CS 170 Homework 2

Due **Monday 9/11/2023, at 10:00 pm (grace period until 11:59pm)**

1 Study Group

List the names and SIDs of the members in your study group. If you have no collaborators, you must explicitly write "none".

2 Two sorted arrays

You are given two sorted arrays of integers, each of size k. Give an efficient (i.e., better than $O(k)$-time) algorithm to find the k-th smallest element in the union of the two arrays. You may assume that all the elements are distinct.

Your solution should contain a description of the algorithm, a proof of correctness and runtime analysis. (i.e a 3-part solution). **In addition, please provide corresponding pseudo-code.**

Solution:

We compare the middle elements of both lists (if k is odd; when k is even, we need to be slightly more careful - see pseudocode) If the middle element first list is smaller than the median of the second list, then the median can't be in the left half of the first list or the right half of the second list. If the median of the second list is smaller, the opposite is true. So we can cut these elements off, and recurse on the two new arrays.

When each list is size 1, $k = 1$ and we can just compare the elements and return the smaller one.

To understand how to deal with the case where k is even, we provide the following pseudo-code and write the proof of correctness in terms of the pseudocode. Students who did not consider the case where k is even and there is a rounding issue can still receive full credit if their main idea is otherwise correct.

Pseudocode

```
procedure TwoArraySelection($a[1..k]$, $b[1..k]$, element rank $k$)
    Let $\ell_1 = \lfloor k/2 \rfloor$ and $\ell_2 = \lceil k/2 \rceil$
    while $a[\ell_1] \neq b[\ell_2]$ and $k > 1$ do
        if $a[\ell_1] > b[\ell_2]$ then
            Set $a := a[1, \cdots, \ell_1]$; $b := b[\ell_2 + 1, \cdots, k]$; $k := \ell_1$
            Let $\ell_1 = \lfloor k/2 \rfloor$ and $\ell_2 = \lceil k/2 \rceil$
        else
            Set $a := a[\ell_1 + 1, \cdots, k]$; $b := b[1, \cdots, \ell_2]$; $k := \ell_2$
            Let $\ell_1 = \lfloor k/2 \rfloor$ and $\ell_2 = \lceil k/2 \rceil$
    if $k = 1$ then
        return min($a[1]$, $b[1]$)
    else
        return $a[\ell_1]$
```

Proof of correctness

Let s_k be the element we are looking for. Our algorithm starts off by comparing elements $a[\ell_1]$ and $b[\ell_2]$. Suppose $a[\ell_1] > b[\ell_2]$.
Then, in the union of a and b there can be at most $k-2$ elements smaller than $b[\ell_2]$, i.e. $a[1, \cdots, \ell_1 - 1]$ and $b[1, \cdots, \ell_2 - 1]$, and we must necessarily have $s_k > b[\ell_2]$. Similarly, all elements $a[1, \cdots, \ell_1]$ and $b[1, \cdots, \ell_2]$ will be smaller than $a[\ell_1 + 1]$; but these are k elements, so we must have $s_k < a[\ell_1 + 1]$.
This shows that s_k must be contained in the union of the subarrays $a[1, \cdots, \ell_1]$ and $b[\ell_2 + 1, \cdots, k]$. In particular, because we discarded ℓ_2 elements smaller than s_k, s_k will be the ℓ_1th smallest element in this union.
We can then find s_k by recursing on this smaller problem. The case for $a[\ell_1] < b[\ell_2]$ is symmetric.
If we reach $k = 1$ before $a[\ell_1] = b[\ell_2]$, we can cut the recursion a little short and just return the minimum element between a and b. You can make the algorithm work without this check, but it might get clunkier to think about the base cases.

Alternatively, if we reach a point where $a[\ell_1] = b[\ell_2]$, then there are exactly k greater elements, so we have $s_k = a[\ell_1] = b[\ell_2]$.

Running time analysis

At every step we halve the number of elements we consider, so the algorithm will terminate in $\log(k)$ recursive calls. Assuming the comparison takes constant time, the algorithm runs in time $\Theta(\log k)$.

(Why should we expect the runtime to be $\Theta(\log k)$? The lists are sorted, so a single comparison lets us rule out a constant fraction of the elements, similar to algorithms like binary search. As a lower bound, a comparison can only provide 1 "bit" of information, and there are $2k$ possibilities, so we need $\Omega(\log k)$ bits to distinguish between these possibiilities).

