Assignment 4

CSCI 270

Paul Kim | pbkim@usc.edu | 1723717002

Question 1 . . . . . **1**

Question 2 . . . . . **3**

Question 3 . . . . . **8**

Question 1

(1) [2+2+2+2+2=10 points]
For each of the following recurrences, using a method of your choice (Master Theorem or something else), determine the solution. When possible, give a precise (rather than big-O) solution. If your method of choice is guessing, then you need to provide a proof of your claimed solution. You do not have to worry about rounding up or down, or whether numbers are integers.

(a) $T(1) = 1$ and $T(n) = 7T(n/2) + \Theta(n^2)$ for $n > 1$.
(b) $T(1) = 1$ and $T(n) = T(n-1) + n$ for $n > 1$.
(c) $T(1) = 1$ and $T(n) = 4T(n/2) + \Theta(n^2)$ for $n > 1$.
(d) $T(1) = 1$ and $T(n) = 3T(n/5) + \Theta(n \log n)$ for $n > 1$.
(e) $T(1) = 1$ and $T(n) = 2T(n-1) + 1$ for $n > 1$.

*Master's Theorem takes the general form: T(N) = A*T(N/B) + f(N). *N is the size of input, A is the of sub-problems in the equation, (N/B) is the size of each sub-problem, f(N) is the cost of work outside of the recursive call.

*We assume that A >= 1 and B > 1 are constants, and f(N) is an asymptotically positive function.

Answer:

1. **Master's Theorem:** T(N) = A*T(N/B) + f(N)
   A=7, B=2, f(N) = N^2, $\log_2 7 = 2.8$, C=2.
   $\log_B A$ > C... **(CASE 1 of Master Theorem)**.
   Since 2.8 > 2, **T(N)= Θ ($N^{2.80735...}$)**

2. **(Not in the form of Master's Theorem)**. We know that this recurrence is a common recurrence that yields the well-known identity of the sum of natural numbers. We know from

previous classes that the formula yields:

$$\sum_{i=1}^{n} i = (n(n+1))/2$$

Formally to show the pattern:

*We are building out the sequence. (Arithmetic Series)

T(N) = T(N-1) + N

T(N-1) = T(N-2) + N - 1

T(N-2) = T(N-3) + N - 2

...

T(N) = 1 + 2 + 3 + ... + N = N(N+1)/2.

And from this summation, we can get an exact running time of **T(N) = N(N+1)/2.**

***Selection Sort Recurrence Relation (CSCI 104, CSCI 170).**

3. **Master's Theorem:** T(N) = A*T(N/B) + f(N)

   A=4, B=2, f(N) = N^2, $\log_2 4 = 2$, C=2.

   $\log_B A$ = C... **(CASE 2 of Master Theorem)**.

   Since 2 = 2, **T(N)**= Θ (N^2*log N)

4. **Master's Theorem:** T(N) = A*T(N/B) + f(N)

   A=3, B=5, f(N) = N log N, $\log_5 3 = 0.68$, K=1, C=1.

   $\log_B A$ < C... **(CASE 3 of Master Theorem)**.

   Furthermore, it is true that 3*F(N/5) <= k*F(n) for some constant k < 1, implying that the total is dominated by F(n). We can see that all these conditions hold true, and F(N) dominates.

   *Intuition is that the total is dominated by F(n), (Parent node does more work than the children node).

   Since 0.68 < 1, **T(N)**= Θ (N log N)

5. **(Not in the form of Master's Theorem)**

   Proof by Substitution: (We can derive a pattern by doing this)

   T(N) = 2T(n-1)+1 (*Substitute T(n-1) for 2T(n-2)+1)

   T(N) = 2(2T(n-2)+1)+1

   *T(N) = (2*2)T(n-2)+2+1 (*Substitute T(n-2) for 2T(n-3)+1)

   T(N) = (2*2)(2T(n-3)+1)+2+1

   T(N) = (2*2*2)T(n-3)+4+2+1 (Here we can start to see a pattern)

   ...

   We can derive an exact running time of **T(N)** = 2^n-1

   *We add the -1 to the end because T(1) = 1. When N = 1.

