COMP 775 - Advanced Design and Analysis of Algorithms Assignment 3

Jordan Sturtz

2-19-2023

Problem 15-2: Longest Palindrome Subsequence

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: Print Neatly

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, $\ldots l_n$ 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 cost in this problem we seek to minimize is the sum of the cubes of the leftover whitespace in each line. Suppose that we already know the cost of printing neatly words j_{k+1} to j_n, j_{k+2} to j_n, ...and j_n to j_n. A recursive solution would require us to consider how to use this knowledge to compute the next value in this sequence, which is the cost of printing neatly from j_k to j_n.

A helpful way to understand the problem is to list all the possibilities we must consider to solve the optimal cost of printing neatly from j_k to j_n:

- j_k is on a line by itself
- j_k, j_{k+1} are on a line
- j_k, j_{k+1}, j_{k+2} are on a line
- ...and so on (until there is no longer space on that line)

If we put j_k on a line by itself, then the total cost of this choice would be $(M - j_k)^3$ plus the cost from j_{k+1} to j_n. And if we instead put j_k, j_{k+1} are on a line, then the total cost of this choice would be $(M - j_k - j_{k+1} - 1)^3$ plus the the cost of printing neatly from j_{k+2} to j_n. Since we said at the outset that we are assuming we have already computed the costs of printing neatly from j_{k+1} to j_n onward, we can look up those values in a table.

Thus, the problem requires two loops. The first loops from $k = n$ to 1 to compute the costs of printing neatly from l_n to l_1 in descending order. Inside this loop, we must then consider all the possibilities to find the one with the minimum cost, which is the second loop from $j = 1$ to $n - k$. This second loop defines a new variable, j, that represents how many words we are considering to add to a new line. The possibility which best reduces the total cost is set as the cost from k_j to j_n, and at that point we also record the position of the last word to be used to actually print the results.

The implementation is below:

```
def print_neatly(text, max_len):

    n = len(text)

    costs = {n: 0} # cost of last word guaranteed to be zero
    positions = {}

    for k in list(reversed(range(n))):

        charactersum = sum(len(text[i]) for i in range(k, n))
        whitespacesum = n - 1 - k

        if charactersum + whitespacesum < max_len:
            costs[k] = 0

        mincost=float('inf')

        for j in range(n-k):
            space_left = sum(len(text[k+i]) + i for i in range(j+1))
            if space_left > max_len:
                # more looping will only add to space_left, so quit early
                break

            candidate_cost = pow(max_len - space_left, 3) + costs[k + j + 1]
            if candidate_cost < mincost:
                mincost = candidate_cost
                positions[k] = k + j

        costs[k] = mincost
    return positions, costs
```

Algorithm 4: Print Neatly Algorithm

The inner loop loops $1, 2, 3, \ldots, n$ times, which in closed form sums to $\frac{n(n+1)}{2} = O(n^2)$ for the running time. The costs dictionary is guaranteed to hold n entries, and the positions dictionary is guaranteed to hold no more than n entries, so the space complexity is $O(n)$.

Problem 15-5: Edit Distance

Question (Page 405 - 408)

a. Given two sequences x[1..m] and y[1..n] and set of transformation-operation costs, the edit distance from x to y is the cost of the least expensive operation sequence that transforms x to y. Describe a dynamic-programming algorithm that finds the edit distance from x[1..m] to y[1..n] and prints an optimal operation sequence. Analyze the running time and space requirements of your algorithm.

Answer:

Suppose we already have k transformations in the optimal transformations from x to y. The optimal next choice cannot be known without considering how that choice affects the later transformations. So, we must consider all the possibilities. We must compute from among the acceptable possibilities the cost of that choice, which requires the cost of the next choice and so on. Eventually, this recursion stops when z[j] = y[j] for all $j = 1, 2, \ldots, n$.

```python
def edit_distance(x, y):

    operations = {
        "copy": 1,
        "twiddle": 2, # should be cheap
        "replace": 3,
        "insert": 4,
        "delete": 4,
        "kill": 0,
    }

    def aux(x, y, memo, operations):

        # memoize
        if (x, y) in memo:
            return memo[(x, y)]

        # base case: nothing else to transform
        if len(y) == 0:
            return (operations["kill"], ["kill"])

        # decide on the valid operations
        valid_operations = ["insert", "delete"]
        if len(x) == 0 and len(y) > 0:
            # There's only one thing to do, which is insert
            valid_operations = ["insert"]
        elif x[0] == y[0]:
            # copy only valid if the current leading characters match
            valid_operations.append("copy")
        else:
            # and replace is only valid otherwise
            valid_operations.append("replace")

        # twiddle can only be used if the next two characters match
        if len(x) > 1 and len(y) > 1 and x[0] == y[1] and x[1] == y[0]:
            valid_operations.append("twiddle")

        mincost = float('inf')
        ops = []
        for candidate_op in valid_operations:
            opcost = operations[candidate_op]
            if candidate_op == "copy":
                nextop = f"copy {x[0]}"
                cost_after_op, candidate_ops = aux(x[1:], y[1:], memo, operations)
            elif candidate_op == "twiddle":
                nextop = f"twiddle {x[0]}, {x[1]} to {x[1]}, {x[0]}"
                cost_after_op, candidate_ops = aux(x[2:], y[2:], memo, operations)
            elif candidate_op == "replace":
                nextop = f"replace {y[0]} with {x[0]}"
                cost_after_op, candidate_ops = aux(x[1:], y[1:], memo, operations)
            elif candidate_op == "delete":
                nextop = f"delete {y[0]}"
                cost_after_op, candidate_ops = aux(x[1:], y, memo, operations)
            elif candidate_op == "insert":
                nextop = f"insert {y[0]}"
                cost_after_op, candidate_ops = aux(x, y[1:], memo, operations)

            if opcost + cost_after_op < mincost:
                mincost = opcost + cost_after_op
                ops = [nextop] + candidate_ops

        # memoize
        memo[(x, y)] = (mincost, ops)
        return (mincost, ops)

    return aux(x, y, {}, operations)
```

