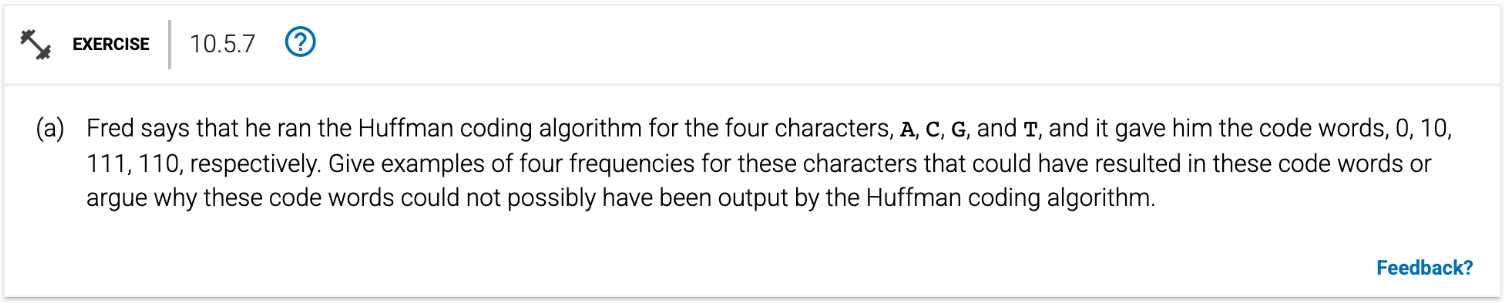
Homework 5 Solution
Komal Wavhal

Student ID- 20034443

Chapter 10 Exercises: 10.5.7, 10.5.16,10.5.27
Chapter 11 Exercises: 11.6.1, 11.6.10, 11.6.17
Chapter 12 Exercises: 12.9.5, 12.9.14, 12.9.30

Chapter 10

Exercises: 10.5.7



EXERCISE | 10.5.7

(a) Fred says that he ran the Huffman coding algorithm for the four characters, **A**, **C**, **G**, and **T**, and it gave him the code words, 0, 10, 111, 110, respectively. Give examples of four frequencies for these characters that could have resulted in these code words or argue why these code words could not possibly have been output by the Huffman coding algorithm.

Feedback?

Answer:

Fred claims that he obtained the following Huffman codes for the characters A, C, G, T :

$$A \rightarrow 0$$
$$C \rightarrow 10$$
$$G \rightarrow 111$$
$$T \rightarrow 110$$

Check for the Prefix Property

Huffman coding ensures that no codeword is a prefix of another. Observing the given codes:
- A = 0 - C = 10 - G = 111 - T = 110
None of these codewords are prefixes of another, meaning they satisfy the prefix property.

Constructing a Huffman Tree

Huffman coding builds an optimal binary tree by iteratively merging the two least frequent symbols. The given codes suggest the following tree structure:

```
   (*)
   /\
 (A)(*)
   /\
 (C)(*)
   /\
 (T )(G)
```

From this structure: - G = 111 and T = 110 share the same parent node. - C = 10 is combined at a higher level. - A = 0 is the root and must have the highest frequency.

Assigning Frequencies

To satisfy the Huffman algorithm, we need to assign frequencies such that the lower-frequency symbols are merged first. Let:

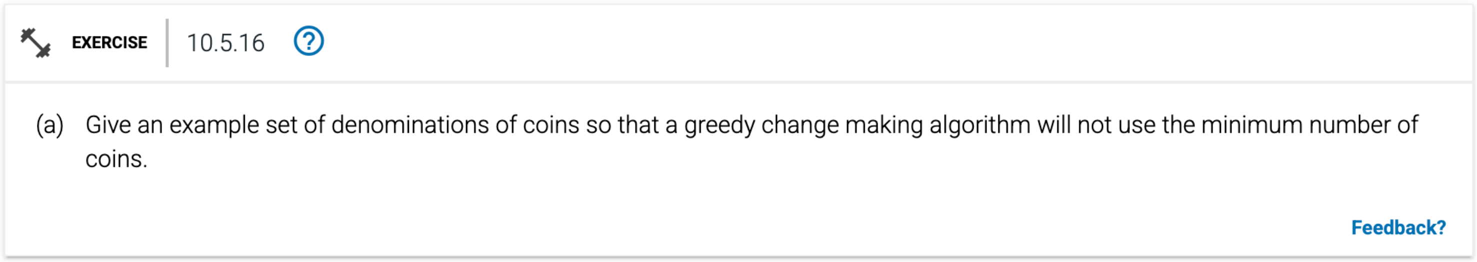
$$f_A = 0.4$$
$$f_C = 0.3$$
$$f_G = 0.15$$
$$f_T = 0.15$$

These frequencies satisfy the conditions: - $f_T = f_G$, as they were merged first. - $f_C > f_T + f_G$, ensuring C is merged next. - $f_A > f_C + f_T + f_G$, ensuring A is the root.