Question 2

You are given a sorted (from smallest to largest) array b of n real (floating point) numbers, as well as a target number x. The array may contain the same number multiple times. Your goal is to count how often x occurs in b. Give and analyze an algorithm for this problem that runs in time $O(\log n)$.

[Note: Finding an algorithm here is probably going to be fairly easy. The important thing is to be precise in your correctness proof.]

Answer:

*We are given a sorted array "b" of n real (floating point) numbers, as well as a target number x. The array may contain the same number multiple times. Your goal is to count how often x occurs in b. Algorithm must run in O(log n).

*Brute Force Solution (Linear Search) achieves O(n), we can just iterate from i = 0, to the end of the array, and count how much times the target number occurs.

Binary Search Algorithm with Modifications.

***The intuition is to use Binary Search to find the first occurrence and the last occurrence of a target number, and return one more of the differences between the two indices/values. We can do this because the array is sorted, so know that every number between the first occurrence and last will be the same number. *We achieve O(log n) time by using Binary Search that reduces the search space for each search.**

To Formally State the Algorithm:

***We are given an array of integers (vector<double> numbers), and a target value (double target)... *It really doesn't make a difference if floating point number or integer.**

IF (numbers.size() == 0) *(*Base Case (No possible occurrence could exist))*

RETURN 0;

int left = 0

int right = numbers.size()-1

int leftmostIndex = -999 *(*Capture first occurrence of element, -999 is an arbitrary number)*

int rightmostIndex = -999. *(*Capture last occurrence of element, -999 is an arbitrary number)*

bool found = false; *(*To check if target exists)*

**Intuition is that we will be running regular Binary Search to find the left most index of the target (if it exists)*

WHILE (left < right) *(*We want to obtain the left most element of the target in this first iteration of Binary Search)*

int mid = (left+right)/2

IF (numbers[mid] >= target) *(*Checks if element is equal or in the left sub-array, if so, we shift the right pointer)*

right = mid

ELSE *(*Target is greater than the mid, then must be in the right sub-array, since array is sorted)*

left = mid+1

IF (numbers[left] == target) *(*Our Binary Search algorithm returns the left most position of target, a check for correctness, if this is not true, the element did not exist and the boolean is still set to false)*

found = true

leftmostIndex = left ***(*Now we captured our first occurrence of the element here)***.

right = numbers.size()-1 **Reset right parameter, now we must finish*

**Intuition here is now we want to capture the last occurrence of the target element, if it exists.*

**It is important to note that the pointer to left, does not change, and stays at the left-mostIndex. However, the right most pointer is re-initialized to the max*

WHILE (left < right) *(*We want to obtain the right most (last) element of the target in this second iteration of Binary Search).*

int mid = ((left+right)/2)+1; *(*Before, our mid count was biased toward the left (we always rounded down, we now +1 to the mid index to solve this issue to keep the iterator moving.)*

IF (numbers[mid] > target) *(*Must be in the left sub-array, reduce search space)*

right = mid-1

ELSE *(*If element is equal or in the right sub-array, we must search for the right most element.)*

left = mid

rightmostIndex = right; *(*We now captured our last occurrence of our element in this index), we don't need to change the boolean because it would've already set the first condition if target existed.)*

IF (found == false) *(*Element was never found (No Count))*

RETURN 0

ELSE

int solution = (rightmostIndex - leftmostIndex) + 1

**By capturing the first occurrence and last occurrence of the target element, and since we know the fact that the array is sorted, then we can use basic arithmetic in constant time to achieve the count.*

RETURN solution

TOTAL RUN-TIME: O(log n) *We know that the derived runtime of Binary Search is O(log n), in our algorithm we are basically running Binary Search twice to capture two indexes, however, we can drop the constants, and thus our algorithm runs in this time.

**Because the algorithm entails the running of two Binary Search, the runtime is basically O(2log N) where it can be simplified to O(log N) dropping the constants.*

Proof:

*The arithmetic that captures the count of the target element is correct by inspection because we know that the array is sorted in increasing order. For example, let us be given an array [1,2,3,3,3,3,4,5], and our target number is 3. If we say that our algorithm works as intended, then the first occurrence of 3 lies in index 2 (leftmostIndex) and the last occurrence of 3 lies in index (5). By our arithmetic we yield 4: (5-2) + 1 = 4. By inspection, we can see that there are 4 elements in the array. If there is only one occurrence of the target element, then the boolean condition of found should still be set, and both rightmost and leftmost indexes should be the same, and therefore, only 1 element will be counted by the plus 1 count at the end. This is proved enough. ALGORITHM IS SORTED!