Algorithm 5: Edit Distance Top-Down Memoization

The bottom up implementation is below:

```python
def edit_distance_bottomup(x, y):

    operations = {
        "copy": 1,
        "twiddle": 2, # should be cheap
        "replace": 3,
        "insert": 4,
        "delete": 4,
        "kill": 0,
    }

    # regardless of size of x, if y is empty, we just kill
    memo = {(x[i:], ""): (operations["kill"], ["kill"]) for i in list(range(len(x)+1))}

    for j in list(reversed(range(len(y)))):
        for i in list(reversed(range(len(x)+1))):

            suby = y[j:]
            subx = x[i:]

            # decide on the valid operations
            valid_operations = ["insert", "delete"]
            if len(subx) == 0 and len(suby) > 0:
                # There's only one thing to do, which is insert
                valid_operations = ["insert"]
            elif subx[0] == suby[0]:
                # copy only valid if the current leading characters match
                valid_operations.append("copy")
            else:
                # and replace is only valid otherwise
                valid_operations.append("replace")

            # twiddle can only be used if the next two characters match
            if len(subx) > 1 and len(suby) > 1 \
                and subx[0] == suby[1] and subx[1] == suby[0]:
                valid_operations.append("twiddle")

            mincost = float('inf')
            ops = []
            for candidate_op in valid_operations:
                opcost = operations[candidate_op]
                if candidate_op == "copy":
                    nextop = f"copy {subx[0]}"
                    cost_after_op, candidate_ops = memo[(subx[1:], suby[1:])]
                elif candidate_op == "twiddle":
                    nextop = f"twiddle {subx[0]}, {subx[1]} to {subx[1]}, {subx[0]}"
                    cost_after_op, candidate_ops = memo[(subx[2:], suby[2:])]
                elif candidate_op == "replace":
                    nextop = f"replace {suby[0]} with {subx[0]}"
                    cost_after_op, candidate_ops = memo[(subx[1:], suby[1:])]
                elif candidate_op == "delete":
                    nextop = f"delete {suby[0]}"
                    cost_after_op, candidate_ops = memo[(subx[1:], suby)]
                elif candidate_op == "insert":
                    nextop = f"insert {suby[0]}"
                    cost_after_op, candidate_ops = memo[(subx, suby[1:])]

                if opcost + cost_after_op < mincost:
                    mincost = opcost + cost_after_op
                    ops = [nextop] + candidate_ops
            memo[(subx, suby)] = (mincost, ops)

    return memo[(x, y)]
```

Algorithm 6: Edit Distance Bottom-Up Memoization

The running time of this implementation is $O(n \times m)$, where n is the length of x and m the length of y. The memo must hold an entry for every nested loop iteration, so its space complexity is also $O(n \times m)$.

b. Explain how to cast the problem of finding an optimal alignment as an edit distance problem using a subset of the transformation operations copy, replace, delete, insert, twiddle, and kill.

Answer: The goal is to minimize the total cost of aligning one DNA sequence to another. So for instance, to align the sequence x = GATCGGCAT to the sequence y = CAATGTGAATC. We can use our edit distance algorithm, modified in some key ways. For this, we define an insert to mean inserting only a space in the source string and we define a delete to mean only inserting a space in the target string. Since both insert and delete produce a new space, the cost of insertion/deletion should be 2, since evaluated at that point in the two sequences, there would be one character and one space.

The copy / replace can be modified to merely increment the pointers. However, if the copy operation would be performed, this indicates a match in the alignment, which would have a cost of -1. The replace, by contrast, indicates a location where the two sequences have the wrong character, so the cost of replacement would be +1. We are not permitted to "twiddle".