Since we can construct a valid Huffman tree and assign appropriate frequencies, the given codewords {0, 10, 111, 110}

could be a valid Huffman coding output.

Exercises: 10.5.16



EXERCISE 10.5.16

(a) Give an example set of denominations of coins so that a greedy change making algorithm will not use the minimum number of coins.

Feedback?

Answer:

Consider the set of coin denominations:

$$\{1, 3, 4\}$$

and the task of making change for 6 units.

Greedy Algorithm Approach

The greedy algorithm always selects the largest coin that does not exceed the remaining amount:

1. It first picks the coin with denomination 4, since $4 \leq 6$.
2. The remaining amount is $6 - 4 = 2$, so the largest coin that does not exceed 2 is 1.
3. Another 1 is needed to make up the remaining amount. Thus, the greedy algorithm uses:

$$4 + 1 + 1 = 6$$

with a total of 3 coin.

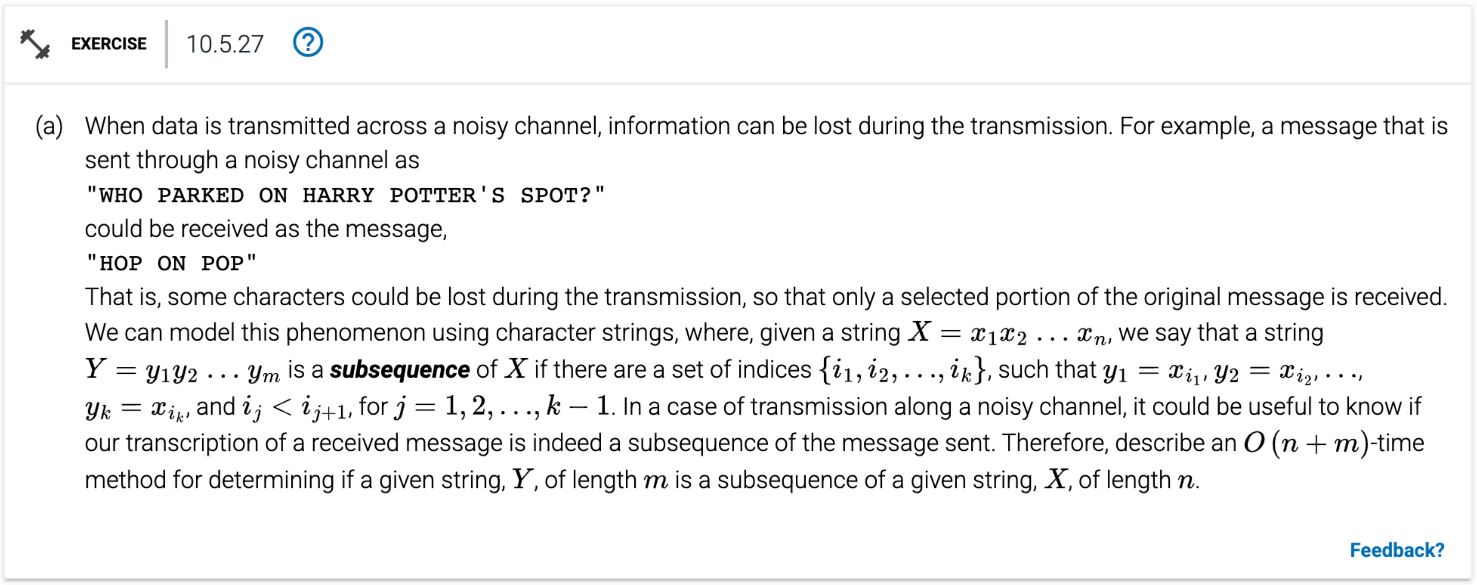
Optimal Solution A better approach is to use two coins of denomination 3:

$$3 + 3 = 6$$

which requires only 2 coins, fewer than the greedy algorithm's solution.

This example demonstrates that the greedy algorithm does not always find the optimal solution because it makes locally optimal choices at each step without considering the globally optimal outcome.

Exercises: 10.5.27



EXERCISE 10.5.27

(a) When data is transmitted across a noisy channel, information can be lost during the transmission. For example, a message that is sent through a noisy channel as

"WHO PARKED ON HARRY POTTER'S SPOT?"

could be received as the message,

"HOP ON POP"

That is, some characters could be lost during the transmission, so that only a selected portion of the original message is received. We can model this phenomenon using character strings, where, given a string $X = x_1 x_2 \ldots x_n$, we say that a string $Y = y_1 y_2 \ldots y_m$ is a ***subsequence*** of X if there are a set of indices $\{i_1, i_2, \ldots, i_k\}$, such that $y_1 = x_{i_1}$, $y_2 = x_{i_2}, \ldots$, $y_k = x_{i_k}$, and $i_j < i_{j+1}$, for $j = 1, 2, \ldots, k-1$. In a case of transmission along a noisy channel, it could be useful to know if our transcription of a received message is indeed a subsequence of the message sent. Therefore, describe an $O(n+m)$-time method for determining if a given string, Y, of length m is a subsequence of a given string, X, of length n.

