CSCI 270 Problem Set 1 Solutions

Fall 2025

Problem 1. *You are given an unsorted array $A = (a_1, \ldots, a_n)$ of n numbers. We say that a pair (a_i, a_j) is an inverse pair if $i < j$ and $a_i > a_j$. For example, if $A = (3, 4, 1, 2, 5)$, then $(3, 1)$ is an inverse pair. In total, there are 4 inverse pairs in this example: $(3, 1), (4, 1), (3, 2), (4, 2)$. Your goal is to count how many inverse pairs are in A. Please provide an $O(n \log n)$-time algorithm with pseudocode, a running-time analysis, and a correctness proof. [Hint: think about the MergeSort.]*

Solution. We use the following modification of MergeSort.

```python
def count_inversions(A, return_list=False):
    if len(A) <= 1:
        return 0

    # Divide & conquer
    r = len(A) // 2
    left, left_count = count_inversions(A[:r], return_list=True)
    right, right_count = count_inversions(A[r:], return_list=True)

    # Merge
    merged = []
    i, j, merge_count = 0, 0, 0
    while i < len(left) and j < len(right):
        if left[i] <= right[j]:
            merged.append(left[i])
            i += 1
        else:
            merged.append(right[j])
            merge_count += len(left) - i # Key alteration!
            j += 1

    # Append any remaining elements
    merged.extend(left[i:])
    merged.extend(right[j:])

    total_inversions = left_count + right_count + merge_count
    if return_list:
```

```
28          return merged_list, total_inversions
29      return total_inversions
```

We first demonstrate that the algorithm is correct. Let A be an arbitrary array and let B, C denotes its left and right halves, i.e., $B = A[:r]$ and $C = A[r:]$ for $r = \text{len}(A)/2$. Furthermore, for an element $y \in C$, let $f(y)$ denote the number of elements $b \in B$ with $b > y$. Lastly, let $\phi(\cdot)$ denote the number of inversions in an array.

We claim that

$$\phi(A) = \phi(B) + \phi(C) + \sum_{y \in C} f(y). \tag{1}$$

To see why, note that

$$\begin{aligned}
\phi(A) &= \left|\left\{i, j : i < j, a_i > a_j\right\}\right| \\
&= \left|\left\{i, j : i < j \le r, a_i > a_j\right\} \bigsqcup \left\{i, j : r < i < j, a_i > a_j\right\} \bigsqcup \left\{i, j : i \le r < j, a_i > a_j\right\}\right| \\
&= \left|\left\{i, j : i < j \le r, a_i > a_j\right\}\right| + \left|\left\{i, j : r < i < j, a_i > a_j\right\}\right| + \left|\left\{i, j : i \le r < j, a_i > a_j\right\}\right| \\
&= \phi(B) + \phi(C) + \sum_{y \in C} f(y).
\end{aligned}$$

The first line simply uses the definition of inversions, i.e., of $\phi(A)$, and the second and third lines condition on whether i and j are both less than r, both greater than r, or contain r between them. The final equality uses that

$$\left|\left\{i, j : i \le r < j, a_i > a_j\right\}\right| = \sum_{j > r} \left|\left\{i : i \le r, a_i > a_j\right\}\right| = \sum_{y \in C} \left|\left\{b \in B : b > y\right\}\right| = \sum_{y \in C} f(y).$$

Thus, in order to demonstrate correctness of our algorithm, it remains to show only that `merge_count` correctly tracks $\sum_{y \in C} f(y)$. To this end, fix a particular $y \in C = A[r:] =$ `right`. We will demonstrate that y contributes exactly $f(y)$ to the value of `merge_count`.

Consider the moment when y is added to the array `merged`. If it occurs on line 18, then `merge_count` is incremented by the number of entries in `left` which have *not* been added to `merged`. This is equal to precisely the number of entries in `left` which are strictly larger than y, by correctness of the standard Merge method in MergeSort. On the other hand, if it occurs on line 24, then the `left` array was exhausted during the original merge processing (lines 11-20), meaning that y is greater than all elements in `left` and $f(y) = 0$. In each case, `merge_count` correctly tracks $f(y)$ for each $y \in$ `right`, demonstrating the algorithm's correctness.

