CS 430 – Fall 2024
Introduction To Algorithms
Homework #2

1a) We can prove correctness by induction. For the base case let the size of the array equal 2. The algorithm will check if the two elements are sorted and if they are not, it exchanges them, which will make them sorted. So we can conclude that STOOGE-SORT sorts correctly for n = 2.

Assume Stooge-Sort correctly sorts an input array that is less than or equal to n.
Consider a subarray of size (j - i + 1), which is greater than 2. The algorithm proceeds as follows:

- The algorithm recursively calls STOOGE-SORT(A, i, j - k), where k = ⌊(j - i + 1) / 3⌋ This means the first two-thirds of the subarray (from i to j - k) is sorted by this recursive call.
- Then the algorithm recursively calls STOOGE-SORT(A, i + k, j), which sorts the last two-thirds of the subarray (from i + k to j).
- Finally, the algorithm calls STOOGE-SORT(A, i, j - k) again to ensure that any disorder introduced by sorting the last two-thirds is corrected in the first two-thirds.

By using recursion to repeatedly sort overlapping two-thirds of the subarray and combining the results, STOOGE-SORT ensures that the entire array A[1 … n] is sorted. Thus, STOOGE-SORT(A, 1, n) correctly sorts the input array.

1b) The Recurrence relation is as follows:

$$T(n) = 3T\left(\frac{2}{3}n\right) + \Theta(1)$$

Using Master Theorem:

$$T(n) = 1 + 3T\left(\frac{2}{3}n\right)$$

$$T(n) = \Theta(n^{\log_{\frac{3}{2}} 3})$$

$$T(n) = \Theta(n^{2.7})$$

1c) Stooge-Sort is the worst of all the algorithms and the professors probably don't deserve tenure.

InsertionSort: $\Theta(n^2)$
MergeSort: Θ(n log n)
Heapsort: Θ(n log n)
Quicksort: $\Theta(n^2)$
Stooge-Sort: $\Theta(n^{2.7})$

2a) After Iteration 1: $i = 1, j = 11$, array: A{6, 19, 9, 5, 12, 8, 7, 4, 11, 2, 13, 21}
After Iteration 2: $i = 9, j = 10$, array: A{6, 2, 9, 5, 12, 8, 7, 4, 11, 19, 13, 21}
In Iteration 3: Since $i \geq j$, the loop terminates, and the algorithm returns $j = 9$, which is the index where this partitioning happened.
After the end of the loop, the variables have the following values: $x = 13, j = 9$ and $i = 10$.

2b) When we start, $k =1$, $i = p$ and $p \leq j \leq r$. When Hoare-Partition is running, $p \leq i \leq j < r$ will always hold. Due to the above condition, i and j won't access any element of A outside the subarray A[p … r].

2c) Hoare-Partition terminates when $i \geq j$, so from the idea mentioned in question 2b, we can see that the returned value of j has to be $p \leq j < r$.

2d) After the kth iteration the elements A[p … i] $\leq x$ and the elements A[j … r] $\geq x$. After the while loop termination, which occurs when $i \geq j$, all the elements A[p … j] will be less than or equal to A[j + 1 … r].

3) Function rearrangeKeys(A, n):

```
low = 0
mid = 0
high = n - 1
while mid <= high:
    if A[mid] == "red":
        swap(A[low], A[mid])
        low = low + 1
        mid = mid + 1
    else if A[mid] == "white":
        mid = mid + 1
    else:
        swap(A[mid], A[high])
        high = high - 1
```

low: This pointer will track the boundary for red keys.
mid: This pointer will traverse the array.
high: This pointer will track the boundary for blue keys.

Elements before low will be "red".
Elements between low and mid will be "white".
Elements after high will be "blue".
Elements between mid and high are yet to be sorted.

4a) We know that Heapify(A, i) requires the two subtrees of node i to already be heaps. The recursive version below, makes the two subtrees of node i heaps by calling itself recursively, and then invokes Heapify at node i.

```
Function Recursive-Build-Heap(A, i):
    if i > heap-size[A]:
        return
    left = 2 * i
    right = (2 * i) + 1
    Recursive-Build-Heap(A, left)
    Recursive-Build-Heap(A, right)
    Heapify(A, i)
```

4b) We know that Heapify takes time O(log n) and there are two recursive calls to Recursive-Build-Heap, the running time can be expressed using the following equation:

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

4c) Solving the Recurrence:
We can solve this using the Master Theorem, since it is of the form, T(n) = aT(n/b) + O(n^d), where a = 2, b = 2 and d = 0.

$$T(n) = O(n^{\log_b a}) = O(n^{\log_2 2})$$

Thus, we can conclude that the worst-case running time of the Recursive-Build-Heap procedure is O(n), just like the iterative version.

5) We can sort the array in O(n log k) time using heaps with the following implementation:

- We first create a min-heap with first k + 1 elements. This ensures that the smallest element within the first k + 1 elements will be at the root of the heap.
- Then, we remove the smallest element from the min-heap(at the root) and put it at the correct position in the sorted output.
- Next, insert another element from the unsorted array into the min-heap to maintain the size of k + 1, now, we have the second smallest element in this. Extract it from the min-heap and continue this until no more elements are in the unsorted array.
- We finally use a simple heap sort for the remaining elements from the heap and place them in the correct positions.

Time Complexity:

Each insertion and extraction operation on the heap takes O(log k) time and there are n elements for which we perform these insertions and extractions from the heap
Thus, the total time complexity is O(n log k)

6) Minimum Number of Elements occurs when the heap is filled completely up to level h - 1 and has only one node at level h. So the minimum number of elements is 2^h.
Maximum Number of Elements occurs when the heap is completely filled, including all nodes at every level up to h. So the maximum number of elements is $2^{h+1} - 1$.

7a) Multiplying Two n × n Matrices by the Naïve Algorithm:
For each element in the resulting n × n matrix, we perform n multiplications and n - 1 additions. This results in a total of $O(n^3)$ operations.
The modern computer is 10^6 times faster. Therefore, it can perform 10^6 times more operations in the same amount of time.
If $T_1 = O(n^3)$ and $T_2 = O(m^3)$ for the modern computer, then $m^3 = 10^6 \cdot 6^3$

So $m = \sqrt[3]{10^6 \cdot 6^3} = 600$
The modern computer can handle matrices of size approximately 600 × 600.

7b) Heap Sort on an Array of n Integers:
The time complexity of Heap Sort is O(n log n).
If $T_1 = O(n \log n)$ and $T_2 = O(m \log m)$ for the modern computer, then $m \log m = 10^6 \cdot 32 \log 32$
So $m \log m = 10^6 \cdot 32 \cdot 5 = 10^6 \cdot 160$
$m \approx 10^4$
The modern computer can handle arrays of size approximately 10^4 integers.

7c) Selection Sort on an Array of n Integers:
The time complexity of Selection Sort is $O(n^2)$.
If $T_1 = O(n^2)$ and $T_2 = O(m^2)$ for the modern computer, then $m^2 = 10^6 \cdot 32^2$

So $m = \sqrt{10^6 \cdot 32^2} = 32000$
The modern computer can handle arrays of size approximately 32,000 integers.

7d) Linear Search on an Array of n Integers:
The time complexity of linear search is O(n) since we have to check each element in the array sequentially.
If $T_1 = O(n)$ and$T_2 = O(m)$ for the modern computer, then $m = 10^6 \cdot 200$
So m = 200,000,000
The modern computer can handle arrays of size approximately 200 million integers.

8a) Worst-Case Scenario:
In the worst case, the array is in reverse order. For each element inserted into the sorted portion of the array, it will be compared with all the other elements that have already been sorted.
For the first element, no comparisons are needed.
For the second element, it is compared with 1 element.

For the third element, it is compared with 2 elements.
For the fourth element, it is compared with 3 elements.
So, the total number of comparisons in the worst case is 1 + 2 + 3 = 6.

8b) Best-Case Scenario:
In the best case, the array is already sorted.
For the first element, no comparisons are needed.
For the second element, it is compared with 1 element.
For the third element, it is compared with 1 element (since it is already greater than the first two elements).
For the fourth element, it is also compared with only 1 element.
So, the total number of comparisons in the best case is 1 + 1 + 1 = 3.

8c) Average-Case Scenario:
In the average case, we assume that all permutations of the array are equally likely.
To compute the average number of comparisons, we need to consider the total number of comparisons across all permutations and then average them.
For each element, the number of comparisons depends on how many elements in the sorted portion of the array are greater than the current element. Here's a breakdown for each element:
For the first element, no comparisons are needed.
For the second element, on average, it will be compared with 1/2 element.
For the third element, on average, it will be compared with 2/2 = 1 element.
For the fourth element, on average, it will be compared with 3/2 1.5 elements.
Total Average Comparisons = $\frac{n(n-1)}{4} = \frac{4\times 3}{4} = 3$
Thus, the average number of comparisons for the mentioned comparison tree is 3.