*We can use the generic Proof of Correctness of Binary Search with a few modifications to determine that the algorithm correctly finds the first occurrence of the target, and the last occurrence of the target.

Correctness of Binary Search:

1.) If the leftmostIndex returned by the first call to the iterative Binary Search is equal to the target, then we set the boolean condition (found) to true. We will use this condition to accurately determine if the target exists or not, and if we should return 0, or follow through with the arithmetic.

2.) If the algorithm does not set the boolean to true, then the target was never found in the first call to Binary Search, and therefore, the target number does not exist. We return 0 in this case.

*We assume the array is sorted from left to right in increasing order.

*Key observation is that binary search works in a divide and conquer fashion, calling itself on arguments to make the search space becomes always smaller. That is why we achieve the O(log n) time.

Proof by Induction on (N = R+1-L):

Base Case: If the vector/array given as input has a size of 0 (is empty), we return 0. *(*It is an empty array.)*

Induction Step (Strong Induction): (Very similar to proof from Kempe's Lectures)

- For some n, for all k <= n, if Binary Search is called on inputs where right - left = n, then the specified index returned is correct.

- Let us consider n+1, so that R > L.

First Call to Iterative Binary Search (Our Goal/Inductive Hypothesis, is the algorithm captures the first occurrence of the target number:

- Algorithm computes the mid point (biased toward the left because it is always rounded down).

1.) If **target <= numbers[mid]**, then the algorithm sets the right pointer to the mid index. We do this because it must be true that either the mid index yields the target number we are trying to find, or the target number must exist in the left sub array since the input is sorted). Since we are trying to obtain the leftmostIndex, if the mid index yields the target number, there is potential that there might be more instances of the target leftward, we must further check the left sub array for other (previous) occurrences of the target to capture the left most point.

Because we set the right index to mid, we run this iterative searching algorithm again on a smaller input range, so we can apply the inductive hypothesis to this iteration again. (Because the sub-array is sorted in increasing order).

a.) If the iterative call yields the same condition,**target <= numbers[mid]**, the same thing will happen and the right pointer will move closer to the left pointer, thus reducing the search space and continuing to search for possible target values to the left.

b.) If target > numbers[mid], then intuitively, it must be so that the newly calculated midpoint of the new pointers of the Left and Right index, went ”too far”, and did not equal the target (target is greater, and thus lies to the right of mid, since sorted). Thus here, we change the left pointer to be equal to the midpoint+1. Here we are reducing the search space again, and calculating a new mid value.

c.) If the iterative call exits, then we know that we have searched for the most leftmost (possible) index of the target number. We know that the right pointer is now exhausted, because Right < Left. And we know that we checked every mid index if it is the target, and changed pointers accordingly. If the mid index yielded a value equal to or less than the target, we kept searching left to find another occurrence (dividing the search space). If the mid index yielded a value greater than the target, we know that the target must’ve been to the right, thus we had to change the left pointer to not mid, but rather mid+1. At every iteration of searching in the new smaller array, we changed the left and right pointers accordingly to narrow in on each other until they both stopped and equalled to the left index (at this point, left = right, and the while condition was not satisfied (right > left). Since we are also searching for the left most index, we don’t need to care about any instance of the target in the right index because the array is sorted. What this means is that the first call to this search, is biased toward the left.

2.) if **target > numbers[mid]**, then it is the same proof that we did above but to the

right sub-array. Basically if the condition states that the target is greater than the mid, the target must/could be in the right sub-array (Array is sorted). When this instance is hit, we change the left pointer inward towards the right pointer and calculate a new mid to check for the target and change pointers accordingly, with the similar idea to the proof above. The following cases will be the same as well until the left and right pointers are exhausted.

*This is the Proof of Correctness that the Binary Search algorithm returns and exhaustively checks for the left most (first occurrence) of an element. We then check if this left value is indeed is equal to the target, and if so, we set the boolean to true, leading to further implications down the algorithm.

