Homework 2

Due September 22 (11:59 p.m.)

Instructions

- **Answers should be written in the boxes by modifying the provided Latex template or some other method answers are clearly marked and legible. Submit via Gradescope.**
- **Honors questions are optional. They will not count towards your grade in the course. However you are encouraged to submit your solutions to these problems to receive feedback on your attempts.**
- **You must enter the names of your collaborators or other sources (including use of LLMs) as a response to Question 0. Do NOT leave this blank; if you worked on the homework entirely on your own, please write "None" here. Even though collaborations in groups of up to 3 people are encouraged, you are required to write your own solution.**
- **For the late submission policy, see the website.**
- Acknowledgment - Thank you to Amir, Daji, Oded, Peter and Rotem for help in setting the problem set.

1-0 List all your collaborators and sources: ($-\infty$ points if left blank)

The tutor and the textbook, GeeksforGeeks, Simon another kid at tutoring

Problem 2-1 – Pigeonhole Principle (16 Points)

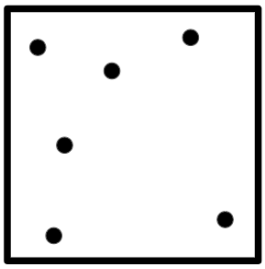

Figure 1: Sparse set of points in a unit square

1. (2 point) State the Pigeon Hole Principle in your own words and give some intuition as to why it should be true.

Solution: Three people are going on a trip, but have only booked 2 rooms. There will be at least one room where there are multiple people sleeping. If the number of available slots/holes is fewer than the number of pigeons, there will be more than one pigeon in at least one slot/hole. In the hotel example, if there are three people, you would need at least three rooms for each person to have their own room. But only two rooms were booked...thus, the Pigeon Hole Principle.

Homework 2

2. (7 point) Here is a more formal statement of the Pigeon Hole Principle: Let A and B be finite sets such that $|A| > |B|$. Then there does not exist any one to one function $f : A \to B$. Prove this using induction.

Solution: Base case. Consider $B = \emptyset$.

$\to |A| > |B|$ Proving for all A where |A| > |B|

If B is the empty set, there is not only no one-to-one function, but there isn't any function at all for A to map to b/c there are no B values.

Hypothesis. Let X = some arbitrary set.
Consider B = X
Assume for all sets A where |A| > |B|, there doesn't exist a one-to-one fxn $f : A \to B$.

Inductive Step. Consider $|A| > |B|$, then one element y is added to B. One element z must also be added to A to ensure that $|A| > |B|$ (from the hypothesis above).

To make a valid function $f : A \to B$, z needs to be mapped to an output in B, in this case y, so that $f(z) = y$.

The rest of the elements haven't been changed/matched one-to-one as per the hypothesis. This follows that for A without z and B without y (even though z and y are mapped), you fall back to the hypothesis that you can't form a one-on-one fxn. Even with the addition of z and y, a one-on-one function isn't formed because the prev. elements in A and B aren't correctly mapped.

3. (7 points) Consider a unit square (side length is 1 unit). A set of points S is called *sparse* if any two points in S are at least $\frac{1}{4}$ distance away from each other. Prove that there exists a constant C such that no sparse set of points within the square can contain more than C many points. (One approach to proving this is to try to apply the Pigeon Hole Principle)

Solution: When doing this proof, I saw that it was helpful to recognize that I'm not proving an exact number of points within the square, but rather that there just *exists* some constant C.

Proof: Initially, dividing the square into a grid of cells with a diameter less than $\frac{1}{4}$. At first thought, this appeared to successfully limit the number of dots per cell to just one. However, you would just be able to draw infinitely smaller grids, fitting in more points. Rather, circles within the main square accurately fit the situation. If you have circles that are allowed to bleed out of the square of radius $\frac{1}{8}$ with the point being the center of the circle, you're able to prove that the number of points in the square reach a constant C based on the number of circle-centers in the square.

Problem 2-2 – Recurrence Practice (24 points)

(6 points each) Solve the following recurrences by finding a function $f(n)$ such that $T(n) = O(f(n))$ using the method specified. Show your work. You only need to show an upper bound for each (i.e., big-O), but you will only get partial credit if you do not find the best possible asymptotic.

1. Use the recurrence tree method to solve $T(n) = 4T(n/2) + n$ with $T(1) = T(0) = 3$.

Solution: Recurrence Tree:
first level: $T(n)$; n work done
second level: breaks off into four branches of $T(\frac{n}{2})$; $2n$ work being done
third level: each of those four branches breaks off into four additional branches each of $T(\frac{n}{4})$; $16n$ work being done
and so on...with each previous branch breaking into four new ones with half the work being done on each branch.
Summation for work done:

$\sum_{i=1}^{\log_2 n} 2^{i-1}n$

Have to use geometric progression to simplify sum: $a_1 * \frac{1-r^k}{1-r}$ where a is the first term, k is the nmber of items and r is the growth factor:

$n * \frac{1-2^{log_2 n}}{1-2} = n * (\frac{1-n}{-1}) \rightarrow n(n-1) = n^2 - n = \mathcal{O}(n^2)$

2. Use the substitution method to verify your function found in part 1.

Solution: Need to prove that given a substitution, that by induction, our function is correct. We want to show that for all powers of 2 $n \geq 1$, let's say $T(n) \leq Cn^2 - n, C = 4$, which needs to imply our $\mathcal{O}(n^2)$ function.

