CS218, Fall 2025, Assignment #1 **Due: 07:00 AM, Mon 10/14, 2025**
Name: Vedant Deepak Borkute **Student ID: 862552981**

---

1 A Complex Complexity Problem (24pts)

1. $$\sqrt{n} = \omega\big(\log^2 n\big)$$

   Because polynomial functions such as $\sqrt{n}$ grow asymptotically faster than any polylogarithmic function like $(\log n)^2$.

2. $$3^n = o(n!)$$

   If we take logarithm to base n on both sides of the equation, we get $n \log 3$ on the left, and $\log(n) + \log(n-1) + \cdots + \log(1)$ on the right. The right-hand expression is of order $n \log n$ while the left hand expression is order n, therefore as $n \to \infty$, the right hand side is greater.

3. $$3^n = \omega(2^n)$$

   Because as $n \to \infty$, $\lim_{n\to\infty} \frac{3^n}{2^n}$ approaches ∞.

4. $$\log \sqrt{n} = \Theta(\log^2 n)$$

   Because $\lim_{n\to\infty} \frac{\log \sqrt{n}}{\log n^2}$ simplifies to $\lim_{n\to\infty} \frac{(1/2)\log n}{2 \log n}$ which reduces to a constant $1/4$.

2 Solve Recurrences (18pts)

1. The recurrence is not in a form required by the master theorem. In order to apply the master theorem, we need to get it in an appropriate form.

   Consider $k = \log n$, this would make the recurrence in terms of k look like:

   $$T(2^k) = T(2^{k/2}) + 1$$

   To arrive at a form required by the master theorem, we also need to consider $T(2^k) = F(k)$.

   $$F(k) = F(k/2) + 1$$

   Now, $f(n) = 1$ which is a constant, and $n^{\log_b a} = k^{\log_1 2}$ simplifying to 1, which is constant too. Therefore, using the master theorem we can say

   $$F(k) = \Theta(k^{\log_1 2} \log k)$$

   which simplifies to $F(k) = \Theta(\log k)$.

   Reverse-substituting k, we get

   $$T(2^k) = \Theta(\log k)$$

   Again, reverse-substituting for n, we get

   $$T(n) = \Theta(\log(\log n))$$

2.

$$T(n) = T(n-1) + n^2$$

If we keep on expanding the right term,

$$T(n) = T(n-2) + (n-1)^2 + n^2$$

If we continue till the end, we get

$$T(n) = T(n-n) + 1^2 + 2^2 + \cdots + n^2$$

with the n^2 term repeated n times. Assuming $T(0) = k$, a constant term, we get

$$T(n) = k + \frac{n(n+1)(2n+1)}{6}$$

,

$$T(n) = k + (2n^3 + 3n^2 + n)/6$$

From this, we can conclude $T(n) = \Theta(n^3)$.

3.

$$T(n) = T(n/2) + n \log n$$

As the relation is in an appropriate format for the master theorem, and since $n^{\log_2 1}$ is a constant (1) while $f(n) = n \log n$, we can say that its polynomially less than f(n). We want to check two conditions for applying master theorem in this case.

1. $f(n) = \Omega(n^{\log_2(1+\epsilon)})$ for $\epsilon > 0$ with $f(n) = n \log n$, since $n^{\log_2(1+\epsilon)}$ will evaluate to some constant and f(n) will be asymptotically lower bounded by that.

2. Considering a = 1, b = 2 from master theorem's form,

$$f(n/2) = \frac{n}{2} \log(n/2) = \frac{n}{2}(\log n - \log 2) = \frac{1}{2} n \log n - \frac{n}{2} \log 2$$

therefore

$$f(n/2) \leq 0.5(n \log n)$$

Therefore, by master theorem

$$T(n) = \Theta(n \log n)$$

3 Special Sorting (42pts)

1. We can use a binary decision tree method to analyse the number of comparisons needed to get to a sorted array from all possible permutations of that sorted array, as discussed in the class. At each step, we have a condition comparing an element of the array with others to decide the branches for each set of permutations. To get to the sorted array which is a leaf from the root of this tree which contains all solutions, we would need minimum as manuy comparisons as the depth of the tree which is $log(n!)$, which simplifies to $\Theta(nlogn)$.

