CS4102 - ALGORITHMS - SPRING 2022

DUE FRIDAY, MARCH 4, 2022 AT 11:30 PM VIA GRADESCOPE

AADV(SSB3VK)

Collaboration Policy: You are encouraged to collaborate with up to 3 other students, but all work submitted must be your own *independently* written solution. List the computing ids of all of your collaborators in the `collabs` command at the top of the tex file. Do not share written notes, documents (including Google docs, Overleaf docs, discussion notes, PDFs), or code. Do not seek published or online solutions for any assignments. If you use any published or online resources (which may not include solutions) when completing this assignment, be sure to cite by naming the book etc. or listing a website's URL. Do not submit a solution that you are unable to explain orally to a member of the course staff. Any solutions that share similar text/code will be considered in breach of this policy. Please refer to the syllabus for a complete description of the collaboration policy.

Collaborators: list collaborators's computing IDs

Sources: Cormen, et al, Introduction to Algorithms. *(add others here)*

PROBLEM 1 *Bazinga!*

Theoretical Physicist Sheldon Cooper has decided to give up on String Theory in favor of researching Dark Matter. Unfortunately, his grant-funded position at Caltech is dependent on his continued work in String Theory, so he must search elsewhere. He applies and receives offers from MIT and Harvard. While money is no object to Sheldon, he wants to ensure he's paid fairly and that his offers are at least the median salary among the two schools' Physics departments. Therefore, he hires you to find the median salary across the two departments. Each school mantains a database of all of the salaries for that particular school, but there is no central database.

Each school has given you the ability to access their particular data by executing *queries*. For each query, you provide a particular database with a value k such that $1 \leq k \leq n$, and the database returns to you the k^{th} smallest salary in that school's Physics department.

You may assume that: each school has exactly n physicists (i.e. $2n$ total physicists across both schools), every salary is unique (i.e. no two physicists, regardless of school, have the same salary), and we define the *median* as the n^{th} highest salary across both schools.

1. Design an algorithm that finds the median salary across both schools in $\Theta(log(n))$ total queries.

   Solution: *A key consideration for this problem is that we are only constrained by the number of accesses we can make. So, let's just make a $log_2(n)$ number of accesses to both sets to get a representative sample of values from both sets. This means that we have to employ some sort of search algorithm that uses a log-based procedure.*

   *The other property of medians that we have to acknowledge is that a median is in the **middle** of all the elements. So we will know we have the median when there are n elements on either side of the median value that we selected.*

```
salary(sizeOfDepartment){

    index1 = sizeOfDepartment/2;
    index2 = sizeOfDepartment/2;

    for ( int i = 2; i < sizeOfDepartment; i*=2 ) {
        median1 = department1[index1]
        median2 = department2[index2]

        if ( median 1 > median 2 ) {
```

```
            index1 += -1 * sizeOfDepartment/i
            index2 += 1 * sizeOfDepartment/i
        } if ( median2 > median1 ) {
            index1 += 1 * sizeOfDepartment/i
            index2 += -1 * sizeOfDepartment/i
        }


    }

    if (median1 > median2) {
        return median2;
    }

    return median1;

}
```

2. State the complete recurrence for your algorithm. You may put your $f(n)$ in big-theta notation. Show that the solution for your recurrence is $\Theta(log(n))$.

   Solution: *The recurrence for this algorithm should be something along the lines of:*

$$T(n) = T(\frac{n}{2}) + 2 \tag{1}$$

   This recursion is a little difficult to identify because it is in a loop but the principle is the same. Per loop we divide the search space in half and do two access operations as well as two divisions.

   To convert this recurrence into an overall runtime analysis, the easiest thing to do is to use the Master Theorem. In this case, Master Theorem Case 2 is the easiest to apply. This is because $f(n) = 2$ which is an element of $\Theta(n^{log_2(1)})$ or equivalently:

$$2 \in \Theta(n^0) \implies 2 \in \Theta(c) \tag{2}$$

   From this, we can conclude that the overall recurrence is:

$$T(n) \in \Theta(n^{log_2(1)} log(n)) \implies T(n) \in \Theta(c * log(n)) \implies T(n) \in \Theta(log(n)) \tag{3}$$

3. Prove that your algorithm above finds the correct answer. *Hint: Do induction on the size of the input.*

   Solution: *We intelligently select which half to eliminate by checking whether one median is greater than the other. If median1 is greater than median2, we know that the elements that are greater than median1 cannot be the median and that none of the elements less than median2 can be the median either and vice versa. But note that this procedure does not eliminate the median that we are currently checking. If we make the right eliminations, we consistently reduce the search space of medians until we end up on the pair medians we need. If we recall our analysis above, we find that the number of*

elements greater than both medians is equal to the number of elements smaller than both medians, in their respective lists.

