CS4102 - ALGORITHMS - SPRING 2022

DUE FRIDAY, FEBRUARY 18, 2022 AT 11:30 PM VIA GRADESCOPE **A3BASIC(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: none

Sources: Cormen, et al, Introduction to Algorithms.

PROBLEM 1 *QuickSort*

1. Briefly describe a scenario when Quicksort runs in $O(n \log n)$ time.

   Solution: *A scenario where Quicksort runs in $O(n \log n)$ time is when the selected partition is the exact median. This enables Quicksort to split the set/list of values in half and recurse on each half independently. If this splitting is done perfectly, or to put it another way, if the median value is selected as the partition value at each level, then we can have $\log n$ levels of partitions each of which require n total time per level (to partition) yielding us with an $O(n \log n)$ algorithm.*

   But the scenario described above is only one scenario, another scenario is where the partitions divide the set into fractional subsets of n (1/10 and 9/10). Pretty much any partition that doesn't divide the set into (n-1 and 1) or constant subsets will be $O(n \log n)$

2. For Quicksort to be a stable sort when sorting a list with non-unique values, the Partition algorithm it uses would have to have a certain property (or would have to behave a certain way). In a sentence or two, explain what would have to be true of Partition for it to result in a stable Quicksort. (Note: we're not asking you to analyze or explain a particular *implementation* of Partition, but to describe a general behavior or property.)

   Solution: *Stability is the idea that elements with equivalent sorting-key values will be in the same order before and after sorting. The best way to go about this is to identify whether or not an element is equal to our partition element and then check whether or not that equivalent element appeared before or after our partition element, determining where the equivalent element ends up (which side of the partition element) depending on its initial position relative to the partition element. This would ensure that Quicksort is stable.*

PROBLEM 2 *QuickSelect and Median of Medians*

1. When we add the median-of-medians method to QuickSelect in order to find a good pivot for QuickSelect, name the algorithm we use to find the median value in the list of medians from the 5-element "chunks".

   Solution: *We use QuickSelect on the list of medians to return our median value.*

2. Let's say we used the median-of-medians method to find a "pretty good" pivot and used that value for the Partition we use for Quicksort. (We're *n*ot using that value with QuickSelect to find the real median, but instead we'll just use this "pretty good" value for the pivot value before we call QuickSort recursively.) Fill in the blanks in this recurrence to show the time-complexity Quicksort if the size of the two sub-lists on either side of the pivot were as uneven as possible in this situation:

$$T(n) \approx T(??) + T(??) + \Theta(n)$$

Replace each "??" with some fraction of n, such as $0.5n$ or $0.95n$ etc.

Solution: *The median-of-medians method described in class gives us a partition element that is guaranteed to be in the middle 40% of values. In the worst case, this would result in a partition of 4/10 and 6/10:*

$$T(n) \approx T(\frac{4n}{10}) + T(\frac{6n}{10}) + \Theta(n)$$

PROBLEM 3 *Other Divide and Conquer Problems*

1. What trade-off did the arithmetic "trick" of both Karatsuba's algorithm allow us to make, compared with the initial divide and conquer solutions for the problem that we first discussed? Why did making that change reduce the overall run-time of the algorithm?

   Solution: *Say we have to multiply the values ab by cd where each letter represents a unique digit. For example, the number 10 would be $a = 1, b = 0$. For a larger number such as 1020, $a = 10, b = 20$. The traditional method would recursively perform a series of multiplications and additions based on quadratic multiplication to get the original $ab * cd$ term. The problem is that with this method, per level we have 4 times more sub-problems each of which are only half the size of the original problem. Additionally, we require 5 operations per subproblem to recombine (two multiplications and three additions).*

   The trick with Karatsuba's algorithm is that Karatsuba exchanges the number of subproblems for a greater number of steps in the recombination step. Instead of operating on a larger number of subproblems, we instead expend more energy on the recombining these subproblems. The reason this is favorable is because the number of supbroblems scales exponentially while the recombination step scales multiplicatively (with the number of subproblems). Therefore by lowering the number of subproblems we can get a direct decrease in the recursive step as well as an indirect decrease in the recombination step. Lets see how the number of subproblems at each level stack up between the naive approach and karatsuba's approach:

Level	Naive	Karatsuba
1	1	1
2	4	3
3	16	9
4	64	27
5	256	81

2. Would it be feasible (without reducing the time complexity) to implement the closest pair of points algorithm from class by handling the points in the runway first, and then recursively solving the left and right sub-problems? If your answer is "no", briefly explain the reason why.

   Solution: *It would not be feasible to initially handle the points in the runway before recursing. The reason is because the basis of this approach is that we are able to bound our search space, preventing us from having to tank the egregious n^2 penalty of completing pair-wise checking. Recursing in halves is a useful tool because per recursive step, we are able to further reduce the width of our runway. By reducing the width of our runway, we can do pair-wise comparisons over a significantly smaller portion of points, the key advantage to this recursive approach. If we were to simply operate on the runway first, we would perform a n^2 operation and find our solution eliminating the need to recurse at all as well as any advantage to our recursive approach.*

3. In the closest pair of points algorithm, when processing points in the runway, which of the following are true?

   (a) It's possible that the pair of points we're seeking could be in the runway and both points could be on the same side of the midpoint.

   Solution: *This is false. While the pair of points we're seeking could be on the runway, if both points of the pair are on either side runway then we've already computed the distance between that pair and accounted for it during the recursive steps.*

   (b) The algorithm will have a worse time-complexity if we needed to check 50 points above a given point instead of 7 (as we did in class).

   Solution: *The time-complexity will not be worse because we also check a constant number of points in the runway in the closest pair of points algorithm. A constant number of checks will not worsen our time-complexity.*

   (c) The algorithm will have a worse time-complexity if we needed to check $\sqrt{n}$ points above a given point instead of 7 (as we did in class).

   Solution: *The time-complexity will be worse if we have to check a number of points that scales with n. So this scenario would demonstrate a worse time-complexity.*

PROBLEM 4 *Lower Bounds Proof for Comparison Sorts*

In class, we saw a lower-bounds proof that general comparison sorts are always $\Omega(n \log n)$. Answer the following questions about the decision tree proof that we did.

1. What did the internal nodes in the decision tree represent?

   Solution: *The internal nodes in the decision tree represented the comparisons required to determine which half of the remaining permutation space the current permutation belongs to.*

2. What did leaf nodes of the decision tree represent?

   Solution: *Each leaf node represents a unique permutation of the initial set of values.*

PROBLEM 5 *Gradescope Submission*

Submit a version of this `.tex` file to Gradescope with your solutions added. You should only submit your `.pdf` and `.tex` files.