Second Call to Iterative Binary Search (Our Goal/Inductive Hypothesis, is the algorithm captures the last occurrence of the target number:

Basically the proof for this is synonymous to the proof for finding the first occurrence of the target element. The only difference is that we change the operators to actively search for the right most occurrence of the target element.

While right > left,

if the target is less than the number yielded at the mid index, then obviously the target will be to the left, since the array is sorted in increasing order. However, if the target number is equal to or greater than the number yielded by the mid index, this index has the potential to be the rightmostIndex, but we must keep checking for more occurrences in the right-sub array to capture the right most index. This is then proven by the fact that the right and left pointers will keep narrowing at each iteration, and in the end, will successfully return the rightmostIndex when the while condition is exhausted, as proven above.

Another difference we make here is the +1, that we add to calculate the mid index. We do this because in the first iteration, we're somewhat in a way, biased to the left sub-array because we always rounded down (for example, ((9+2)/2) = 5 not 6). Therefore, to capture the rightmostIndex, we must aim our bias towards the right array now by adding the +1 count at the end, so that we can always keep the iterator moving, and "mid" will never be stuck.

At the end, if the target was found from the first iteration of the Binary Search call, then the boolean condition would be set to true, and we would perform the following arithmetic operations. Else, if the condition is false, we return 0 because the target was never found.

*We know that no other value between the indexes are different because the array is sorted. We proved this above.

Question 3

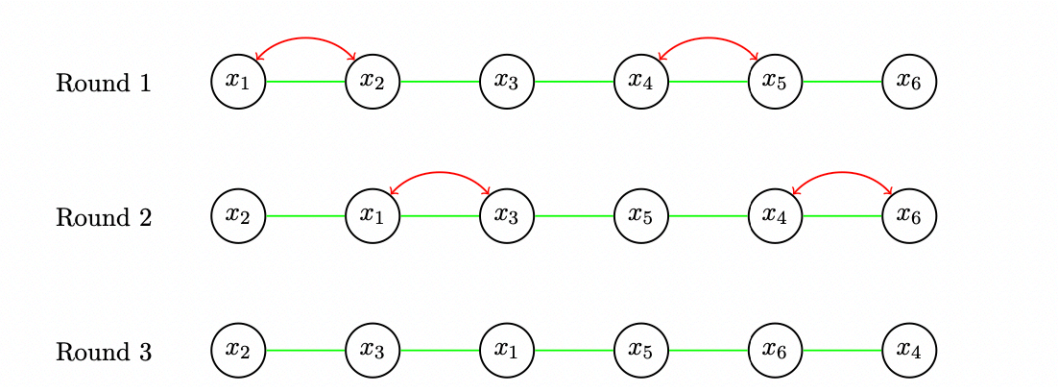
(3) [2+13=15 points]

You have n chairs lined up in a straight line. Each chair is occupied by a person. In each time step, for each chair position i, the person who is currently in chair i meets their two neighbors, in chairs $i-1$ and $i+1$.[1] After all these introductions, there is a "moving" phase: people get to switch chairs with one of their neighbors. These switches must be consistent: for example, you cannot have the people in chairs 1 and 3 both decide to switch with the person in chair 2. People do not *have* to switch; they can also stay put. All the moves happen in parallel, and count as just one round of moving. Then, the next round of introductions starts.

Your goal is to come up with a short sequence of switching instructions such that at the end of your sequence, everyone has met everyone else.

Example: Suppose you have 6 people, $x_1, \ldots, x_6$, initially in chairs $1, \ldots, 6$. Then, x_1 meets x_2, x_2 meets x_1 and x_3, and so on. Now suppose that between rounds 1 and 2, x_1 and x_2 switch places, and x_4 and x_5 switch places. Then, in round 2, x_1 also meets x_3, x_2 does not meet anyone new, x_3 meets x_1 and x_5 (in addition to x_2 and x_4 they met earlier), x_4 meets x_6 (in addition to x_3 and x_5 they met earlier), and so on. Not everyone has met everyone yet, so we need more rounds. For example, we might now switch x_1 with x_3 and x_4 with x_6. This would lead to new meetings in round 3 (but we'd still not be done). We recommend playing around by hand with some coins or other objects. This is illustrated in the figure below, which shows you for each round the positions of the people. Green lines show meetings, and red lines show seat switches.