Feedback?

Answer:

Given two strings, X of length n and Y of length m, we want to determine if Y is a subsequence of X. That is, we need to check whether all characters of Y appear in X in the same order, but not necessarily contiguously.

Efficient O(n + m) **Algorithm**

We use a two-pointer approach, where:

- One pointer iterates over X.
- The other pointer iterates over Y, advancing only when a match is found.
- If we successfully match all characters of Y within X, then Y is a subsequence.

Algorithm Steps

1. Initialize two indices: i = 0 for X and j = 0 for Y.
2. Traverse X from left to right:
   - If X[i] = Y[j], increment j.
   - Always increment i.

   - If j reaches m, return true (all characters of Y matched).

3. If traversal of X ends and $j \neq m$, return false.

Time Complexity Analysis

Since each pointer moves only forward and never backtracks, each character in X and Y is examined at most once. Hence, the time complexity is:

$$O(n + m)$$

Pseudocode for Checking if Y is a Subsequence of X

We present an efficient algorithm that determines whether a string Y (of length m) is a subsequence of another string X (of length n) in $O(n + m)$ time using a two-pointer approach.

Algorithm 1 Check if Y is a Subsequence of X

```
function IsSubsequence(X, Y)
    n ← length(X)
    m ← length(Y)
    i ← 1                          ▷ Pointer for X (1-based index)
    j ← 1                          ▷ Pointer for Y (1-based index)
    while i ≤ n and j ≤ m do
        if X[i] = Y[j] then
            j ← j + 1              ▷ Move pointer in Y
        end if
        i ← i + 1                  ▷ Always move pointer in X
    end while
    return (j = m + 1)             ▷ True if all characters of Y were found
end function
```

Example Example 1:

- Input: X = ”WHO PARKED ON HARRY POTTER’S SPOT?”, Y = ”HOP ON POP”
- Output: True

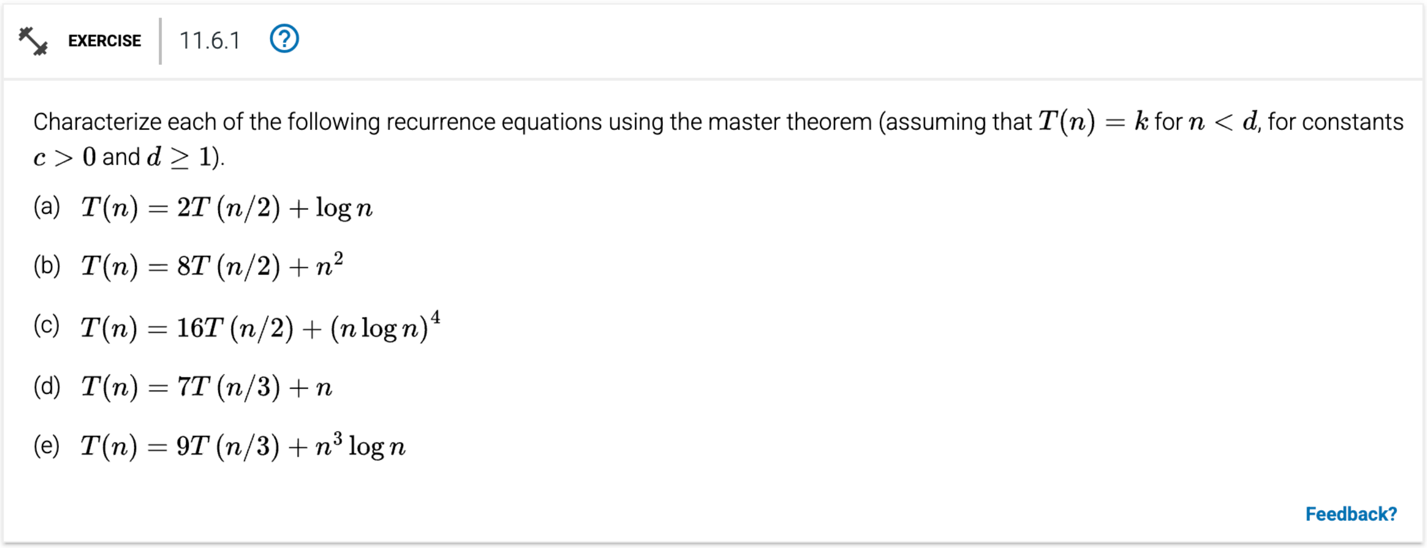
Example 2:

- Input: X = "HELLO WORLD", Y = "HOW"
- Output: False

This approach efficiently checks for subsequences in linear time O(n + m), making it ideal for large-scale string comparison tasks.