Alternative solution

Notice that for a given element $A[i]$, we can determine in $O(1)$ whether the answer is before or after $A[i]$ by comparing it to $B[k-i]$ and $B[k-i+1]$. Assuming that the element we are looking for is in A, binary search as follows. Let $A[m]$ be the middle element of the array, if $B[k-m] < A[m] < B[k-m+1]$, then the answer is $A[m]$. In the case $A[m] > B[k-m+1]$, then we recurse on the left half of A. Otherwise if $A[m] < B[k-m]$, we recurse on the right half.

If the search is unsuccessful on A, we repeat the same strategy on B.

Correctness: In the case when $B[k-m] < A[m] < B[k-m+1]$, the rank of $A[m]$ in the union is $m + k - m = k$ as needed. If $B[k-m] > A[m]$, then we know that there are at less than k elements before $A[m]$, so the answer should be in the right half. Similarly if $B[k-m+1] < A[m]$, then we know there are more than k elements before A and the answer should be to its left.

Runtime analysis: Each step in the binary search requires $O(1)$ comparisons and then we recurse on an array half the size. Hence, the runtime is $O(\log n)$.

3 Counting multiples of 3

You are given an array A of n distinct non-negative integers. Give an $O(n(\log n)^2)$ time algorithm to count the number **odd**-sized subsets of A whose elements add up to a multiple of 3.

You can assume that multiplying two k bit integers can be done in $O(k \log k)$ time (see Harvey and van der Hoeven).

Hint 1: Note that the number of such subsets can scale exponentially with n, so we cannot assume that arithmetic operations take $O(1)$ time. To bound the cost of adding and multiplying, show that the number of subsets of a size n array can be stored in an n-bit integer.

Hint 2: Try keeping track of the number of odd and even sized subsets that sum to 0, 1, or 2 mod 3 using divide and conquer. How do you combine the subproblems of size k in $O(k \log k)$ time?

Solution: We divide the array in half and recursively compute `num_odd[m]` and `num_even[m]` for $m = 0, 1, 2$. These denote the number of odd-size and even-sized subsets that have sum m modulo 3 respectively. To combine, we consider all the 6×6 pairs of values from the left and right subproblems and add them into the correct entry. The final output will then be `num_odd`[0].

Pseudocode for calculating `num_odd` array in the combine step:

```
Initialize num_even[$m$] and num_odd[$m$] to 0 for $m = 0, 1, 2$
for $i = 0, 1, 2$ do
    for $j = 0, 1, 2$ do
        $m \leftarrow (i + j) \mod 3$
        num_odd[$m$] $\leftarrow$ num_odd[$m$] + left_num_even[$i$] $*$ right_num_odd[$j$]
        num_odd[$m$] $\leftarrow$ num_odd[$m$] + left_num_odd[$i$] $*$ right_num_even[$j$]
```

We can calculate `num_even` analogously using $O(1)$ number of arithmetic operations.

Runtime Analysis The combine step has $O(1)$ multiplication and additions. Each operation takes $O(k \log k)$ time for a subproblem of size k. Since we divide into half at every level, the runtime is given by the recurrence, $T(n) = 2T(n/2) + O(n \log n)$. Expanding out the recurrence,

$$\begin{aligned} T(n) &= 2T(n/2) + cn\log n && (1)\\ &= \sum_{i=1}^{\log n} 2^i \cdot c(n/2^i)\log(n/2^i) && (2)\\ &\le \sum_{i=1}^{\log n} 2^i \cdot c(n/2^i)\log n && (3)\\ &= \sum_{i=1}^{\log n} cn\log n && (4)\\ &= cn(\log n)^2 && (5) \end{aligned}$$

(for some constant c). Thus, the runtime is $O(n(\log n)^2)$.

4 The Resistance

We are playing a variant of The Resistance, a board game where there are n players, s of which are spies. In this variant, in every round, we choose a subset of players to go on a mission. A mission succeeds if the subset of the players does not contain a spy, but fails if at least one spy goes on the mission. After a mission completes, we only know its outcome and not which of the players on the mission were spies.

