COMP 775 - Advanced Design and Analysis of Algorithms Assignment 3

Jordan Sturtz

2-19-2023

Problem 15-2

Question A palindrome is a nonempty string over some alphabet that reads the same forward and backward. Examples of palindromes are all strings of length 1, civic, racecar, and aibohphobia (fear of palindromes). Give an efficient algorithm to find the longest palindrome that is a subsequence of a given input string. For example, given the input *character*, your algorithm should return *carac*. What is the running time of your algorithm?

Answer:

```python
def longest_palindrome_subsequence(s):

    # Base case: string has one character in it
    if len(s) == 1:
        return s

    # Recursive case
    if s[0] == s[-1]:
        middle = longest_palindrome_subsequence(s[1:-1])
        return s[0] + middle + s[-1]
    else:
        left = longest_palindrome_subsequence(s[1:])
        right = longest_palindrome_subsequence(s[:-1])

        val = left if len(left) > len(right) else right
        return val
```

Algorithm 1: Longest Palindrome Subsequence without DP

```python
def longest_palindrome_subsequence_dp(s):

    def recursive_helper(s, memo):

        # Don't recompute something we've memoized!
        if s in memo:
            return memo[s]

        # Base case: string has one character in it
        if len(s) == 1:
            return s

        # Recursive case
        if s[0] == s[-1]:
            middle = recursive_helper(s[1:-1], memo)
            val = s[0] + middle + s[-1]
            memo[s] = val
            return val
        else:
            left = recursive_helper(s[1:], memo)
            right = recursive_helper(s[:-1], memo)

            val = left if len(left) > len(right) else right
            memo[s] = val
            return val

    return recursive_helper(s, {})
```

Algorithm 2: Longest Palindrome Subsequence with Memoization

```python
def longest_palindrome_subsequence_bottom_up(s):

    # the longest palindrome for every c in s is c itself
    memo = {c: c for c in s}
    memo[""] = ""

    # we slide over s with window to compute subproblems in bottom-up fashion
    for window in range(2, len(s) + 1):
        for i in range(len(s) - window + 1):
            subproblem = s[i:i+window]

            if subproblem[0] == subproblem[-1]:
                palindrome = subproblem[0] + memo[subproblem[1:-1]] + subproblem[-1]
            else:
                left = memo[subproblem[1:]]
                right = memo[subproblem[:-1]]
                palindrome = left if len(left) > len(right) else right

            memo[subproblem] = palindrome

    return memo[s]
```

Algorithm 3: Longest Palindrome Subsequence Bottom Up DP

From the bottom-up implementation in Algorithm 3, we can see that the algorithmic complexity is $O(n^2)$ where n is the length of s, since the algorithm involves a nested for loop over s.

Problem 15-4

Question Consider the problem of neatly printing a paragraph with a monospaced font (all characters having the same width) on a printer. The input text is a sequence of n words of lengths l_1, l_2, $\dots l_n$ measured in characters.

We want to print this paragraph neatly on a number of lines that hold a maximum of M characters each. Our criterion of "neatness" is as follows. If a given line contains words i through j, where $i \leq j$, and we leave exactly one space between words, the number of extra space characters at the end of the line is $M - j + i - \sum_{k=i}^{j} l_k$, which must be nonnegative so that the words fit on the line. We wish to minimize the sum, over all lines except the last, of the cubes of the numbers of extra space characters at the ends of lines. Give a dynamic-programming algorithm to print a paragraph of n words neatly on a printer. Analyze the running time and space requirements of your algorithm.

Answer

The optimal substructure in this problem is simple: if the optimal amount of words that can fit on the first k lines is j where $j < n$, then we need only neatly print words from $j + 1$ to n. That would produce the optimal solution.

To measure the space complexity, we assume the goal is to store the minimum information necessary to print via indices. Thus, we will store a dictionary, R, mapping row indices to the index of the last word that should be printed on that row. To find the optimal solution for the kth line, we must find the solution for the $k - 1th$ line. So we build up the problem in a bottom-up fashion from the first line.

FIXME: Give running time and space requirements of algorithm