based lower bound under this assumption.
4. Are there scenarios where insertion sort may perform better than Merge Sort under this assumption? An informal explanation (without proof) will be sufficient.

Solution. We analyze the following standard version of insertion sort.

```
for $i = 2$ to $n$ do
    $x \leftarrow A[i]$
    $j \leftarrow i - 1$
    while $j \geq 1$ and $A[j] > x$ do
        $A[j+1] \leftarrow A[j]$
        $j \leftarrow j - 1$
    $A[j+1] \leftarrow x$
```

Only comparisons inside the `while` loop contribute asymptotically to the running time.

Part 1: Running Time in Terms of Inversions

Definition. An *inversion* in an array $A[1 \dots n]$ is a pair (i, j) such that $i < j$ and $A[i] > A[j]$. Let $\text{inv}(A)$ denote the total number of inversions.

Claim 1.1

Insertion sort executes the `while` loop exactly $\text{inv}(A)$ times.

Proof. We prove by induction on i, the index of the outer loop:
Base case ($i = 2$): If $A[1] > A[2]$, one inversion exists and one `while` iteration occurs; otherwise none.
Inductive hypothesis: Assume that after iteration $i-1$, the total number of `while` executions equals the number of inversions among $A[1 \dots i-1]$.
Inductive step: At iteration i, element $x = A[i]$ is compared with preceding elements. Each `while` execution corresponds to an inversion (j, i) with $A[j] > x$. The loop terminates after eliminating all inversions involving i. By induction, total executions equal $\text{inv}(A)$. □

Corollary 1.2

The running time of insertion sort is

$$\Theta(n + \text{inv}(A)).$$

Proof. Constant-time work outside the `while` loop contributes $O(n)$; the loop executes $\text{inv}(A)$ times. □

Part 2: k elements in correct final positions, algorithm knows which ones

2.1 Insertion sort running time: Let k be the number of elements in correct final positions (known). Each of the remaining $n-k$ elements may form inversions with each other and possibly with the k fixed elements.

$$\text{inv}(A) \leq \Theta(n^2 - k^2)$$

Thus, the Insertion sort running time is

$$T(n,k) = \Theta(n + \text{inv}(A)) = \Theta(n^2 - k^2)$$

2.2 Comparison-based lower bound: Since the algorithm knows which elements are fixed, it can ignore them. Only $(n-k)$ elements need sorting. Number of valid input permutations: $(n-k)!$. Decision-tree depth $\geq \log_2((n-k)!) = \Theta((n-k)\log(n-k))$

2.3 Achievability: A modified mergesort skipping fixed elements achieves

$$T(n,k) = \Theta((n-k)\log(n-k) + n)$$

which is asymptotically optimal.

Part 3: k Elements in Correct Final Positions (Identities Unknown)

3.1 Upper bound on inversions: Each of the k elements can form inversions with the remaining $n-k$ elements. Remaining $n-k$ elements may be fully reversed.

$$\text{inv}(A) = \Theta(n^2 - k^2)$$

3.2 Insertion sort running time:

$$T(n,k) = \Theta(n^2 - k^2)$$

3.3 Observation It is the same. Insertion sort is agnostic to the information regarding how many elements are correctly placed and which ones those are.

3.4 Comparison-based lower bound: Choose which k elements are fixed: $\binom{n}{k}$ Permute remaining $n-k$ elements freely: $(n-k)!$

$$\#\text{ possible permutations} = \binom{n}{k}(n-k)!$$

Decision-tree depth $\geq \log_2(\binom{n}{k}(n-k)!) = \Theta(n\log n - k\log k)$

3.5 Merge Sort Comparison. Merge sort always runs in $\Theta(n\log n)$. Insertion sort can outperform merge sort when $(n^2 - k^2) = o(n\log n)$.

Part 4: k Elements in Correct Relative Order (not necessarily final positions)

Setup: Let S denote the set of k elements that are already in correct relative order.

- The remaining $n-k$ elements may be arbitrarily permuted.
- Two subcases:
  - The algorithm *knows* which k elements are relatively sorted.
  - The algorithm *does not know* which k elements are relatively sorted.

4.1 Upper bound on number of inversions

- Inside S, elements are relatively ordered: no inversions among themselves.
- Inside $A \setminus S$, elements may form $\binom{n-k}{2}$ inversions in worst case.
- Cross inversions between S and $A \setminus S$: each element of S can be inverted with up to $n-k$ elements.

Thus, total inversions:

$$\text{inv}(A) \leq \binom{n-k}{2} + k(n-k) = \Theta(n^2 - k^2)$$

4.2 Insertion sort running time

Is $\Theta(n + \text{inv}(A))$, becomes

$$T(n,k) = \Theta(n^2 - k^2)$$

regardless of whether the algorithm knows which elements are in S, because insertion sort does not inherently use this knowledge.

4.3 Comparison-based lower bound

Case a: Algorithm does *not* know which k elements are sorted.

- Number of ways to choose the k relatively ordered elements: $\binom{n}{k}$
- Their internal order is fixed (1 possibility)
- Remaining $n-k$ elements can be permuted freely: $(n-k)!$
- Number of ways to interleave S with $A \setminus S$: $\binom{n}{k}$

$$\text{Number of valid input permutations} = \binom{n}{k}^2 (n-k)!$$

Decision-tree depth $\geq \log_2(\binom{n}{k}^2(n-k)!)$, so the asymptotic lower bound is:

$$\boxed{\Omega(n \log n - k \log k)}$$

Case b: Algorithm *knows* which k elements are sorted.

- Since the algorithm knows the sorted set S, it can safely ignore these elements.
- Only the remaining $n - k$ elements need to be sorted.
- Number of valid input permutations: $(n - k)!$
- Decision-tree depth $\geq \log_2((n - k)!)$
- Lower bound: $\Omega((n - k)\log(n - k))$

4.4 Observation on insertion sort vs vanilla mergesort

- Insertion sort performs $\Theta(n + \text{inv}(A)) = \Theta(n^2 - k^2)$ in the worst case, depending on the number of inversions.
- Vanilla mergesort does not exploit any pre-existing order and always performs $\Theta(n \log n)$ comparisons.
- Comparison:
  - If the number of inversions is very small ($\text{inv}(A) \ll n \log n$), insertion sort can be faster in practice.
  - Otherwise, for large n or when inversions are not extremely few, vanilla mergesort dominates asymptotically.
- Key point: Whether the algorithm knows which k elements are sorted or not does *not* affect vanilla mergesort; it always performs $\Theta(n \log n)$.

□