Chapter 11

Exercises: 11.6.1



EXERCISE 11.6.1

Characterize each of the following recurrence equations using the master theorem (assuming that $T(n) = k$ for $n < d$, for constants $c > 0$ and $d \geq 1$).

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

(b) $T(n) = 8T(n/2) + n^2$

(c) $T(n) = 16T(n/2) + (n \log n)^4$

(d) $T(n) = 7T(n/3) + n$

(e) $T(n) = 9T(n/3) + n^3 \log n$

Feedback?

Master Theorem The recurrence relation of the form:

$$T(n) = aT(n/b) + f(n)$$

is analyzed using the **Master Theorem**, where we compare f(n) to $O(n^c)$, where:

$$c = \log_b a$$

The solution follows these cases:

- **Case 1:** If $f(n) = O(n^{c-\epsilon})$ for some $\epsilon > 0$, then
$$T(n) = \Theta(n^c).$$
- **Case 2:** If $f(n) = \Theta(n^c)$, then
$$T(n) = \Theta(n^c \log n).$$

- **Case 3:** If $f(n) = \Omega(n^{c+\epsilon})$ for some $\epsilon > 0$ and the regularity condition $af(n/b) \leq kf(n)$ (for some $k < 1$) holds, then
$$T(n) = \Theta(f(n)).$$

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

- $a = 2$, $b = 2$, $f(n) = \log n$.
  - Compute $c = \log_2 2 = 1$.
  - $f(n) = \log n$ is smaller than $O(n^{1-\epsilon})$.
  - **Case 1 applies**, so:
$$T(n) = \Theta(n).$$

(b) $T(n) = 8T(n/2) + n^2$

- $a = 8$, $b = 2$, $f(n) = n^2$.
  - Compute $c = \log_2 8 = 3$.
  - $f(n) = n^2$ is smaller than $O(n^3)$.
  - **Case 1 applies**, so:
$$T(n) = \Theta(n^3).$$

(c) $T(n) = 16T(n/2) + (n \log n)^4$

- $a = 16$, $b = 2$, $f(n) = (n \log n)^4$.
- Compute $c = \log_2 16 = 4$.
- $f(n) = (n \log n)^4$ is asymptotically the same as $O(n^4)$, meaning $f(n) = \Theta(n^c)$.
- **Case 2 applies**, so:
$$T(n) = \Theta(n^4 \log n).$$

(d) $T(n) = 7T(n/3) + n$

- $a = 7$, $b = 3$, $f(n) = n$.
  - Compute $c = \log_3 7 \approx 1.77$.
  - $f(n) = n$ is smaller than $O(n^{1.77})$.
  - **Case 1 applies**, so:
$$T(n) = \Theta(n^{1.77}).$$

(e) $T(n) = 9T(n/3) + n^3 \log n$

- $a = 9$, $b = 3$, $f(n) = n^3 \log n$.
  - Compute $c = \log_3 9 = 2$.

- $f(n) = n^3 \log n$ is asymptotically larger than $O(n^2)$, and we check the regularity condition:

$$9f(n/3) \leq kf(n), \text{for some } k < 1.$$

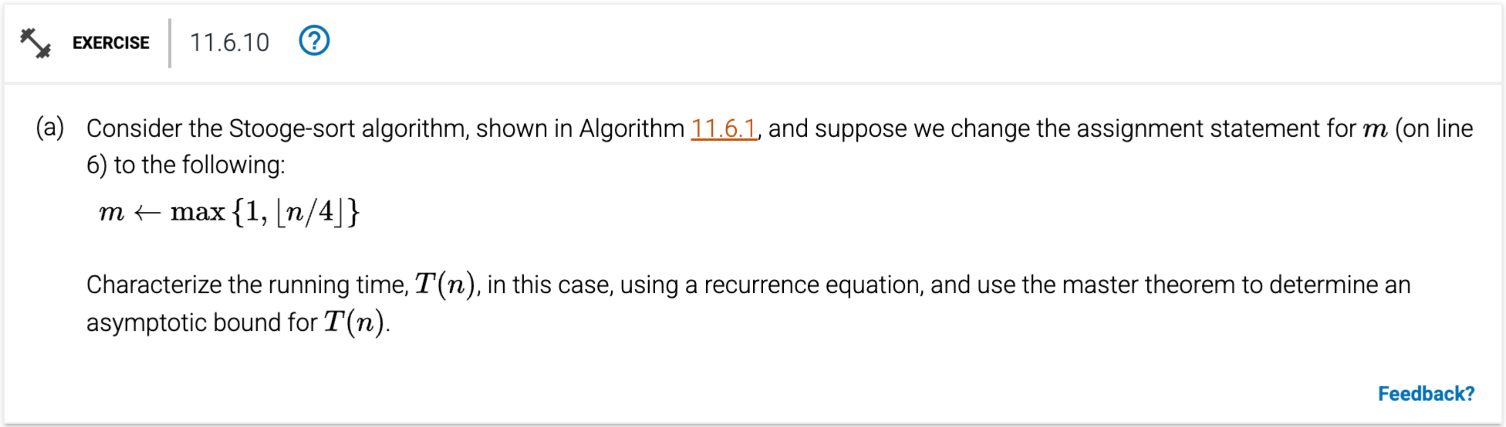
- Since $n^3 \log n$ grows faster than n^2, the regularity condition is satisfied.
- **Case 3 applies**, so:

