

CS375 Weekly Assignment 4

Due Mar 10th 10PM ET

A general note for CS375: When writing up your homework, please write neatly and explain your answers clearly, giving all details needed to make your answers easy to understand. Graders may not award full credit to incomplete or illegible solutions.

You are encouraged to discuss the problems with your classmates, **but your submission should be your own**. When writing your solutions up, you should not be looking at any other student's submission.

Clear communication **is the point**, on every assignment. In general in CS375, unless explicitly specified otherwise, answers should be accompanied by explanations. Answers without explanations may not receive full credit. Please feel free to ask me any questions about explanations that might come up!

Necessary acknowledgements: many of these questions are reproductions or adaptations of questions of Eric Aaron's devising. Others are from our course texts as listed on the syllabus.

1. A pair of numbers in a list are *inverted* if one is bigger than the other and appears before it. For example, the list [3, 8, 2, 5, 1, 4, 7, 6] has the following 12 inversions: (3, 2), (3, 1), (8, 2), (8, 5), (8, 1), (8, 4), (8, 7), (8, 6), (2, 1), (5, 4), (5, 1), (7, 6). Design and analyze a divide and conquer algorithm for counting the number of inversions and show that its runtime is $\Theta(n \log n)$ (note that this is both a lower and upper bound). (**Hint:** Start with the pseudocode for MERGESORT and adjust it for this problem)

Solution: The idea for this algorithm is to split the input into a left half and right half and count:

- The inversions solely on the left half
- The inversions solely on the right half
- The inversions where the big item is on the left and the small item on the right

To make the algorithm easier, we'll sort the array while we're going.

```
def SortAndCount(A[0:n]):
    if n <= 1:
        return (A, 0) # Can't have an inversion in a list of at most 1 item

    mid = n / 2

    (L, L_count) = SortAndCount(A[:mid])
    (R, R_count) = SortAndCount(A[mid:])
    (M, Cross_count) = MergeAndCount(L, R)

def MergeAndCount(L, R):
    n = len(L) + len(R)
    inversion_count = 0
    merged = []

    L_idx = 0
    R_idx = 0
    for i in range(n):
        if L_idx < len(L) and L[L_idx] <= R[R_idx]:
            add L[L_idx] to merged
            L_idx++
        else:
            inversion_count += len(L) - L_idx
            add R[R_idx] to merged
```

```

        R_idx++

    return (merged, inversion_count)

```

Let's start by proving the correctness of **MergeAndCount**: specifically, that it counts the number of inversions correctly and that it correctly merges **L** and **R** into a sorted list. For the sorting part, I'm just going to appeal to the fact that the creation of **merged** follows the code we had for MERGESORT exactly. As for counting inversions, let's show that at the start of iteration **i** it will hold that **inversion_count** is exactly the number of inversions crossing the middle of **A** to the elements of **R** that are already in **merged**. As a base case, when **i** = 0 this statement is vacuously true, since **merged** is still empty. Now, on iteration **i**, we either add the next item in **L** or the next item in **R** to **merged**.

- If the item is from **L**, then **merged** doesn't contain anything new from **R**, so the statement still holds.
- If the item is from **R**, then we increase **inversion_count** by the number of items remaining in **L**: note that none of the earlier items in **L** would form inversions with this item from **R** because all items processed earlier are smaller in value. However, every remaining item in **L** is at least as big as the next item in **L** and therefore is bigger than this item from **R**, which shows that we increase **inversion_count** correctly for this iteration.

As for the runtime, we haven't changed the runtime from MERGESORT at all, so we still have a $\Theta(n \log n)$ runtime.

2. Recall the PARTITION algorithm from class:

```

def Partition(A[1, ..., n], r) -> int:
    """ Partitions the array A in such a way that all items smaller than A[r]
    appear before all other items. Note that this may result in A[r] being moved
    to a new location: the new location is returned. """

    Swap A[r] and A[n]

    bigs = 1 # This will mark the start of the indices which are at least as big
              # as the pivot (A[n])

    for i in range(1, n):
        if A[i] < A[n]:
            Swap A[i] and A[bigs]
            bigs += 1

    swap A[bigs] and A[n]

    return bigs

```

Using induction, show that this algorithm is correct. Namely, you're showing that It's a good exercise to come up with it yourself, so I'll sort of hide it, but the exact statement you should be trying to show is in the comments of the tex file if you'd like a hint.

Solution: We're going to show that at the beginning of iteration **i**, **A[1:bigs]** consists only of items smaller than **A[n]**, while **A[bigs:i]** consists only of items that are at least as big as **A[n]**.

On iteration **i** = 1, we have **bigs** = 1 so **A[1:bigs]** = **A[bigs:i]** = [], which means our claim holds vacuously.

Now, suppose we know our claim holds for a particular iteration **i**: we'll show it holds for the next as well. When processing $x = A[i]$:

- if $x < A[n]$, then we swap **A[i]** and **A[bigs]** and increase **bigs** by 1. By doing so, all the things before **bigs** now also includes x , which by induction are all smaller than **A[n]**. Additionally, what was originally at index **bigs** at the start of the loop is now at index **i**, so the contents of **A[bigs:i + 1]** are all at least as big as **A[n]**.
- if $x \geq A[n]$, then we do nothing, but this is fine since **A[bigs:i + 1]** are all at least as big as **A[n]**.

Either way our claim holds, completing the induction.

In particular, when the for-loop ends, we have that $A[1:\text{bigs}]$ consists only of items smaller than $A[\text{n}]$, while $A[\text{bigs}:\text{n}]$ consists only of items that are at least as big as $A[\text{n}]$. As a result, when we swap $A[\text{n}]$ and $A[\text{bigs}]$, we have segregated the array into a left and right portion where $A[:\text{bigs}]$ are all smaller than our midpoint $A[\text{bigs}]$, and all the values to the right $A[\text{bigs} + 1 :]$ are all at least as big as $A[\text{bigs}]$.

Remaining solutions (for the recursive formulae) coming on next assignment