Come up with a strategy that identifies all the spies in $O(s\log(n/s))$ missions. **Describe your strategy and analyze the number of missions needed**

Solution: Observe that if we partition the players into m disjoint groups and send each group on a mission, at least $m - s$ groups will succeed and we can rule out the players in those groups.

To discard at least half of the players in each iteration, partition them into $2s$ groups. For the missions that succeed, we remove those groups, and split the remaining groups in half to form a new set of groups. We repeat this procedure until each group has one person left, at which point we know they are a spy.

Runtime analysis: In the first iteration of this procedure we have $2s$ groups going on missions. After each group goes on a mission, since there are s spies, at most s of the missions fail, which means after splitting the groups that failed, we have at most $2s$ groups again.

In each iteration, at least half of the players are removed from groups. So we identify the spies after $O(\log(n/s))$ iterations. So the total number of missions needed is $O(s\log(n/s))$.

Alternate solution: Divide the players into two groups and send each group on a mission. If a group succeeds then discard those players and if a group fails then recursively use the same strategy on that group until a group of size 1 is reached.

Analysis: After drawing out the recursion tree you'll see that at level i, at most $\min(2^i, s)$ missions can fail and those are the nodes that will branch out to the next level. There are at most $\log n$ levels. Hence, the total number of missions is bounded by,

$$\begin{aligned}\sum_{i=i}^{\log n} \min(2^i, s) &\leq \sum_{i=1}^{\log s} 2^i + \sum_{i=\log s}^{\log n} s && (6)\\ &= O(s) + s(\log n - \log s) && (7)\\ &= O(s \log(n/s)) && (8)\end{aligned}$$

5 Werewolves

You are playing a party game with n other friends, who play either as werewolves or humans. You do not know who is a human and who is a werewolf, but all your friends do. There are always more humans than there are werewolves.

Your goal is to identify one player who is certain to be a human.

Your allowed 'query' operation is as follows: you pick two players as partners. You ask each player if their partner is a human or a werewolf. When you do this, a human must tell the truth about the identity of their partner, but a werewolf doesn't have to (they may lie or tell the truth about their partner).

Your algorithm should work regardless of the behavior of the werewolves.

(a) For a given player x, devise an algorithm that returns whether or not x is a human using $O(n)$ queries. Just an informal description of your test and a brief explanation of why it works is needed.

(b) Show how to find a human in $O(n \log n)$ queries (where one query is taking two players x and y and asking x to identify y and y to identify x).

Hint: Split the group into two groups, and use part (a). What invariant must hold for at least one of the two groups?

Give a 3-part solution.

(c) (**Optional, not for credit**) Can you give a $O(n)$ query algorithm?

Hint: Don't be afraid to sometimes 'throw away' a pair of players once you've asked them to identify their partners.

Give a 3-part solution.

Solution:

(a) To test if a player x is a human, we ask the other $n-1$ players what $x's$ identity is. Claim: x is a human if and only if at least half of the other players say x is a human. To see this, notice that if x is a human, at least half of the remaining players are also humans, and so regardless of what the werewolves do at least half of the players will say x is a human. On the other hand, if x is a werewolf, then strictly more than half of the remaining players are humans, and so strictly less than half the players can falsely claim that $x$is a human.

(b) **Main idea** The divide and conquer algorithm to find a human proceeds by splitting the group of friends into two (roughly) equal sets A and B, and recursively calling the algorithm on A and B: $x = \mathit{human}(A)$ and $y = \mathit{human}(B)$, and checking x or y using the procedure in part (a) and returning one who is a human. If there is only one player left, they are guaranteed to be a human, so return that player.

Proof of correctness We will prove that the algorithm returns a human if given a group of n players of which a majority are humans. By strong induction on n:

Base Case If $n = 1$, there is only one player in the group who is a human and the algorithm is trivially correct.

Induction Hypothesis The claim holds for $k < n$.

Induction Step After partitioning the group into two groups A and B, at least one of the two groups has more humans than werewolves. By the induction hypothesis the algorithm correctly returns a human from that group, and so when the procedure from part (a) is invoked on x and y at least one of the two is identified as a human.