$$T(n) = \Theta(n^3 \log n).$$

Final Answers

Recurrence	Master Theorem Case	Complexity
$T(n) = 2T(n/2) + \log n$	Case 1	$\Theta(n)$
$T(n) = 8T(n/2) + n^2$	Case 1	$\Theta(n^3)$
$T(n) = 16T(n/2) + (n \log n)^4$	Case 2	$\Theta(n^4 \log n)$ $\Theta(n^{1.77})$
$T(n) = 7T(n/3) + n$	Case 1	

Exercises: 11.6.10



EXERCISE 11.6.10

(a) Consider the Stooge-sort algorithm, shown in Algorithm 11.6.1, and suppose we change the assignment statement for m (on line 6) to the following:

$$m \leftarrow \max\{1, \lfloor n/4 \rfloor\}$$

Characterize the running time, $T(n)$, in this case, using a recurrence equation, and use the master theorem to determine an asymptotic bound for $T(n)$.

Feedback?

Answer:

The standard Stooge Sort algorithm follows this recurrence:

$$T(n) = 3T(\lfloor 2n/3 \rfloor) + O(1).$$

In the modified version, we change the assignment of m to:

$$m = \max(1, \lfloor n/4 \rfloor).$$

Thus, the recursive calls occur for input sizes $n - m$, where:

$$n - m = n - \lfloor n/4 \rfloor = \lfloor 3n/4 \rfloor.$$

Since the algorithm recursively calls itself three times on size $\lfloor 3n/4 \rfloor$, the recurrence relation is:

$$T(n) = 3T(\lfloor 3n/4 \rfloor) + O(1).$$

Applying the Master Theorem

The recurrence follows the form:

$$T(n) = aT(n/b) + f(n),$$

where: - a = 3 (number of recursive calls), - b = 4/3 (scaling factor for n), - f(n) = O(1) (constant extra work per call).

To apply the Master Theorem, we calculate:

$$c = \log_b a = \log_{4/3} 3.$$

Using logarithm properties:

$$c = \frac{\log 3}{\log(4/3)} \approx 2.41.$$

Since f(n) = O(1) = O(n^0), we compare 0 with c:

$$0 < 2.41.$$

Thus, by Case 1 of the Master Theorem:

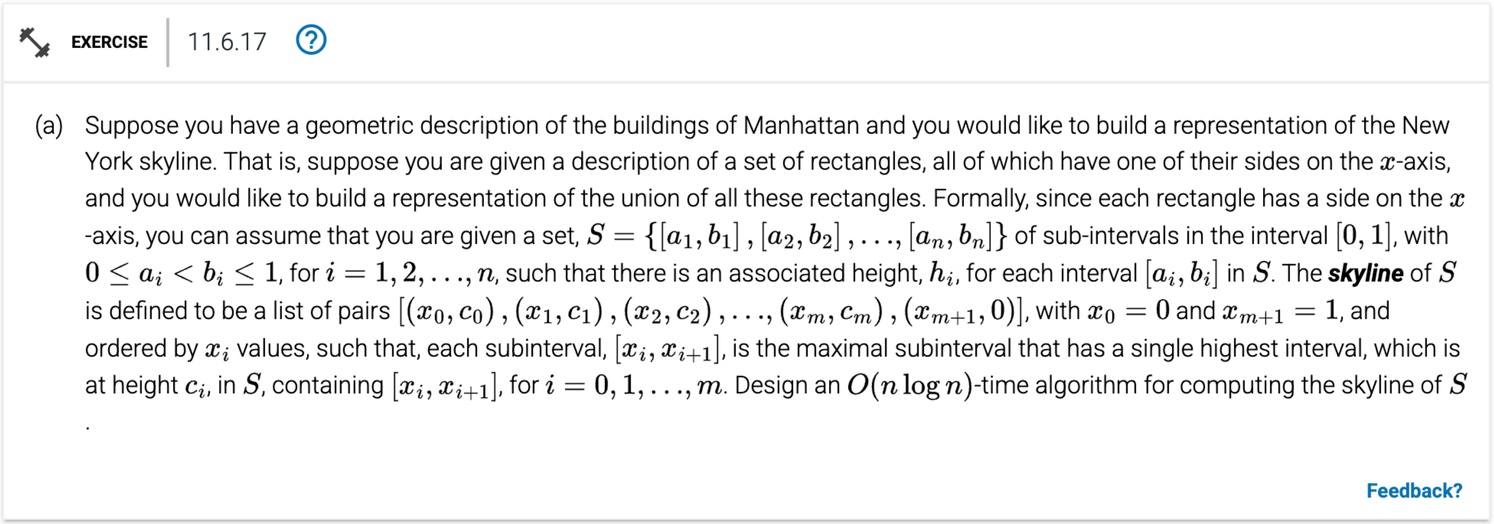
$$T(n) = \Theta(n^c) = \Theta(n^{2.41}).$$