[1] Except for the persons at the left and right ends; they only have one neighbor, whom they meet.



(a) Prove that every sequence under which everyone meets everyone else takes at least $\Omega(n)$ rounds.

(b) Describe a general (for arbitrary n) sequence under which everyone meets everyone else in $O(n)$ rounds, and analyze it (correctness and number of rounds). To keep your analysis simple, you may assume (if you want) that n is a power of 2.

[Note: If your approach works in $O(n)$ rounds, but your analysis claims $O(n \log n)$, you get partial credit. If your approach actually takes $\Theta(n \log n)$ rounds, you get less partial credit. There is no credit for an $\Omega(n^2)$-round solution.]

[Hint[2]: .hsilpmocca dluow ti tahw dna ,ekat dluow ti gnol woh ,flah thgir eht htiw secalp hctiws ot elpoep fo flah tfel eht teg dluow uoy woh tuoba knihT]

Answer:

1. *We will assume that n is the number of nodes.

   *Big Omega is the lower bound.

   Basic Intuition: Let us consider that we have a set of nodes from (X1, X2,... Xn), and let our first focus be on the Node X1 and Xn which are the farthest nodes apart at each end. You can make these two nodes meet at the middle with n/2 swaps by bringing them to the middle which takes O(n) time. We can swap one by one from the left node, and one by one from the right node simultaneously until they reach the middle. Now the intuition is, if it takes O(n) rounds to just make these two meet, how much time now will it take for X2, and Xn-1 to meet at the half way mark... it would also take O(n) rounds repeating the same process. Extending this, if it takes O(n) rounds to make the two meet, and the next two meet, then it will at least a total of Ω(n) rounds for all nodes to meet each other in the middle. Since you want everyone to meet, then it must be the case that the time everyone takes is at least Ω(n) rounds, total.

2. * Goal is to find an algorithm of sequences of switching places with another node, so that at the end of the sequence, everyone has met each other (adjacent).

   **We can assume that n is a power of two. So there are 2^n nodes.*

   I will state the algorithms in steps and examples nodes: (Very Intuitive Level)

   Step 1: Initialize a set of people from (P1, P2... Pn), (For Example: We have (P1,P2,P3,P4,P5,P6))

   Step 2: Swap every pair of two. (This will result in our set being (P2,P1,P4,P3,P6,P5))

   Step 3: After a swap of every pair, keep the ends stationary, and swap the inside pairs. (This results in, (P2, P4, P1, P6, P3, P5))

   Keep Repeating Step 2 and Step 3 until the ordering is reversed from (Example: 1,2,3,4,5,6 –> 6,5,4,3,2,1.)

   *Basically we have a pattern where we swap every pair from the left to right, and in the next step, we swap the inside pairs, except the outside ones, and we repeat this pattern until the ordering is reversed. What we can see this is doing, is that it is shifting P1 (the first person node) to the right until it reaches the end. However, the general idea we can extract from this, is that the algorithm itself is shifting the left numbers to the right, and the right numbers to the left. When they do this, it is so that every node will meet each other until we know it has met everyone when P1 reaches the end and consequently, the last node (Pn) reaches the beginning. We can use this idea later.

   We know that the algorithm runs best at O(N) time because we can see that for every swap we make, we are moving the first node (P1) closer to the end, until it gets to the end and then we can make the final middle swap. From inspection, at this point we have an inverted ordering, indicating that the shift worked. This takes N steps to do, but while doing so, the algorithm satisfies the condition that everyone has met each other from the general idea above. Thus the algorithm is O(n).

   The general idea of the pattern of the algorithm is that we are making exhaustive swaps of every node pairing that brings node 1 closer to the end, and the last node closer to the

beginning. What this swap/ordering does, is that it introduces new pairings by moving the left half of the people to the right half, and the right half moving to the left half. *Half the people are moving to the left and half the people are moving to the right.

Here we can keep track of the swapping of P1, till the end, and can see that every swap makes P1 closer to the end...