Modified in this way, the algorithm would minimize the cost of finding the alignment of x to y.

Problem 15-6

Question

Professor Stewart is consulting for the president of a corporation that is planning a company party. The company has a hierarchical structure; that is, the supervisor relation forms a tree rooted at the president. The personnel office has ranked each employee with a conviviality rating, which is a real number. In order to make the party fun for all attendees, the president does not want both an employee and his or her immediate supervisor to attend. Professor Stewart is given the tree that describes the structure of the corporation, using the left-child, right-sibling representation described in Section 10.4. Each node of the tree holds, in addition to the pointers, the name of an employee and that employee's conviviality ranking. Describe an algorithm to make up a guest list that maximizes the sum of the conviviality ratings of the guests. Analyze the running time of your algorithm.

Answer

To understand the optimal substructure here, consider an arbitrary node, k. We do not know in advance whether the optimal solution includes k. If the optimal solution does include k, then we cannot include any of k's direct children. So, the optimal substructure requires us to know the optimal cost of each of k's children if (a) that child were excluded and (b) if that child were not excluded. Once we have (a) and (b), we can compare the following two costs:

- The cost of including k plus the costs of k's children's costs if each of k's children were excluded
- The cost of excluding k plus the costs of k's children

Once we compare those costs, we can then compute the maximal cost and return the correct guestlist accordingly. The algorithm is below:

```python
def optimal_company_party(root):
    """
    Assumes root is a Node class, defined as such:

    class Node():

        def __init__(self, name, conviv, parent=None, leftchild=None, rightsib=None):
            self.name = name
            self.conviv = conviv
            self.parent=parent
            self.leftchild=leftchild
            self.rightsib=rightsib
    """
    # base case
    if root.leftchild is None:
        return {
            "optimal_cost": root.conviv,
            "optimal_cost_without": 0,
            "optimal_party": [root],
            "optimal_party_without": [],
        }

    # recursive cases
    cost_with_root = root.conviv
    party_with_root = [root]

    cost_without_root = 0
    party_without_root = []

    child = root.leftchild
    while child is not None:
        results = optimal_company_party(child)

        cost_with_root += results["optimal_cost_without"]
        party_with_root.extend(results["optimal_party_without"])

        cost_without_root += results["optimal_cost"]
        party_without_root.extend(results["optimal_party"])

        child = child.rightsib

    if cost_with_root > cost_without_root:
        optimal_cost = cost_with_root
        optimal_party = party_with_root
    else:
        optimal_cost = cost_without_root
        optimal_party = party_without_root

    return {
        "optimal_cost": optimal_cost,
        "optimal_party": optimal_party,
        "optimal_cost_without": cost_without_root,
        "optimal_party_without": party_without_root,
    }
```

Algorithm 7: Optimal Party Algorithm

Problem 16-2

Question

Suppose you are given a set $S = \{a_1, a_2, \ldots, a_n\}$ of tasks, where task a_i requires p_i units of processing time to complete, once it has started. You have one computer on which to run these tasks, and the computer can run only one task at a time. Let c_i be the completion time of task a_i, that is, the time at which task a_i completes processing. Your goal is to minimize the average completion time, that is, to minimize $(1/n)\sum_{i=1}^{n} c_i$. For example, suppose there are two tasks, a_1 and a_2, with $p_1 = 3$ and $p_2 = 5$, and consider the schedule in which a_2 runs first, followed by a_1. Then $c_2 = 5$, $c_1 = 8$, and the average completion time is $(5 + 8)/2 = 6.5$. If task a_1 runs first, however, then $c_1 = 3$, $c_2 = 8$, and the average completion time is $(3 + 8)/2 = 5.5$.

a. **Question** Give an algorithm that schedules the tasks so as to minimize the average completion time. Each task must run non-preemptively, that is, once task a_i starts, it must run continuously for p_i units of time. Prove that your algorithm minimizes the average completion time, and state the running time of your algorithm.

   Answer

b. **Question** Suppose now that the tasks are not all available at once. That is, each task cannot start until its release time r_i . Suppose also that we allow preemption, so that a task can be suspended and restarted at a later time. For example, a task a_i with processing time $p_i = 6$ and release time $r_i = 1$ might start running at time 1 and be preempted at time 4. It might then resume at time 10 but be preempted at time 11, and it might finally resume at time 13 and complete at time 15. Task a_i has run for a total of 6 time units, but its running time has been divided into three pieces. In this scenario, a_i's completion time is 15. Give an algorithm that schedules the tasks so as to minimize the average completion time in this new scenario. Prove that your algorithm minimizes the average completion time, and state the running time of your algorithm.

   Answer

Hacker Earth Problem

Question

Answer