We will proceed with proof by induction to prove that this algorithm runs correctly in $\log(n)$ *time. But before we go about doing that, there is some groundwork we have to lay. Primarily, we have to ensure that at each level of the iteration, the median is maintained within the search range. Another consideration to make is that I return the lesser of the two "medians" as the median of the entire set. According to the rules, I should return the nth element of the set. Therefore, according to the rules, I return the lesser of the two medians.*

Base Case: *Consider a pair of lists with size 1 each. Here, both median1 and median2 will point to the respective values in both lists. And when we ask for the median, the 1st smallest element in both lists we simply return the smaller of the two elements.*

Inductive Hypothesis: *Assume that salary() is correct for all input sizes k where* $k < n$ *and* $k \geq 1$*.*

Inductive Step: *Prove that salary() is correct for an input list size of n, where* $n > k$*.*

Proof for Inductive Step: *This algorithm finds the pair of medians that are at the closest to the medians of the overall combined list by eliminating half of the remaining elements in both lists at each step. So on either side of our "median" we are already gauranteed to have n/2 elements greater than our median and n/2 elements less than our median. This procedure continues until all but one element in each list remain and each of these elements are in the same but opposite index of the other. The number of elements in list1 that are greater than median1 is equal to the number of elements in list2 that are less than median2 and vice versa.*

We have proved that salary() is correct for all cases on the basis that eliminate elements in such a way that the sum of the elements greater than median1 and median2 is equal to the sum of the elements lower than median1 and median2 in their respective lists. And as we consistently maintain this until we have only one element left, we can guarantee that our algorithm is correct for all values of n.

PROBLEM 2 *Castle Hunter*

We are currently developing a new board game called *Castle Hunter*. This game works similarly to *Battleship*, except instead of trying to find your opponent's ships on a two dimensional board, you're trying to find and destroy a castle in your opponent's one dimensional board. Each player will decide the layout of their terrain, with castles placed on each hill. Specifically, each castle is placed such that they are higher than the surrounding area, i.e. they are on a local maximum, because hill tops are easier to defend. Each player's board will be a list of n floating point values. To guarantee that a local maximum exists somewhere in each player's list, we will force the first two elements in the list to be (in order) 0 and 1, and the last two elements to be (in order) 1 and 0.

To make progress, you name an index of your opponent's list, and she/he must respond with the value stored at that index (i.e., the altitude of the terrain). To win you must correctly identify that a particular index is a local maximum (the ends don't count), i.e., find one castle. An example board is shown in Figure 1. [We will require that all values in the list, excepting the first and last pairs, be unique.]

0	1	4	23	18	14	15	13	1	0
0	1	2	3	4	5	6	7	8	9

Figure 1: An example board of size $n = 10$. You win if you can identify any one local maximum (a castle); in this case both index 3 and index 6 are local maxima.

1. Devise a strategy which will guarantee that you can find a local maximum in your opponent's board using no more than $O(\log n)$ queries, prove your run time and correctness.

   Solution: *A strategy to determine the local maximum is to use a strategy similar to a binary search algorithm. But instead of comparing the values of the elements, we check whether the prior and post elements are greater than or lower than the current element and deciding the next value we check based off that.*

```
castleHunterInit(list)
    return castleHunter(0, list.length() - 1, list)

castleHunter(mindex, maxdex, list)
    index = (maxdex - mindex)/2

    if ( list[index] < list[index + 1] )
        return castleHunter(index, maxdex, list)
    elsif ( list[index] < list[index - 1] )
        return castleHunter(mindex, index, list)
    else: //the case where list[index] > list[index - 1] &&
          //list[index] > list[index + 1] a local max
        return index
```

   This has a O(log n) runtime by definition. Either we divide the sample space in half (by passing the index as either the higher or lower index or we simply return the current element. Now I will prove that this algorithm is correct. A key part of this algorithm is that we use the "trajectory" of elevations to select our next element and at each index we test, we ensure that the index is not a local max. The trajectory of elevations is an essential concept hear because for each index we check, we determine whether or not a side is higher and simply go in the direction where the side is higher. This guarantees that we will find a local maximum because by definition, a local maximum will be found when the point has a value that is greater than the values of the prior and post points. This implies that the trajectory towards this point (from either side) has to have a positive trajectory. In a visualization, this would appear as a "mound" with the point that we want to identify as the local max at the center of the mound. Which is exactly what this algorithm does. Regardless of where the peak of the mound is, we can bounce around the mound and consistently select the direction that leads to an increase in our value, much like a gradient ascent algorithm from machine learning.