Example:

1,2,3,4,5,6 (Initial)

2,1,4,3,6,5 (Swap 1,2 | Swap 3,4 | Swap 5,6)

2,4,1,6,3,5 (Swap 1,4 | Swap 3,6) (Keep 2 and 5 Stationary)

4,2,6,1,5,3 (Swap 2,4 | Swap 1,6 | Swap 3,5)

4,6,2,5,1,3 (Swap 2,6 | Swap 1,5) (Keep 4 and 3 Stationary)

6,4,5,2,3,1 (Swap 4,6 | Swap 2,5 | Swap 1,3)

6,5,4,3,2,1 (Swap 4,5 | Swap 2,3)

Here we are finished. We go from 1,2,3,4,5,6 and we end where the ordering is completely reversed to 6,5,4,3,2,1. (We see that it takes 6 steps to complete this process and N = 6, therefore, O(N)).

Intuitively, by inspection it is so that every node meets every other node at some point in time. The main idea here is that if the list gets inverted, then it must be the case that every node will get touched, because our algorithm is inherently moving the left half of the people right, and the right half of the people left.

Proof By Induction:

*Keep in mind that there is only pairs/2^i nodes.

Base Case: Two People (P1, P2), the algorithm does nothing as P1 and P2 has meet each other initially.

Strong Induction:

Assume for some K, for all n <= K, that the swaps/pattern performed on the line of nodes works correctly.

By Inductive Hypothesis, we want to perform the swaps suggested by the algorithm above that inherently shifts the first half to the last half, and the last half to the first half. We want to perform swaps that ”reverse” the order of the list. We claim that this takes O(n) rounds/steps.

Induction Step: P(N+2): Let us consider an instance of 4 nodes, (P1,P2,P3,P4). What the algorithm will do is swap P1,P2 and P3,P4... resulting in P2,P1,P4,P3. Here we swap P1,P4 to get P1 closer to the end. We get P2,P4,P1,P3. Here we make the swap P2,P4 and P1,P3 yielding P4,P2,P3,P1. We make the final swap in the middle and we get P4,P3,P2,P1. Some observations we can make here is that the algorithm indeed took 4 steps of swaps to get the solution. The solution yields a reverse ordering of the initial list. Each swap is made P1 closer to the end. These swaps, made every node/person meet with every neighbor, and last but most importantly, the first half went into the second half, and the second half became the first half.

To a wider scope of context, we can think of a bigger problem of the nodes of people, as a divide and conquer problem, and use the key observations we found above from the case of 4 people.

Let us assume a case of 6 people. You can divide the number of problems in this case into two, and now we have two groups of 3 people. However, what we can see here, is that the pattern that the algorithm follows for every iteration, a node from the left group is replacing a node from the right group, and nodes remaining nodes in their current group are rotating because they already met.At the end we will know when we are finished when node P1 reaches the end, and evidently, the left half shifted into the right half meeting everyone, and the right half shifted into the left half meeting everyone. This takes again 6 steps following the path of P1 to the last spot.

Now let us assume an extension and a case of 8 people. You can divide the number of problems into two again, and see that we have two groups of 4 people. However in our inductive step, we already proved what happens in this instance of 4 nodes. We saw that left half shifted into the right half, and vice versa, and since it is a bigger problem, we can combine the solution as say the same thing will happen for the bigger instance. So we can say that in 4 steps, everyone in the left sub-group will have met each other. And in 4 steps, everyone in the right sub-group will have met each other. However, in this case, new nodes from the right group are being introduced into the left group, and new nodes from the left group are being introduced to the right group, and this in sum, will take 8 total steps.

Basically, the underlying idea is that half the people are moving left, and half the people are moving right. We use induction to show the base case, and the smaller cases, to show how it can build up to a bigger instance. And we can view this as a sub-problem of these smaller instances of people, and then combine the smaller solutions to form a solution for the original problem instance. By seeing what happens with smaller sub problems, we can apply and combine the solution/steps into the larger picture. That is why this can be seen as a divide and conquer proof/problem.

*Just to reiterate: We can see the runtime/total work this algorithm requires is O(n) by just looking at the path of P1. We can see that every swap makes P1 closer to the end, and once P1 is at the end with the appropriate swaps applied to the P1 nodes and all other necessary nodes, then we know that our algorithm is complete.