Runtime analysis Two calls to problems of size $n/2$, and then linear time to compare the two players returned to each of the friends in the input group: $T(n) = 2T(\frac{n}{2}) + O(n) = O(n \log n)$ by Master Theorem.

Note to graders: $O(n)$ query solutions, as in part (c), should also get full credit.

(c) **Main idea** Split up the friends into pairs and for each pair, if either says the other is a werewolf, discard both friends; otherwise, discard any one and keep the other friend. If n was odd, use part (a) to test whether the odd man out is a human. If yes, you are done, else recurse on the remaining at most $n/2$ friends.

Proof of correctness After each pass through the algorithm, humans remain in the majority if they were in the majority before the pass. To see this, let n_1, n_2 and n_3 be the number of pairs of friends with both humans, both werewolves and one of each respectively. Then the fact that humans are in the majority means that $n_1 > n_2$. Note that all the n_3 pairs of the third kind get discarded, and one friend is retained from each of the n_1 pairs of the first kind. So we are left with at most $n_1 + n_2$ friends of whom a majority n_1 are humans. It is straightforward to now turn this into a formal proof of correctness by strong induction on n.

Runtime analysis In a single run of the algorithm on an input set of size n, we do $O(n)$ work to check whether f_1 is a human in the case that n is odd and $O(n)$ to pair up the remaining friends and prune the candidate set to at most $n/2$ players. Therefore, the runtime is given by the following recursion:

$$T(n) = T\left(\frac{n}{2}\right) + O(n) = O(n) \text{ by Master Theorem.}$$

Alternative solution: Pick a player at random and use part (a) to check whether they are human, if not then repeat the process again. Since at least half the players are humans, the **expected** number of players we will have to check until we find a human is $O(1)$. Since the check requires $O(n)$ queries, the overall solution also uses $O(n)$ queries **in expectation**.

6 [Coding] Quickselect

For this week's homework, you'll implement the quickselect algorithm in a python jupyter notebook called `quickselect.ipynb`. There are two ways that you can access the notebook and complete the problems:

1. **On Local Machine**: `git clone` (or if you already have it, `git pull`) from the coding homework repo,

   `https://github.com/Berkeley-CS170/cs170-fa23-coding`

   and navigate to the `hw02` folder. Refer to the `README.md` for local setup instructions.

2. **On Datahub**: Click here and navigate to the `hw02` folder if you prefer to complete this question on Berkeley DataHub.

Notes:

- *Submission Instructions:* Please download your completed `quickselect.ipynb` file and submit it to the gradescope assignment titled "Homework 2 Coding Portion".
- *OH/HWP Instructions:* Designated coding course staff will provide conceptual and debugging help during office hours and homework parties.
- *Academic Honesty Guideline:* We realize that code for some of the algorithms we ask you to implement may be readily available online, but we strongly encourage you to not directly copy code from these sources. Instead, try to refer to the resources mentioned in the notebook and come up with code yourself. That being said, we **do acknowledge** that there may not be many different ways to code up particular algorithms and that your solution may be similar to other solutions available online.

quickselect

September 6, 2023

0.0.1 Step 0: Setup

If you're using Datahub:

- Run the cell below **and restart the kernel if needed** ##### If you're running locally:
- Make sure you've activated the conda environment: `conda activate cs170`
- Launch jupyter: `jupyter lab`
- Run the cell below **and restart the kernel if needed**

1 Quickselect

In this notebook, we will implement the quickselect algorithm. The quick select algorithm is an efficient divide and conquer algorithm for finding the k-th smallest element of an unsorted array. We will first demonstrate a naive solution for this problem, then implement and compare it with quick select.

The full algorithm is detailed here https://people.eecs.berkeley.edu/~vazirani/algorithms/chap2.pdf#page=10.

```
[ ]: # Install dependencies
     !pip install -r requirements.txt --quiet
```

```
[ ]: import otter
     assert otter.__version__ >= "4.2.1", "Please reinstall the requirements and␣
      ↪restart your kernel."

     grader = otter.Notebook("quickselect.ipynb")
     import numpy.random as random
     rng_seed = 42
```

1.1 Naive Solution

The naive solution to the problem is as follows: 1. sort the input array 2. return the k-th element

```
[ ]: def naive_select(array, k):
         sorted_array = sorted(array)
         return sorted_array[k]
```