   Base case: *Consider a list of size 5 (the smallest size list). Here we identify our local maximum immediately because of the fact that we can only have unique values except for the two values at the beginning and end of the list which are 0, 1 and 1, 0 respectively. Further, because we can not have any negative, non-unique values our middle value is > 1 and is our local maximum.*

   Inductive Step: *The algorithm will calculate the local maximum to be either between the current median and the beginning or between the current median and the end of the list. The way we determine which sublist to check is if the current median either has a value greater than it on the right or on the left and proceeding in the direction of the greater value. The reason we proceed in the direction of the greater value is because proceeding in said direction ensures that at least the subsequent element (median + 1) is a local maximum whereas proceeding in the opposite direction (where the element is lower) has no such guarantee.*

   We keep this decision-making up until we find an element where the surrounding pair of elements are lower than the index element that we selected. We subsequently return the index element that we selected.

Thus, we have proved that castleHunter is correct for all cases where the index element is a local max, a local min, on a right-increasing slope, or on a left-increasing slope, and our algorithm is correct for all n.

End of Proof

2. Now show that $\Omega(\log n)$ queries are required by *any* algorithm (in the worst case). To do this, show that there is a way that your opponent could dynamically select values for each query as you ask them, rather than in advance (i.e. cheat, that scoundrel!) in such a way that $\Omega(\log n)$ queries are required by *any* guessing strategy you might use.

   Solution: *An adversarial actor that is able to dynamically select values in each query as I ask them would force my algorithm to take* $\Omega(\log n)$ *queries because per query, we effectively halve our sample space. So even in the worst case scenario, regardless of the actual values, we are gauranteed to terminate in* $\Omega(\log n)$ *queries. On the basis of halving our sample space (passing in the median index value on each successive castleHunter call) we can ensure that we terminate in* $log_2(n)$ *steps.*

   If we recall the tree structure from our lecture, we found that to sort a list we would take $\log(n!)$ *time in the worst case. To adapt that idea to our solution where we can have our sample space of n possible solutions in each case, we find that because we divide our search space in half every time and only have n possible solutions, we can conclude our program in* $\log(n)$ *time in the worst case.*

PROBLEM 3 *Goldilocks and the n Bears*

BookWorld needs your help! Literary Detective Thursday Next is investigating the case of the mixed up porridge bowls. Mama and Papa Bear have called her to help "sort out" the mix-up caused by Goldilocks, who mixed up their n bear cubs' bowls of porridge (there are n bear cubs total and n bowls of porridge total). Each bear cub likes his/her porridge at a specific temperature, and thermometers haven't been invented in BookWorld at the time of this case. Since temperature is subjective (without thermometers), we can't ask the bears to compare themselves to one another directly. Similarly, since porridge can't talk, we can't ask the porridge to compare themselves to one another. Therefore, to match up each bear cub with their preferred bowl, Thursday Next must ask the cubs to check a specific bowl of porridge. After tasting a bowl of porridge, the cub will say one of "this porridge is too hot," "this porridge is too cold," or "this porridge is just right."

1. Give a *brute force* algorithm for matching up bears with their preferred bowls of porridge which performs $O(n^2)$ total "tastes." Prove that your algorithm is correct and that its running time is $O(n^2)$.

   Solution: *A brute force algorithm for matching up bears with their preferred bowl of porridge is to simply iterate over each bowl of soup for each bear eliminating a bowl/bear duo when the bear determines that the bowl is to its liking. This would be a* n^2 *algorithm as there are n bears and n goals.*

   The reason that this is correct is because we are gauranteed to find a pairing of b, s for each pair simply off of the fact that we compare every single element of b to (potentially) every single remaining element of s.

   The reason that this is $O(n^2)$ *is because we compare each b to s, s-1, s-2 ... 1. This results in a maximum of:*

$$\frac{n(n-1)}{2}$$

   comparisons for this algorithm. Now for the algorithm in question:

```
Goldilocks(stack bears, list soups ) {
    set answers;
    while ( ! bears.empty() ){
        b = bears.pop();
        for s in soups{
            if ( b.justRight(s) ) {
                delete s;
                answers.push(b, s);
                break;

            }

        }

    }

}
```

2. Give an *randomized* algorithm which matches bears with their preferred bowls of porridge and performs expected $O(n \log n)$ total "tastes." Prove that your algorithm is correct. Then, intuitively, but precisely, describe why the expected running time of your algorithm is $O(n \log n)$. *Hint: while this is not a sorting problem, your understanding of the sorts we've discussed in class may help when tackling this problem.*