Base case. When $n = 1$, the function outputs 3; $T(1) = 3 \leq 4 * 1^2 - 1 = 3$.
Inductive Hypothesis. Need to assume that claim is true for values after base case and for all smaller powers of two.

$$\begin{aligned} T(n) &= 4T(\frac{n}{2}) + n \\ &\leq 4(C(\frac{n}{2})^2 - (\frac{n}{2})) + n \\ &= 4(Cn^2/4 - \frac{n}{2}) + n \\ &= Cn^2 - 2n + n \\ &= Cn^2 - n \end{aligned}$$

Claim is proven for n, being $\mathcal{O}(n^2)$. Also noted that I set $C = 4$ because the base case needs to be ≥ 3.

Homework 2

3. solve $T(n) = T(n/3) + \log_3 n$ with $T(0) = T(1) = T(2) = 2$.

Solution: At the start of the recurrence tree, you have the first level: $T(n)$. The work begin done here is the initial $log_3 n$ because the levels will be divided into thirds.
Second level: There will be one new node of size $\frac{n}{3}$, where the work being done is $log_3 n - 1$.
The next level will have one additional node of size $\frac{n}{3^2}$, with a total work on that level of $log_3 n - 2$.
Like this, the tree will go on for $log_3 n$ levels until it bottoms out where size = 1.

$$\begin{aligned} T(n) &= \sum_{i=0}^{log_3 n - 1} (log_3 n - i) + T(1) \\ &= (log_3 n) log_3 n - \sum_{i=0}^{log_3 n - 1} i + T(1) \\ &= (log_3 n) log_3 n - \frac{(l-1)L}{2} + T(1) \\ T(n) &= \mathcal{O}((logn)^2) \quad \text{only one child per node, so single } logn \text{ chain.} \end{aligned}$$

Each level work begin done decreases pretty linearly, summing work done gives log in quadratic total.

4. Use the substitution method to verify your function found in part 3.

Solution: Doing powers of three for n since it's cleaner for the recursive calls.
Base case. $L = 0, n = 3^0 = 1$. With $T(1) = 2$, the RHS is $2 + \frac{0*(0+1)}{2} = 2$, so the base case holds for L = 0.
Inductive Step. Assuming formula holds for L - 1, assuming $L \geq 1$, so $T(3^(L-1) = 2 + \frac{(L-1)L}{2}$.

$$\begin{aligned} T(3^L) &= T(\frac{3^L}{3}) + log_3(3^L) \\ &= T(3^{L-1}) + L \\ &= (2 + \frac{(L-1)L}{2}) + L \quad \text{b/c inductive hypothesis} \\ &= 2 + \frac{(L-1)L + 2L}{2} = 2 + \frac{L^2 + L}{2} = 2 + \frac{L(L+1)}{2} \end{aligned}$$

Matches claimed formula, holding for L, affirming $\mathcal{O}((logn)^2)$

Problem 2-3 – Fast Exponentiation (20 Points+Honors Problem)

Naively, in order to compute 2025^n given input n, we need to perform $n - 1$ many multiplications.

Homework 2

1. (8 points) For this problem, assume the input n is a power of 2. That is, assume $n = 2^k$ for some positive integer k. Give an algorithm that computes 2025^n using only $O(k)$ many multiplications.

Solution: Substitute $n = 2^k$, for $2025^n = 2025^{2^k}$. If im assuming n is a power of 2, we can: set k to 0 set m to n while m > 1: divide m by 2 and increment k by 1 after the while loop breaks, set an x variable to 2025 and then run a for loop of sorts from 1 to k where $x = x * x$, output x.

2. (12 points) Justify the correctness and runtime complexity of your algorithm.

Solution: For proof correctness, run a loop invariant that tracks i iterations, where $x = 2025^{2^i}$. It'll work for i = 0 but afterward, squaring will result in $(2025^{2^i})^2 = 2025^{2^{i+1}}$. After k iterations, it'll result in 2025^{2^k} which is the same as 2025^n.
For runtime complexity, the number of mults. will equal k itself, because it's one k per squaring operation. Since we're dividing k (or m in the algo above) by 2 eveyr time, $k = log_2 n$, which results in $\mathcal{O}(logn)$.