The modified Stooge Sort has a time complexity of:

$$T(n) = \Theta(n^{2.41}).$$

This is an improvement over the original Stooge Sort, which runs in $\Theta(n^{\log_{3/2} 3}) \approx \Theta(n^{2.7095})$.

Exercises: 11.6.17



EXERCISE 11.6.17

(a) Suppose you have a geometric description of the buildings of Manhattan and you would like to build a representation of the New York skyline. That is, suppose you are given a description of a set of rectangles, all of which have one of their sides on the x-axis, and you would like to build a representation of the union of all these rectangles. Formally, since each rectangle has a side on the x-axis, you can assume that you are given a set, $S = \{[a_1, b_1], [a_2, b_2], \ldots, [a_n, b_n]\}$ of sub-intervals in the interval $[0, 1]$, with $0 \le a_i < b_i \le 1$, for $i = 1, 2, \ldots, n$, such that there is an associated height, h_i, for each interval $[a_i, b_i]$ in S. The ***skyline*** of S is defined to be a list of pairs $[(x_0, c_0), (x_1, c_1), (x_2, c_2), \ldots, (x_m, c_m), (x_{m+1}, 0)]$, with $x_0 = 0$ and $x_{m+1} = 1$, and ordered by x_i values, such that, each subinterval, $[x_i, x_{i+1}]$, is the maximal subinterval that has a single highest interval, which is at height c_i, in S, containing $[x_i, x_{i+1}]$, for $i = 0, 1, \ldots, m$. Design an $O(n \log n)$-time algorithm for computing the skyline of S.

Feedback?

Answer:

Given a set of rectangles represented as intervals with heights:

$$S = \{[a_1, b_1, h_1], [a_2, b_2, h_2], \ldots, [a_n, b_n, h_n]\}$$

where each rectangle starts at a_i, ends at b_i, and has height h_i, we aim to compute the **skyline**, which represents the outline of the union of these rectangles.

Divide and Conquer Algorithm

We solve the problem using a **Divide and Conquer** approach, similar to the merge step in *Merge Sort*.

Algorithm Steps

1. **Base Case:** If there is only one building, return its skyline directly.
2. **Divide:** Split the list of buildings into two equal halves.
3. **Conquer:** Recursively compute the skyline for each half.
4. **Merge:** Combine the two skylines into a single skyline.

Merging Two Skylines Given two skylines:

$$S_1 = [(x_1, h_1), (x_2, h_2), \ldots, (x_m, h_m)]$$

$$S_2 = [(y_1, k_1), (y_2, k_2), \ldots, (y_p, k_p)]$$

We merge them by iterating through both, keeping track of the maximum height at each point.

Algorithm 2 ComputeSkyline

```
1: function ComputeSkyline(S)
2:   if $|S| = 1$ then
3:       return Single-building skyline
4:   end if
5:   Divide S into $S_1$ and $S_2$
6:   $L \leftarrow$ ComputeSkyline($S_1$)
7:   $R \leftarrow$ ComputeSkyline($S_2$)
8:   return MergeSkyline(L, R)
9: end function
```

Algorithm 3 MergeSkyline

```
1: function MERGESKYLINE(L, R)
2:    Initialize result list, $h_1 = 0$,
$h_2 = 0$ 3: while L and R are
not empty do 4:        Pick the
smaller x-coordinate
5:        Update height $h_1$ or $h_2$
6:        Append maximum height to result
7:    end while
8:    Append remaining points
9:    return result
10: end function
```