Finally, the algorithm runs in time $O(n \log n)$ because it does not increase in runtime relative to MergeSort; line 19 requires constant work, which is already being performed in the same clause by line 20. More explicitly, the algorithm's runtime follows the recurrence

$$T(n) = 2T(n/2) + O(n),$$

as it calls itself twice on inputs of size $n/2$ and performs $O(n)$ computation in merging the arrays. This yields a runtime of $T(n) = O(n \log n)$, as demonstrated in class. □

Problem 2. *Given integers a and n, and a prime p, design an algorithm to compute $a^n \bmod p$? Provide pseudocode, a running-time analysis, and a correctness proof. You may assume that p is a constant; therefore, the multiplication of two integers smaller than p takes $O(1)$ time.*

Solution. We again employ a divide and conquer strategy. The pseudocode is as follows.

```
1  def exponentiate(a, n, p):
2      if n == 0:
3          return 1
4      base = a % p
5
6      def inner(a, n, p):
7          if n == 1:
8              return base
9          if n % 2 == 0:
10             r = n / 2
11             half = inner(a, r, p)
12             return half * half % p
13         elif n % 2 == 1:
14             r = n // 2
15             half = inner(a, r, p)
16             return base * half * half % p
17
18     return inner(a, n, p)
```

Correctness follows trivially by inducting on n, for arbitrary fixed a and p. When $n \in \{0, 1\}$, the algorithm is correct due to lines 3 and 8, i.e., $a^0 = 1 \bmod p$ and $a^1 = a \bmod p$. Now let $f(a, n, p)$ denote the function computed by `exponentiate`, and fix an even $n \geq 2$. We have

$$\begin{aligned} f(a, n, p) &= f(a, n/2, p) \cdot f(a, n/2, p) \bmod p \\ &= \left(a^{n/2} \bmod p\right) \cdot \left(a^{n/2} \bmod p\right) \bmod p \\ &= a^n \bmod p. \end{aligned}$$

The first equality uses the recursive definition of `inner` (i.e., lines 10-12), the second equality uses the inductive hypothesis concerning the correctness of `exponentiate` (and thus of `inner`), and the final line is a simple algebraic manipulation. When $n = 2 \cdot r + 1$ is odd, we

similarly have

$$\begin{aligned} f(a,n,p) &= \Big(a \bmod p\Big) \cdot f(a,r,p) \cdot f(a,r,p) \bmod p \\ &= \Big(a \bmod p\Big) \cdot \Big(a^r \bmod p\Big) \cdot \Big(a^r \bmod p\Big) \bmod p \\ &= a^n \bmod p, \end{aligned}$$

by using lines 14-16 of the algorithm, the assumed correctness of our algorithm on smaller inputs (i.e., the inductive hypothesis), and algebraic manipulation.

We now analyze the algorithm's runtime. First observe that the computation of $a \bmod p$ in line 4 requires time $O(\log(a))$, i.e., linear in the number of bits used to represent a. (Due to the fact that p is being treated as a constant.) Upon computing this base value, the runtime of the algorithm can be described by the recurrence relation

$$T(n) = T(n/2) + O(1),$$

as the algorithm makes one recursive calls on an instance of size $\leq n/2$ (lines 11, 15), and performs a final multiplication (lines 12, 16) which takes constant time owing to the fact that p is treated as a constant. This leads to a bound of $T(n) = O(\log(n))$, as demonstrated in class. The overall runtime is then $\boxed{O\big(\log(n) + \log(a)\big)}$. □

Problem 3. *We defined the lexicographic order in the lecture. For any strings A and B, we use $A <_L B$ to denote that A is smaller than B in lexicographic order. Please prove that, for any strings A, B, and C, if $A <_L B$ and $B <_L C$, then $A <_L C$.*

Solution. First suppose that A is a prefix of B. Then C cannot be a prefix of A, otherwise C is a prefix of B, producing contradiction with $B <_L C$. Thus either A is a prefix of C, immediately yielding $A <_L C$, or there exists a minimum index $i_{A,C}$ for which $A[i] \neq C[i]$. As A is a prefix of B, however, then $i_{A,B}$ is also the minimum index for which B and C disagree, meaning

$$A[i_{A,C}] = B[i_{A,C}] < C[i_{A,C}]$$

due to the fact that $B <_L C$, and thus $A <_L C$.

Now suppose that B is a prefix of C. Then we may assume there exists a minimum index $i_{A,B}$ with $A[i_{A,B}] \neq B[i_{A,B}]$, otherwise A is a prefix of B (which has been treated above) or B is a prefix of A (producing contradiction with $A <_L B$). Then, as B is a prefix of C, $i_{A,B}$ is also the minimum index for which A and C disagree, meaning

$$A[i_{A,B}] < B[i_{A,B}] = C[i_{A,B}],$$

which again yields $A <_L C$.

