CS2040 Quiz 1

Problem 1 [10 Pts.]

Problem 1.a

$$T(n) = \frac{n^2}{n-1} + \frac{2n}{5} \cdot \frac{n^2}{2} + n^2 \log n \leq 2n + \frac{n^3}{5} + n^2 \log n = \mathcal{O}(n^3)$$

Problem 1.b

The inner two loops aren't accessed for $i \geq 50$. Therefore, $T(n) = T(50) + c(n - 50) = \mathcal{O}(n)$.

Additional comments

Note that this doesn't mean that *nothing* is being done when $i \geq 50$. Some of the operations that are being done at every iteration of the outer loop include: - Creating and assigning the varaible `k`. - Checking if `k < 50`. - Incrementing `i`.

Therefore, for $n \geq 50$, it performs a fixed number of operations for $0 \leq i < 50$ ("Boss" gets printed out exactly 127500 times), then performs a constant amount of work at every iteration for $i \geq 50$.

Problem 1.c

This problem has the same proof with Paranoid-Quicksort but using $\frac{2}{5}$ and $\frac{3}{5}$ instead of $\frac{1}{10}$ and $\frac{9}{10}$.

Problem 1.d

The `doWork(n)` method returns n, and runs in $\mathcal{O}(n)$ time. Hence, each call to `recursiveLoops(n)` performs cn work and makes a single recursive call to `recursiveLoops(n/2)`.

$$
\begin{aligned}
T(n) &= cn + T\left(\frac{n}{2}\right) \\
&= cn + c \cdot \frac{n}{2} + T\left(\frac{n}{4}\right) \\
&= cn + c \cdot \frac{n}{2} + c \cdot \frac{n}{4} + T\left(\frac{n}{8}\right) \\
&\leq cn + c \cdot \frac{n}{2} + c \cdot \frac{n}{4} + c \cdot \frac{n}{8} + \cdots \\
&= c\left(1 + \frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \cdots\right) \\
&= c \cdot \frac{1}{1 - \frac{1}{2}} \\
&= 2c \\
&= \mathcal{O}(n)
\end{aligned}
$$

Additional comments

For a more detailed proof, refer to the appendix.

Problem 2 [10 Pts.]

Sorting Algorithm	Number of Swaps
Insertion Sort	$\mathcal{O}(n^2)$
Selection Sort	$\mathcal{O}(n)$
Quick Sort	$\mathcal{O}(n \log n)$ or $\mathcal{O}(n^2)$
Heap Sort	$\mathcal{O}(n \log n)$

Looking at the number of swaps in the table above, the best algorithm is Selection Sort.

Explanation

This problem asks for the sort that uses the least number of swaps/movements. For simplicity, let's assume the cats are stored in an array we're sorting the cats in ascending order of size.

Insertion Sort

At each iteration, Insertion Sort will move the next unchecked element to its position in the growing sorted array. The swapping occurs within this moving process, where how many swaps depends on where the element ends up in the

growing sorted array. The further up in front an element ends up, the more swaps occur.

A case that maximizes the number of swaps is a descending array. For example, consider the array `[5, 4, 3, 2, 1]`. - Iteration 1: `[5, 4, 3, 2, 1]` (0 swaps) - Iteration 2: `[4, 5, 3, 2, 1]` (1 swap) - Iteration 3: `[3, 4, 5, 2, 1]` (2 swaps) - Iteration 4: `[2, 3, 4, 5, 1]` (3 swaps) - Iteration 5: `[1, 2, 3, 4, 5]` (4 swaps)

Therefore, for an array of n elements, the worst case for the number of swaps is bounded by $\mathcal{O}(n^2)$.

Selection Sort

At each iteration, Selection Sort checks all the unsorted elements to find the minimum element, then swaps the minimum element to its correct sorted position. Notice that regardless of the initial ordering of the array, Selection Sort will perform at most 1 swap per iteration.

Therefore, for an array of n elements, the worst case for the numer of swaps is bounded by $\mathcal{O}(n)$.

Quick Sort

The swaps for Quick Sort occur during the partitioning process. Additionally, the number of swaps depends on the choice of the partitioning algorithm (Hoare's or Lomuto's Partition).

Hoare's Partition

For Hoare's Partition, we have two pointers, l and r, pointing to the start and end of the array to be partitioned. As l and r approach each other, a swap occurs when $A[l] > p$ and $A[r] < p$, where p is the pivot. (E.g. $[4, \underbrace{1, 0}_{< \text{pivot}}, \underline{6}, 5, 3, \underline{2}, \underbrace{8, 7}_{> \text{pivot}}] \rightarrow [4, \underbrace{1, 0, \underline{2}}_{< \text{pivot}}, 5, 3, \underbrace{\underline{6}, 8, 7}_{> \text{pivot}}]$)

To maximize the number of swaps for an array of k elements (including the pivot), the array can be in the following form:

$$[p, g_1, g_2, \dots, g_{\frac{k-1}{2}}, s_1, s_2, \dots, s_{\frac{k-1}{2}}]$$

Where p is the pivot, $g_i > p$ and $s_i < p$ for all $i \in [1, \frac{k-1}{2}]$. This will result in a total of $\frac{k-1}{2}$ swaps during the partition, then 1 more swap when placing the pivot in the correct position. Once that's done, the array is partitioned into two halves, where QuickSort will recurse on both of them.

Hence, if $S(n)$ represents the maximum number of swaps required by Quick Sort that uses Hoare's Partition for an array of size n,

$$S(n) \approx \frac{n}{2} + 2S\left(\frac{n}{2}\right)$$

Solving the recurrence relation, $S(n) = \mathcal{O}(n \log n)$.

An example of an array that exhibits this behaviour is `[3, 4, 6, 5, 1, 0, 2]`, where the first element is always chosen as the pivot.

Lomuto's Partition

For Lomuto's Partition, a swap occurs when a value smaller than the pivot is found while already having encountered a value larger than the pivot (E.g. $[4, \underbrace{1, 2, 3}_{< \text{pivot}}, \underbrace{5, 6}_{> \text{pivot}}, 0, 7] \rightarrow [4, \underbrace{1, 2, 3, 0}_{< \text{pivot}}, \underbrace{6, 5}_{> \text{pivot}}, 7]$).

To maximize the number of swaps for an array of k elements (inlcuding the pivot), the array can be in the following form:

$$[p, g, s_1, s_2, \dots, s_{k-2}]$$

Where p is the pivot, $g > p$ and $s_i < p$ for all $i \in [1, k-2]$. This will result in a total of $(k-2)$ swaps during the partition, then 1 more swap when placing the pivot in the correct position. Once that's done, the array is partitioned into one block of size 1, and another block of size $(k-2)$. You might notice that this is similar to the worst case of Quick Sort, and that's exactly what we're aiming for.

Hence, if $S(n)$ represents the maximum number of swaps required by Quick Sort that uses Lomuto's Partition for an array of size n,

$$S(n) = (n-1) + S(n-2)$$

Solving the recurrence relation, $S(n) = \mathcal{O}(n^2)$.

An example of an array that exhibits this behaviour is `[7, 8, 6, 4, 1, 2, 3, 5]`, where the first element is always chosen as the pivot.

Heap Sort

Performing Heap Sort on an array of n elements consists of two steps: - Construct the Heap - Remove the maximum element n times.

We'll need to find the number of swaps performed during each step.

Constructing the Heap

We know that Heap Construction has a time complexity of $\mathcal{O}(n)$ for an array of size n. Therefore, intuitively, the number of swaps is upper bounded by $\mathcal{O}(n)$ *(because if it were any larger, Heap Construction would be slower than $\mathcal{O}(n)$ time).*

We will not be discussing why there will be $\mathcal{O}(n)$ swaps during Heap Construction in any more detail *(because it will not matter).*

Extract Max

Removing the maximum element of a Heap of size n consists of two steps: - Replace the root with the last element in the Heap.

1 swap (at most) will be performed.

- Perform the *Sink* operation on the new root.

  1 swap is performed each time a node goes down one level. In the worst case, the maximum number of swaps performed is equal to the height of the Heap, which we know is $\mathcal{O}(\log n)$.

Therefore, in a Heap of size n, the "Extract Max" operation will perform at most $\mathcal{O}(\log n)$ swaps.

Since we perform the "Extract Max" operation n times, the total number of swaps performed during Heap Sort is at most $\mathcal{O}(n \log n)$.

An example of an array that maximizes the number of swaps is `[1, 4, 6, 2, 5, 3, 7]`.

Problem 3 [15 Pts.]

$\mathcal{O}(n \log n)$ **[10 Pts.]**

Declare an array T of size n and copy the contents from all k the linked lists into T. Sort T using Merge Sort. Copy the contents of T into another Linked List and return it.

Step	Time complexity
Copy the contents from all linked lists into T	$\mathcal{O}(n)$
Sort T using Merge Sort	$\mathcal{O}(n \log n)$
Copy the contents of T into another Linked List	$\mathcal{O}(n)$

Overall time complexity: $\mathcal{O}(n \log n)$

Additional Comments

This method is the simplest, and most intuitive solution. As such, any solution that is slower than, or as slow as this solution will receive a maximum of 5 out of 10 points for your Algorithm.

$\mathcal{O}(nk)$ **[10 Pts.]**

Declare a new Linked List R to store the output.

Compare the first element of each of the k linked lists. Find the linked list where the first element is the smallest, remove the first element from that list and insert it into R.

Repeat this process until all n elements have been inserted into R. Finally, return R.

Step	Time complexity
1. Find the list where the first element is the smallest.	$\mathcal{O}(k)$
2. Remove the first element from that list and append it to R.	$\mathcal{O}(1)$
3. Repeat steps 1 and 2 n times.	$\mathcal{O}(nk)$

Overall time complexity: $\mathcal{O}(n \log n)$

Additional Comments

A significant number of students implemented this algorithm, but incorrectly determined the time complexity to be $\mathcal{O}(n)$, possibly because they incorrectly stated that finding the minimum element among k Linked Lists took $\mathcal{O}(1)$ time, or they correctly reached $\mathcal{O}(nk)$ but concluded that it is equivalent to $\mathcal{O}(n)$ since k is a constant.

Unless specified, **do not** assume that the variables that appear in a question are constants.

$\mathcal{O}(n \log k)$ **[15 Pts.]**

Label the Linked Lists as $L_1, L_2, \ldots, L_k$. Declare a new Linked List R to store the output.

Declare a Minimum Heap that contains integer pairs. This Heap will compare only the first element of integer pairs (i.e. the minimum integer pair (u, v) is the pair with the smallest value of u).

For each Linked List L_i, insert an integer pair, (First element of L_i, i), into the Min Heap.

Extract the minimum integer pair, (a, i), from the Min Heap. Insert a into R, and remove the first element from L_i. If L_i is not empty, insert (First element of L_i, i) into the Min Heap.

Repeat the above process until all n elements have been inserted into R. Finally, return R.

Step	Time complexity
1. Initialize the Min Heap.	$\mathcal{O}(k \log k)$
2. Extract the minimum pair (a, i) from the Min Heap.	$\mathcal{O}(\log k)$
3. Insert a into R and remove the first element of L_i.	$\mathcal{O}(1)$
4. Insert (First element of L_i, i) into the Min Heap.	$\mathcal{O}(\log k)$
5. Repeat steps 2 to 4 n times.	$\mathcal{O}(n \log k)$

Overall time complexity: $\mathcal{O}(n \log k)$

Additional Comments

The initialization of the Min Heap can be done in $\mathcal{O}(k)$ instead of $\mathcal{O}(k \log k)$, but it will not affect the overall time complexity.

This solution has the same idea as the 10 point $\mathcal{O}(nk)$ solution, but uses a Min Heap to speed up the process of "finding the list with the smallest first element" from $\mathcal{O}(k)$ to $\mathcal{O}(\log k)$.

This model solution included a fair amount of fine details. While we cannot guarantee which parts may be omitted without being at risk of losing marks, do remember that the less vague your solution is, the less likely you will be deducted.

Alternative $\mathcal{O}(n \log k)$ solution

Label the lists $L_1, L_2, \ldots, L_k$.

Perform the Merge algorithm pairwise on the lists i.e. merge L_1 and L_2 into $L_{1,2}$, merge L_3 and L_4 into $L_{3,4}$, and so on. After that, repeat this process i.e. merge $L_{1,2}$ and $L_{3,4}$ into $L_{1,2,3,4}$, merge $L_{5,6}$ and $L_{7,8}$ into $L_{5,6,7,8}$, and so on.

Continue to repeat this process until only one list remains. Return that list.

Every time we merge the lists pairwise, the total number of lists will be cut in half. Therefore, the process will be repeated $\log_2 k$ times.

Step	Time complexity
1. Merge the lists pairwise using the Merge algorithm.	$\mathcal{O}(n)$
2. Repeat step 1 $\log_2 k$ times.	$\mathcal{O}(n \log k)$

Overall time complexity: $\mathcal{O}(n \log k)$

Additional Comments

It is also possible to do this in a Top-down recursive aproach using Divide and Conquer.