We can run this on a few test cases to check that it works.

```
[ ]: array1 = [6, 1, 3, 5, 7, 5, 8]
     array2 = [10, 4, 7, 2, 8, 9]
     array3 = [12, 4, 6 ,8 ,3, 4, 2]

     print("The smallest element of ", array1, " is ", naive_select(array1, 0))
     print("The median element of ", array2, " is ", naive_select(array2,␣
      ↪len(array2)//2))
     print("The largest element of ", array3, " is ", naive_select(array3,␣
      ↪len(array3) - 1))
```

1.1.1 Runtime analysis

This algorithm first sorts the array, which would take $O(n \log n)$ assuming quicksort is used and indexing into the array takes $O(1)$. Thus, the algorithm takes $O(n \log n)$ overall.

This is not a very efficient solution; however, since it is unnecessary to sort the entire array to simply find one element. Thus, we will next explore quickselect.

1.2 Write a D&C Solution

Quickselect is a randomized divide and conquer algorithm which is able to solve this problem in expected $O(n)$ time. See https://people.eecs.berkeley.edu/~vazirani/algorithms/chap2.pdf#page=11 for a detailed runtime analysis. The main idea of the algorithm is as follows:

1. Randomly select a pivot element from the array
2. Partion the array into three partitions (the elements less than, equal too, and greater than the pivot)
3. Recurse on the partition which must contain the k-th smallest element With this in mind, please implement the quickselect algorithm by replacing the elipses "…" with your solution.

```
[ ]: def quick_select(array, k):
         """
         Returns the k-th smallest element of the array.

         Args:
             array (List[int]): List of integers to select from.
             k (int): The order statistic to select (0 is the smallest, len(array)-1␣
      ↪is the largest).
         """

         # randomly pick a pivot
         v = random.choice(array) # SOLUTION

         partition1 = []
         partition2 = []
         partition3 = []
         # assign each element to the appropriate partition
         # BEGIN SOLUTION
```

```
    for n in array:
        if n < v:
            partition1.append(n)
        elif n == v:
            partition2.append(n)
        else:
            partition3.append(n)
    # END SOLUTION

    # recurse on the partition which contains the k-th smallest element
    """ # BEGIN PROMPT
    if ...:
        ...
    elif ...:
        ...
    else:
        ...
    """; # END PROMPT
    # BEGIN SOLUTION NO PROMPT
    if k < len(partition1):
        return quick_select(partition1, k)
    elif k < len(partition1) + len(partition2):
        return v
    else:
        return quick_select(partition3, k - len(partition1) - len(partition2))
    # END SOLUTION
```

We can then test the function on the same set of arrays as before to check for correctness.

```
[ ]: array1 = [6, 1, 3, 5, 7, 5, 8]
     array2 = [10, 4, 7, 2, 8, 9]
     array3 = [12, 4, 6 ,8 ,3, 4, 2]

     print("The smallest element of ", array1, " is ", quick_select(array1, 0))
     print("The median element of ", array2, " is ", quick_select(array2,␣
      ↪len(array2)//2))
     print("The largest element of ", array3, " is ", quick_select(array3,␣
      ↪len(array3) - 1))
```

1.2.1 Verification

For a more thorough test, we can check that quick_select returns the same elements as naive_select for a large number of random arrays. Often times, naive algorithms are much simpler to implement and verify than more efficient algorithms. Thus, one way to verify the correctness of our implementation is to compare it to the naive implementation which we know to be correct.

The following block of code generates a 1000 random arrays and 1000 random values for k, and checks that both solutions return the same answer each time. If your implementation is correct, the following code will print “success”.

This cell is not used for grading, feel free to modify it to help you debug

```
[ ]: for i in range(1000):
         array = random.randint(1000, size = 1000)
         k = random.randint(1000)

         assert naive_select(array, k) == quick_select(array,k)

     print("success")
```

Now, check your implementation against the autograder’s test cases:

Points: 5

```
[ ]: grader.check("q1")
```

1.3 Submission

Make sure you have run all cells in your notebook in order before running the cell below, so that all images/graphs appear in the output. The cell below will generate a zip file for you to submit.

```
[ ]: grader.export(pdf=False, force_save=True, run_tests=True)
```