2. An existing algorithm that can sort an array of size n using $O(n \log n)$ swaps (not necessarily adjacent) would be heapsort.

   Consider that we have an unsorted array in the start. We initially represent it as a complete binary tree, filling it level by level sequentially. An assumption made while doing this conversion is that internal nodes for the tree are found up until $n//2$ (in python) and the rest are leaf nodes.

   Now, we need a function to heapify this tree such that it satisfies max heap property - parent node has to be $\geq$ child node.

For this, we will start from the $n//2$ element in the array as current root, (this will be an internal node in the binary tree), and compare whether its left child(found at $2i+1$) or right child(found at $2i+2$) are larger than it. If yes, we point the 'largest' to that node, and swap the parent and that node.

Then, we recursively call heapify on the new child node's subtree, ensuring that the max heap property is satisfied in the whole binary tree overall.

We need to do this for all the internal nodes from $n//2$ up until the first element, so that all those subtrees starting from those nodes are valid max heaps. Therefore the time complexity for building this heap would be $(n/2)\log n$.

Now, in this max heap, we end up with the largest element as the root, and at max this would have required at most $\log n$ comparisons (since in the worst case, the largest element would have started as a leaf node).

Next, we will swap this largest element with the current last element of the array, our aim here being to end with a sorted (ascending) array. Now, we run the same Heapify function on a reduced heap (not considering the last element), starting with the first element, since the later part of the array remained unaffected with our swap (only the first element changed). We keep on getting the largest element at the root, moving it to the end and running heapify on the reduced heap.

In the end, we would run this $n-1$ times (since the rest of them would be sorted). Therefore the time complexity involved is $O((n-1)\log n)$.

Therefore the total complexity is $O(n\log n)$ and we would end up with the sorted array in the end.

```
HeapSort(A):
    n = len(A)
    for i = (n/2) - 1 to 0:
        Heapify(A, n, i)
    for i = n - 1 to 1:
        swap A[0], A[i]
        Heapify(A, i, 0)

Heapify(A, n, i):
    largest = i
    left = 2i + 1
    right = 2i + 2
    if left < n and A[left] > A[largest]:
        largest = left
    if right < n and A[right] > A[largest]:
        largest = right
    if largest ≠ i:
        swap A[i], A[largest]
        Heapify(A, n, largest)
```

3. Consider the following algorithm, which is a naive solution to sorting an array (bubble sort)

   For each element in the array, if it is greater than the next element, we keep swapping it with the next element until no more swaps are possible (i.e., the swap condition, which is a comparison, fails). In the worst case, when the array is sorted in descending order, $n-1$ adjacent swaps are needed for the first element to move it to its correct position, $n-2$ for the second element, and so on.

   Therefore, in the worst case we would need

   $$(n-1)+(n-2)+\cdots+1$$

   adjacent swaps, i.e. $\frac{n(n-1)}{2}$, which leads to an $O(n^2)$ time complexity in terms of adjacent swaps.

```
for i = 1 to len[A]:
    for j = len[A] to i + 1:
        if A[j] < A[j-1] then
            swap A[j], A[j-1]
```

4. Consider an example array $[6, 4, 3, 1]$. An inversion is a pair of elements in the array where the smaller element appears at a larger index. We can consider No of inversions as the degree of unsortedness of an array. In the above array, there are currently six inversions: $(6, 4), (6, 3), (6, 1), (4, 3), (4, 1), (3, 1)$.

   Therefore, in an array of N elements, in the worst case i.e when the array is sorted in descending order, we would need to deal with

   $$(N-1) + (N-2) + (N-3) + \cdots + 1 = \frac{N(N-1)}{2}$$

   inversions. In the end, we want to achieve an array with zero inversions i.e. a sorted array.

   And with each adjacent swap, we can reduce the number of inversions by exactly one. Therefore, to clear all $\frac{N(N-1)}{2}$ inversions, we would need at least that many adjacent swaps. Hence, the worst-case time complexity of adjacent swaps needed to sort any array of N elements is $O(N^2)$.

5. We need to find the exact number of adjacent swaps needed to sort any given array. In order to know this, we need to find the number of inversions in the array.