Solution: *cont.*

3. (**Honors Problem**, 0 points, *) Extend your result to the general case. That is, given an input any natural number n, give an algorithm that computes 2025^n using only $O(\log_2 n)$ multiplications.

Solution:

Problem 2-4 – Sorting Stability (20 Points)

Recall that a sorting algorithm is said to be *stable* if it does not change the order of identical elements in the input array.

(More formally, if A is the input and A' is the sorted output, then for any two elements $A[i] = A[j]$ in the input, if i', j' are the new locations of elements $A[i]$ and $A[j]$ in the sorted output array, then $i < j$ if and only if $i' < j'$.)

(a) (10 points) Show that Merge Sort is stable. (A convincing explanation is enough, there is no need for a formal proof.)

Solution: Merge Sort is stable because it ensures that order is still preserved upon the merge for values that are the same, using $\geq$ or $\leq$ rather than just $< \mid >$. At the point of merge, the assumption is that we have two sorted sequences, left/right. If done properly, when merging, if the next unused/unmerged item in the left is equal to that in the right, the merge will choose the element in the left subarray first consistently. This ensures that order for equal elements is preserved at each lev of recursion from the original array. An explicit tie break that basically prefers the left-element over the right is the key, basically greater than/equal to and less than/equal to.

(b) (10 points) Show that Quicksort (using the last element as the pivot) is *not* stable, by giving an example of an input array A containing two identical elements, such that running Quicksort on A changes the order of these elements.

Solution: Array A = [2, 2, 1]
If we do the partition step that quicksort uses, using the last elem as the pivot (element 1, index 2):
i = low - 1 = -1
j = low...high - 1
we would compare j = 0 and j = 1 to element 1 (the pivot) respectively, concluding that both are greater
after the loop i = -1, we would swap A[i+1] with the pivot, basically the first and last element, resulting in A = [1, 2, 2]. The initial order of the equal-2s has been changed, proving that quicksort with the last elem as the pivot is unstable.

Problem 2-5 – Weighted Median (20 Points)

Suppose we have n distinct positive numbers $x_1, x_2, \ldots, x_n$ such that $\sum_i x_i = 1$. The goal of this question is to find the weighted median. We define the weighted median as the element x_k such that:

$$\sum_{x_i < x_k} x_i < \frac{1}{2} \qquad \text{and} \qquad \sum_{x_i > x_k} x_i \leq \frac{1}{2}.$$

That is, the weighted median lies somewhere in the array such that everything smaller sums to less than 1/2 and everything larger sum to at most 1/2. With divide and conquer, we can solve this problem in time $\Theta(n)$.

Homework 2

1. (15 points) Use divide-and-conquer to create an algorithm that finds the weighted median of an input list $x_1, x_2, \ldots, x_n$ satisfying $\sum_{x_i} x_i = 1$. (**Hint:** calculate the actual median first, which can be done in time $\Theta(n)$! Then, consider the total sum of elements less than the median and the total sum greater than the median. Try to use these to make some clever partitions for your divide step based on the actual median.)

   Solution: We can get the actual median of the list in linear time. We could then partition the list into three diff groups:
   L = values < m
   E = values = m
   R = values > m
   Get weights for three groups, given t = 1/2, then $W_L \leq t \leq W_L + W_E$, means that any element in group E is a weighted median.
   if $W_L > t$ weighted median is then in L group, need to recurse on L with same target t = 1/2
   if $W_L + W_E < t$, weighted median will be in R group, have to recurse on R with updated target $t' = t - (W_L + W_E)$, because left+equal weight already been removed
   Since we're picking real median of current subarr each time, chosen side for weighted medians has size <= half, so running time of $\mathcal{O}(n)$.

2. (5 points) Justify the runtime of your algorithm. You only need to argue the upper bound (that is, the algorithm runs in $O(n)$).

   Solution: each call will do selection of real median of current subarray in linear time
   it also only takes one pass through the input list to partition and compute the weights for the partitioned groups, which is linear time
   after the partition, the algo should recurse on half the elements maximum because the actual median in the list is the pivot
   The division into subarrays and iteration thru for the real median gives a recurrence of $T(n) \leq T(\frac{n}{2}) + cn$, simplyfing to a runtime of $\mathcal{O}(n)$.

Honors Problem – Quicksort with Duplicates (0 Points)

1. (*): In class, we considered for simplicity Quicksort as running on arrays in which every element is unique. However, in reality, this will not always be the case. Consider the case where any element appears in the array at most $n/2$ times. Show that even if we partition around the median, the Quicksort recursion tree might be of depth $\Omega(n)$.

   Solution:

2. (**): Modify the partition subroutine in Quicksort to eliminate this problem, so that even if the array A has duplicate elements, if we choose the median of A as the pivot at each step, the depth of the Quicksort recursion tree will be $O(\log n)$.

Homework 2

Solution: