Advanced Algorithms

I can provide a detailed report discussing the proposed solutions for each task and their efficiency.

Task-1:

Given an array of characters, give an efficient algorithm for removing the duplicates, if any.

One efficient solution for removing duplicates from an array of characters is to use a hash table or a set to keep track of the unique characters while iterating through the array. For each character in the array, we can check if it is present in the hash table. If it is not present, we can add it to the hash table and the result array.

If it is present, we can skip it. This algorithm has a time complexity of O(n), where n is the size of the input array. The space complexity of this algorithm is also O(n) since we need to store all the unique characters in the result array.

Task-2:

Given two arrays of unordered numbers, check whether both arrays have the same set of numbers.

One efficient solution for this task is to use a hash table or a set to count the occurrences of numbers in each array. Then, we can compare the two sets to check if they contain the same elements. This algorithm has a time complexity of O(n), where n is the total number of elements in both arrays. The space complexity of this algorithm is also O(n) since we need to store all the unique numbers in the hash tables.

Task-3:

Design an ADT for a two-color, double-stack ADT that consists of two stacks - one "red" and one "blue" - and has as its operations color-coded versions of the regular stack ADT operations. For example, this ADT should allow for both a red push operation and a blue push operation. Give an efficient implementation of this ADT using a single array whose capacity is set at some value N that is assumed to always be larger than the sizes of the red and blue stacks combined.

One efficient implementation for the two-color, double-stack ADT is to use a single array and two pointers to keep track of the top of each stack. Each stack's elements can be color-coded to distinguish them. The push operation for each stack can be implemented by incrementing the corresponding pointer and adding the element at that index. The pop operation can be implemented by decrementing the pointer and returning the element at that index. This implementation has a time complexity of O(1) for push and pop operations.

The space complexity of this implementation is

O(N) since we need to allocate an array of size

N to store the elements.

Task-4: Develop a program that simulates the scheduling process of CPU jobs. CPU jobs have priority values from 1 to 10 (highest), and a duration time, which is a value that indicates how many time slices are needed to complete the job. For each time slice, the scheduler should choose the job according to the following policies:

- First Come First Served.
- Highest priority.
- Shortest Remaining Time First.

```
One efficient implementation for this task is to use a
priority queue to store the jobs based on their priority or
remaining time. For the First Come First Served policy, we
can use a queue to store the jobs in the order of their
arrival. The scheduler selects the job at the front of the
queue to execute. For the Highest Priority policy, we can
use a priority queue to store the jobs based on their
priority value. The scheduler selects the job with the
highest priority to execute. For the Shortest Remaining Time
First policy, we can use a priority queue to store the jobs
based on their remaining time. The scheduler selects the job
with the shortest remaining time to execute. These
algorithms have different time complexities, depending on
the implementation details and the number of jobs.
```

Task 5 :

Converting a fully parenthesized, arithmetic expression to a binary expression tree

To convert a fully parenthesized, arithmetic expression to a binary expression tree, we can use a stack data structure to keep track of the operators and operands in the expression. We iterate over each character in the expression and push operands onto the stack. When we encounter an operator, we create a new binary tree with the operator as the root and the top two elements of the stack as its left and right children. We then push the new binary tree onto the stack. After iterating over the entire expression, the top of the stack contains the final binary expression tree.

This algorithm has a time complexity of O(n) and a space complexity of O(n), where n is the length of the input expression.

Task 6:

a possible solution could be to implement a variation of the Levenshtein distance algorithm, which is a measure of the difference between two strings. The algorithm computes the minimum number of operations (insertion, deletion, or substitution of a character) needed to transform one string into another. To adapt this algorithm for spell checking, we can compute the Levenshtein distance between the input string and each word in the lexicon, and return the words that have a distance below a certain threshold (e.g., 2 or 3).

Additionally, we can use phonetic algorithms such as Soundex or Metaphone to check for possible phonetic misspellings. These algorithms assign a code to each word based on its pronunciation, which can be used to compare words that sound similar but are spelled differently. By applying these algorithms to both the input string and the words in the lexicon, we can identify possible matches that would be missed by a simple Levenshtein distance check.

Task 7:

a possible implementation of heapsort using a min heap could follow these steps:

1. Build a min heap from the input list of values.

2. Repeatedly extract the minimum element from the heap and append it to a sorted list, until the heap is empty.

3. Return the sorted list.

The time complexity of heapsort is O(n log n), which is the same as other efficient sorting algorithms such as quicksort and mergesort.

Task 8:

we can use Dijkstra's algorithm to compute the shortest paths between all pairs of nodes in the network. The algorithm maintains a priority queue of nodes ordered by their distance from the source node, and repeatedly extracts the closest node and updates the distances to its neighbors. By running this algorithm once for each node in the network, we can obtain the routing table for each node. Each entry in the routing table would consist of the next hop and the distance to reach the destination node.

The time complexity of Dijkstra's algorithm is O(m log n), where n is the number of nodes and m is the number of edges in the network. Since the input is given as a list of node connectivity lists, we would need to convert this into an adjacency matrix or list representation, which takes O(m) time and space .Overall, the time complexity of building the routing tables would be On * m log n).