   An algorithm with a runtime of $O(nlogn)$, which can help us count these inversions is merge sort. As anyway, merge sort maintains sorted subarrays of an array in its memory in the merging step, we could use that infromation to get the total number of inversions in the original array.

```
def merge_sort(arr):
    if len(arr) ≤ 1:
        return arr
    mid = len(arr) // 2
    left = merge_sort(arr[:mid])
    right = merge_sort(arr[mid:])
    return merge(left, right)


def merge(left, right):
    result = []
    i = j = 0
    while i < len(left) and j < len(right):
        if left[i] ≤ right[j]:
            result.append(left[i])
            i += 1
        else:
            result.append(right[j])
            j += 1
    result.extend(left[i:])
    result.extend(right[j:])
    return result
```

   We can keep track of inversions in the merging step. In the merging step, if a right subarray element was chosen before the current left subarray elements, that meant it was smaller than all those and therefore we had an inversion of size len(left uncompared subarray) for this element in the original subarray. Therefore we can add a step in the merge function that updated the count of inversions in this case. We return the inversion count alongwith the original resultant subarray from the merge function.

In the merge sort algorithm itself, we make correponding changes and add up inversion counts from the left and right subarrays and the merging step.

The final algorithm will look like:

```
def merge_sort(arr):
    if len(arr) ≤ 1:
        return arr, 0
    mid = len(arr) // 2
    left, inv_left = merge_sort(arr[:mid])
    right, inv_right = merge_sort(arr[mid:])
    merged, inv_split = merge(left, right)
    total = inv_left + inv_right + inv_split
    return merged, total


def merge(left, right):
    result = []
    i = j = inv_count = 0
    while i < len(left) and j < len(right):
        if left[i] ≤ right[j]:
            result.append(left[i])
            i += 1
        else:
            result.append(right[j])
            inv_count += len(left) - i
            j += 1
    result.extend(left[i:])
    result.extend(right[j:])
    return result, inv_count
```

This algorithm would allow us to get the no of inversions to take care of in $O(nlogn)$ time, while sorting using adjacent swaps in any array.

4 Being Unique

The Unique in Range property means that every subarray of a given array needs to have at least one unique element. (UIR = Unique in Range)

A naive approach would be to maintain a dictionary of counts for every possible subarray using two for loops and check whether all of them satisfy the condition. This would have a time complexity of $O(n^2)$.

As a start, we can check if the whole array has atleast one unique element or not. If it doesn't we can say that the array itself (which is a valid subarray) does not satisfy the UIR property.

Next, we want to get indices of unique elements in the array. Because, any subarrays that conatin these elements would already be satisfying UIR. Therefore we want to check subarrays between these elements , whether they satisfy the requirement.

For each unique element index from the start, we will consider subarrays from the start to that index (if any known unique element falls in between , then we start from its next element up until the next unique element). After the last unique element, we also want to check the remaining partition after that element for this property.

```
def unique_in_range(arr):
  if len(arr) < 1:
     return True
```

```
freq =
for x in arr:
  freq[x] = freq.get(x, 0) + 1

unique_elem_indices = [i for i, x in enumerate(arr) if freq[x] == 1]

if not unique_elem_indices:
  return False

prev = 0
for idx in unique_elem_indices:
  if not unique_in_range(arr[prev:idx]):
    return False
  prev = idx + 1

if not unique_in_range(arr[prev:]):
  return False

return True
```

In each call of this function, we do O(n) work to count element frequencies and identify unique elements. As we keep finding unique elements, the size of the array passed to the recursion decreases. Each unique element is only processed once initially. Each non unique element is only processed in the partitions that contain it, and multiple unique elements within a partition cause the partitions to shrink considerably. Therefore, each element appears in only a few partitions on average, and the total work for all recursive calls is **O(n log n)** amortized, since each element is involved in roughly log n partitions on average. But worst case, there may be only one unique element per partition and that too at the end, so our partitions would not decrease in size by much. In this case we would have worst case time complexity of $O(n^2)$.

References

[1] The Master Theorem used from the slides.

[2] https://www.geeksforgeeks.org/dsa/heap-sort/

[3] https://www.tutorialspoint.com/data_structures_algorithms/bubble_sort_algorithm.htm

[4] Slides for merge sort.

[5] https://stackoverflow.com/questions/78091917/google-interview-question-check-if-all-subarrays-of-an-array-have-at-least-one

[6] https://www.geeksforgeeks.org/dsa/number-swaps-sort-adjacent-swapping-allowed