Time Complexity Analysis

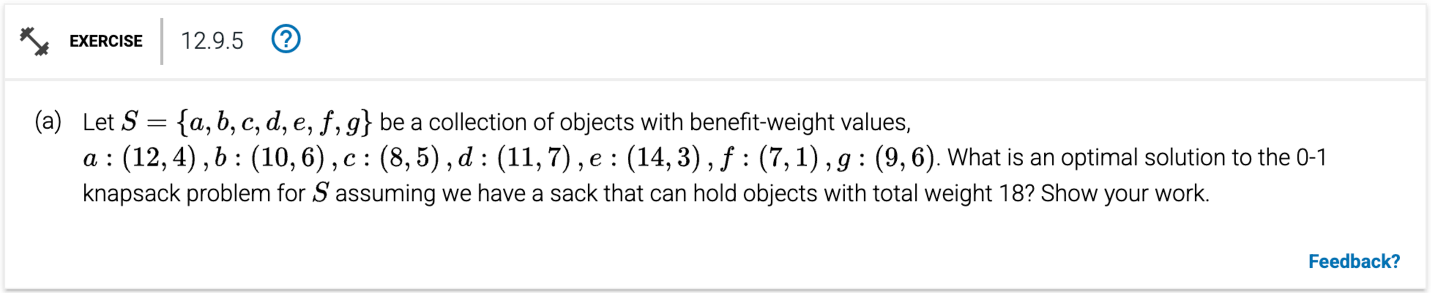
Each merge step takes O(n), and there are O(log n) levels of recursion. Thus, the total complexity is:

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

This divide and conquer approach efficiently computes the skyline in O(n log n), making it suitable for large datasets. 22

Chapter 12

Exercises: 12.9.5



EXERCISE | 12.9.5

(a) Let $S = \{a, b, c, d, e, f, g\}$ be a collection of objects with benefit-weight values, $a : (12, 4), b : (10, 6), c : (8, 5), d : (11, 7), e : (14, 3), f : (7, 1), g : (9, 6)$. What is an optimal solution to the 0-1 knapsack problem for S assuming we have a sack that can hold objects with total weight 18? Show your work.

Feedback?

Answer:

Solution Using Dynamic Programming

Define dp[i][w] as the maximum benefit using the first i items with weight capacity w. The recurrence relation is :

$$dp[i][w] = \begin{cases} dp[i-1][w] & \text{if } w_i > w \text{ (item cannot be included)} \\ \max(dp[i-1][w], dp[i-1][w-w_i] + v_i) & \text{otherwise (include or} \end{cases}$$

e x clude item)

Using dynamic programming, we find the maximum possible benefit is:

$$\mathbf{38}$$

The selected items for this optimal solution are:

$$\{e(14,3), a(12,4), d(11,7), f(7,1)\}$$

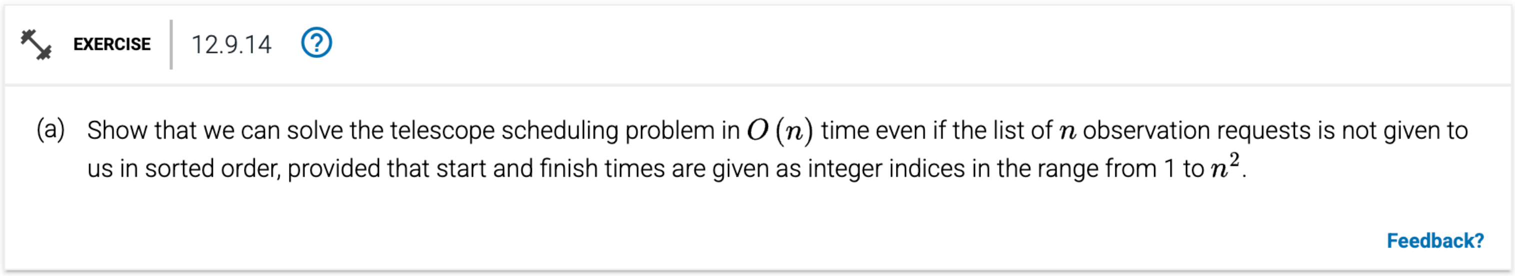
Verification

- Total weight: $3 + 4 + 7 + 1 = 18$
- Total benefit: $14 + 12 + 11 + 7 = 38$

Thus, the optimal benefit for this knapsack problem is:

$$\boxed{38}$$

Exercises: 12.9.14



EXERCISE | 12.9.14

(a) Show that we can solve the telescope scheduling problem in $O(n)$ time even if the list of n observation requests is not given to us in sorted order, provided that start and finish times are given as integer indices in the range from 1 to n^2.

Feedback?