Let $<_P$ denote the prefix relation for strings. We may now assume that none of A, B, or C are prefixes of another. In particular, the cases $A <_P B$, $B <_P C$, and $A <_P C$ have been argued above, while $B <_P A$ and $C <_P B$ each produce contradiction with the conditions $A <_L B$ and $B <_L C$. Lastly, $C <_P A$ and $A <_L B$ would imply that $C <_L B$, as argued above, again producing contradiction.

As there are no prefix relations among A, B, and C, we may define $i_{A,B}$ to be the minimum index i for which $A[i] \neq B[i]$, and likewise for $i_{B,C}$ and $i_{A,B}$. We perform casework.

- $i_{A,C} < i_{A,B}$: Then $i_{B,C} = i_{A,C}$, as $A[i] = B[i]$ for all $i < i_{A,B}$. Then
$$A[i_{A,C}] = B[i_{A,C}] < C[i_{A,C}]$$
due to the fact that $B <_L C$, yielding $A <_L C$.

- $i_{A,C} = i_{A,B}$: Let $j := i_{A,B} = i_{A,C}$. Then $i_{B,C} \geq j$, as $B[i] = A[i] = C[i]$ for all $i < j$. First suppose $i_{B,C} = j$. Then we have
$$A[j] < B[j] < C[j],$$
meaning $A[j] < C[j]$ by transitivity. Otherwise, $i_{B,C} > j$ and we have
$$A[j] < B[j] = C[j],$$
again yielding that $A[j] < C[j]$ and thus $A <_L C$, as desired.

- $i_{A,C} > i_{A,B}$: Then $C[i_{A,B}] = A[i_{A,B}] < B[i_{A,B}]$. As $B <_L C$, however, it must be that
$$i_{B,C} < i_{A,B} < i_{A,C}.$$
This yields contradiction, however, as it cannot be that $A[i_{B,C}] = B[i_{B,C}] \neq C[i_{B,C}]$ yet $A[i_{B,C}] = C[i_{B,C}]$.

This completes the proof. □

Problem 4. *We have studied the suffix array in the lecture. For simplicity, we now assume that $\Sigma = \{0, 1\}$, i.e., we only consider binary strings. As we also mentioned in the lecture, for any string S of length n, it has $(n + 1)$ suffixes.*
Now we assume the following nice uniform property of S:

- *For any string P of length k, the number of suffixes that start with P is upper bounded by $n/1.2^k$, i.e., P can not be the prefix of more than $n/1.2^k$ suffixes of S.*

Now assume that $n = 2^{64}$ and you have 64 GB to store the lookup table. How many memory accesses do you need to determine whether A is a substring of S? Here we assume that you already have the sorted array and the lookup table, and you only need one memory query to compare whether A is a prefix of any suffix.

Solution. We first compute the size of the largest lookup tables which we can store. Recall that we require two lookup tables, L_1 and L_2, and that each table occupies
$$(k + \log_2(n)) \cdot 2^k$$
bits, where k equals the length of the prefixes stored in the table. In particular, each key is a bit string of length k, each value is an index into S of length $\log_2(n)$, and there are 2^k such (key, value) pairs to be stored. Thus the largest lookup tables which can be stored using 64GB are described by the largest k for which:
$$\begin{aligned} 2 \cdot (k + \log_2(n)) \cdot 2^k &\leq 64\,\text{GB} \\ 2 \cdot (k + 64) \cdot 2^k &\leq 64 \cdot 1024^3 \cdot 8 \\ (k + 64) \cdot 2^k &\leq 2^{38} \\ k &\leq 31. \end{aligned}$$

Now assume we are equipped with such lookup tables for $k = 31$. Then, using the special property of S, the number of memory accesses required to determine whether A is a substring of S is at most

$$\begin{aligned}\left\lceil \log_2\left(\frac{n}{1.2^k}\right)\right\rceil &= \left\lceil \log_2\left(\frac{2^{64}}{1.2^{31}}\right)\right\rceil \\ &= \lceil 64 - 31 \cdot \log_2(1.2)\rceil \\ &= \lceil 64 - 8.15\rceil \\ &= 56.\end{aligned}$$

□