   Solution: The easiest way to go about this is to think that each bear is a median. And we can use each bear to divide the space of soups. And then we can use the median soup (that corresponds to the selected partition bear) to then partition the set of bears. This will enable us to use a implementation that echos a quicksort implementation. Further, by constantly dividing the sample size (preferably in half) we can ensure that this algorithm has a $O(n \log n)$ number of taste tests.

```
GoldilocksSetup( list bears, list soups, set Answers ) {
    Answers = []
    return Goldilocks( bears, soups, Answers);

}

Goldilocks( list bears, list soups, set Answers ) {

    b = bears.random();
    listLeftSoup[]
    listRightSoup[]

    listLeftBears[]
    listRightBears[]

    medianSoup

    for s in soups:
        if ( b.temp(s) == tooCold ) {
            listLeftSoup.push(s);
        } elsif ( b.temp(s) == tooHot ) {
```

```
            listRightSoup.push(s);
        } elsif ( b.temp(s) == justRight) {
            medianSoup = s;
            Answers.push(b, s)
        }

    for b in bears:
        if ( b.temp(medianSoup) == tooCold ) {
            listLeftBears.push(b);
        } elsif ( b.temp(medianSoup) == tooHot ) {
            listRightBears.push(b);
        }

    Goldilocks( listLeftBears, listLeftSoup, Answers)
    Goldilocks( listRightBears, listRightSoup, Answers)

    return Answers;

}
```

We can use this strategy reminiscient of the quickselect by recursively subdividing both problems in half and then reapplying the strategy. By randomly seleting a bear prior to starting, we can ensure that we select a decent bear and nearly guarantee that we split the search space into fractional parts for each level of the recursion.

To prove that this runs in O(n log n) time, we can separately analyze each part of the algorithm. The first part (GoldilocksSetup) performs no real operations. Goldilocks produces 2 problems of approximately n/2 size (based off of its divide setup) which require 2n operations in the divide step. This can all be summarized in the following recurrence relationship:

$$T(n) = 2 * T(n/2) + 2n$$

Using the 2nd case of the master theorem, we find that this recurrence relation is $\Theta(n * log(n))$.

$$f(n) = 2n$$
$$2n \in \Theta(n^{log_2(2)} \implies 2n \in \Theta(n)$$

Now that we've verified the first part of the 2nd case,

$$T(n) \in \Theta(n^{log_2(2)} * log(n)) \implies T(n) \in \Theta(n * log(n))$$

Now to prove the correctness of this algorithm, we have to prove the base case (one bear and one soup) as well as ensure that Goldilocks properly partitions the Bears and their corresponding Soups into the same "side". The base case is trivial, when there is only one bear and one soup and the bear states that the soup's temperature is just right, then we know that the we have the right match. Now I will prove that the algorithm properly partitions both bears and their corresponding soups into the right "side" of the list.

Notice that at the beginning of each "Goldilocks" call, we pick the bear that is at the middle of the set of bears that we are passed in. Because the list was initially sorted, each push command

into the listLeftBears and listRightBears preserves the initial sorting of the algorithm. Therefore selecting the middle index is guaranteed to give us a near perfect (50%) partition every time. Further, by using this middle index bear to find the middle index soup and then sorting the respective list of bears/soups by the respective middle index unit of either ensures that all bears/soups with heat preferences/heats greater than the given middle index will end up in the listRight and all with a lower heat preference/heat end up in the listLeft.

Base case: *Consider a pair of lists of size 1, one for soups and one for bears. Assuming that both bear and soup are a matched pair, the algorithm will return the pair of bear and soup.*

Inductive Hypothesis: *Assume that Goldilocks() is correct for a pair of input lists of size n, where* $n > k$.

Proof for Inductive Step: *The algorithm will use the midpoint of bears to partition the list of bears, find the corresponding soup (which will turn out to be the median soup), and partition the list of soups. To do so, we use the bear that we randomly selected to partition the soups depending on whether or not the soup was too hot or too cold. Too hot soups go into the right sub list and vice versa. Eventually we find the soup that corresponds to this bear. We then use the corresponding soup to sort the bears, sorting bears with a too cold preference of the soup to the right sub list and vice versa. This ensures that the pair of soup/bear right/left sublists are gauranteed to contain a pairing between bears and soups.*

The subsequent call will pass in the corresponding list of bears with the corresponding list of soups where each list is guaranteed to have a corresponding pair in the other list. As this procedure continues, we divvy up the lists further and further until we reach the base case and all bear/soup pairs are accounted for.

We have proved that Goldilocks() is correct for all cases of of bear/soup lengths and organizations. Thus we have proved the Inductive Step, and the algorithm is correct for all values of n.

End of Proof