Answer:

Sorting Using Counting Sort

The greedy interval scheduling algorithm typically requires sorting requests by their completion times, which takes O(n log n) time using comparison-based sorting. However, since $\mathbf{f}_i$ is in the range [1, n^2], we can use **to count the sorting** to achieve the sorting time O(n).

Counting Sort Approach

1. Create a frequency array for finish times of size n^2.
2. Store observation requests in buckets indexed by their finish times.
3. Iterate over the frequency array to retrieve requests in sorted order.

Since there are only n requests, iterating through the frequency array in a bucket-sort manner takes at most O(n) time.

Greedy Scheduling Algorithm

Once requests are sorted by finish times, we apply the standard **greedy interval scheduling** algorithm:

1. Initialize an empty schedule.
2. Iterate through the sorted requests:
    - If the current request's start time is at least the finish time of the last scheduled request, add it to the schedule.

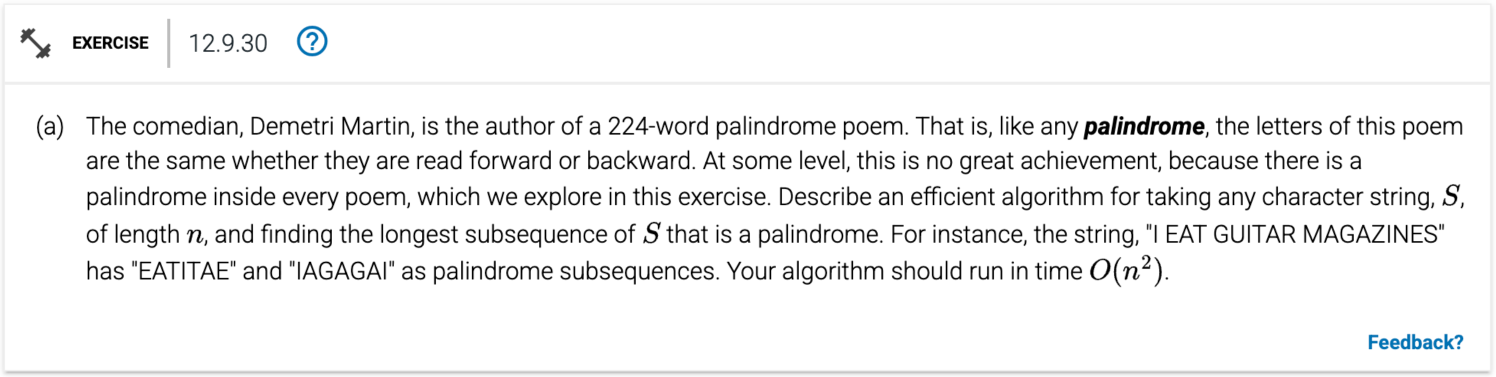
Since we only process each request once, this step runs in O(n) time.

Overall Complexity

- **Sorting**: O(n) (using counting sort)
- **Greedy Scheduling**: O(n)
- **Total Complexity**: O(n)

By leveraging counting sort, we reduce sorting complexity from O(n log n) to O(n), making it possible to solve the telescope scheduling problem in **linear time**.

Exercises: 12.9.30



EXERCISE 12.9.30

(a) The comedian, Demetri Martin, is the author of a 224-word palindrome poem. That is, like any ***palindrome***, the letters of this poem are the same whether they are read forward or backward. At some level, this is no great achievement, because there is a palindrome inside every poem, which we explore in this exercise. Describe an efficient algorithm for taking any character string, S, of length n, and finding the longest subsequence of S that is a palindrome. For instance, the string, "I EAT GUITAR MAGAZINES" has "EATITAE" and "IAGAGAI" as palindrome subsequences. Your algorithm should run in time $O(n^2)$.

Feedback?

Answer:

Recurrence Relation

- If S[i] = S[j], then: $dp[i][j] = dp[i+1][j-1] + 2$

$$dp[i][j] = \max(dp[i+1][j], dp[i][j-1])$$

- If S[i]/= S[j], then:
- Base Case: A single character is always a palindrome of length 1:

$$dp[i][i] = 1$$

Algorithm in Pseudocode

Input: String S of length n
Output: Length of longest palindromic subsequence

```
1. Initialize a 2D array dp[n][n] with 0.
2. For i = 0 to n-1:
       dp[i][i] = 1   // Base case: Single character palindromes

3. For substrings of increasing length L
       = 2 to n: For i = 0 to n-L:
           j = i + L - 1 // Compute ending
           index If S[i] == S[j]:
               dp[i][j] = dp[i+1][j-1]
           + 2 Else:
               dp[i][j] = max(dp[i+1][j], dp[i][j-1])

4. Return dp[0][n-1]
```

Time Complexity Since we fill an $O(n^2)$ table and each entry takes O(1) time to compute, the overall complexity is:

$O(n^2)$