Ribbon cutting problem by Brute force and Dynamic programming approaches

Mayar Ahmed , Mariam Mohamed ,Julie Gamal , Nour Mohey , Farida Hesham
(2023/05693), (2023/04412), (2023/00892), (2023/00968), (2023/00935)
Dr Ashraf AbdelRaof

Abstract—Ribbon cutting is a fundamental problem in combinatorial optimization. Although the problem appears straightforward, it presents significant complexity when approached algorithmically. Ribbon cutting is used to maximize the number of ribbon pieces by determining the most efficient way to divide the ribbon using the given cut lengths. This research compares two different approaches to solve the ribbon cutting problem: brute force and dynamic programming. The brute force approach exhaustively enumerates all possible ways to cut the ribbon using the given lengths, while the dynamic programming method adopts a bottom-up strategy to efficiently determine the optimal solution. These two methods are implemented and analyzed in terms of both time complexity and space complexity.

1. Introduction

The Ribbon Cutting Problem tries to maximize the number of ribbon pieces you can get from a set length, using specific approved cut sizes. To solve it, you can use methods like brute force and energetic programming. Brute force involves checking every possible way to cut the ribbon and picking the one with the most pieces. While this approach is accurate, it quickly becomes slow for larger inputs because it requires checking many options. Energetic programming, on the other hand, offers a smarter way. Instead of recomputing the same calculations repeatedly, it remembers the results of smaller problems and reuses them. This makes solving bigger or more complicated versions much faster and more efficient. Overall, energetic programming isn't just quicker; it's also better suited for real-world problems that are more complex.

2. BRUTE FORCE APPROACH

Brute-force methods, sometimes called 'generate and test' or 'exhaustive search,' are pretty straightforward problem-solving techniques. They work by checking out every possible answer, one by one. Instead of relying on shortcuts or smart guesses, they systematically go through all potential solutions to see if they fit. While these methods can be slow and time-consuming, they're actually quite useful for smaller problems, helping you get a decent idea of the solution and understanding how the problem works.

The Ribbon Cutting Problem is all about figuring out how to cut a ribbon of a fixed length, say n, into the most pieces possible. You're only allowed to cut it into lengths of a, b, or c. The catch is, you have to use up the whole ribbon — no leftover pieces. The main goal isn't to waste less or to get a specific pattern, but just to get the biggest number of valid cuts you can make from the entire ribbon.

3. Bute Force for Ribbon Cutting Problem (Iteratively)

Unlike the recursive brute-force method, which tries every possible combination using recursion and can run into issues like stack overflows or doing the same calculations over and over, the iterative brute-force approach skips recursion altogether. Instead, it uses nested loops to check all possible combos of the given cut sizes. The key is that the total length of all the cuts has to be exactly n. For each combination of cuts made with sizes a and b, the algorithm checks if the remaining length can be perfectly divided by size c. If it can, it calculates the total number of pieces (i + j + k), and from all these options, it picks the biggest one. This way, you're guaranteed to find the best possible number of pieces by exploring every feasible combination within the set constraints.

The Algorithm

Main function

- Prompt user for total ribbon length → n
- Prompt user for number of cut sizes → k
- Read k cut sizes into list → cuts
- For each cut in cuts:
- If (cut 0) Output "Cut lengths must be positive integers." Exit program
- Declare maxCuts ← -1
- Declare minCuts ← largenumber
- Call tryCuts(n, 0, cuts)
- If maxCuts = -1: Output "It is not possible to cut the ribbon using the given lengths." else Output "Maximum number of pieces: " + maxCuts Output "Minimum number of pieces: " + minCuts
-

brute force

- maxCuts ← -1
- minCuts ← largenumber as global variable
- Function tryCuts(remaining, count, cuts)
- If remaining = 0 If count > maxCuts: maxCuts ← count
  If count < minCuts: minCuts ← count
  If remaining < 0
- Return 0 For each cut in cuts
- tryCuts(remaining - cut, count + 1, cuts)

Complexity

Time complexity: T(n)=k

n = total ribbon length

k = number of cut size options

space complexity: O(n)

Explanation

In the main part of the program, the user inputs the total length of the ribbon and how many different cut sizes they want to use. These values get stored in a list of options for cutting. Then, the function tryCuts() kicks in and tries out all possible ways to cut the ribbon based on those sizes. It keeps track of the fewest pieces and the most pieces you can get. In the end, it prints out the best and worst case scenarios, showing how many cuts you might need. This method checks every possible combination without trying to optimize, making it a pretty straightforward example of brute-force searching.

4. DYNAMIC PROGRAMMING APPROACH

The dynamic programming approach breaks down the problem into smaller subproblems and solves larger subproblems using the solutions to the smaller ones. It stores the maximum number of ribbon pieces that can be obtained for each sub-length of the ribbon in a one-dimensional array. The algorithm iterates over the ribbon lengths and computes the optimal solution by considering all possible cuts. The time and space complexity of this algorithm is analyzed, highlighting both its advantages and limitations.

Dynamic Programming Approach to Ribbon Cutting Problem

By decomposing the ribbon cutting problem into smaller, overlapping subproblems, dynamic programming effectively solves it, in contrast to the brute-force method that uses recursion or nested loops to examine every conceivable combination. The optimal answer for every ribbon length from 0 to n is stored in a table (array) rather than being recalculated several times.

Iterating through all lengths i from 1 to n and checking each permitted cut length in the `cuts[]` array for each i is the main idea. The procedure uses previously calculated results $dp[i - cutLen]$ to determine whether cutting at `cutLen` increases the maximum number of pieces for length i for each potential cut length `cutLen`. Iteratively accumulating the ideal number of pieces for each length eliminates the need for extra computations.

Ultimately, $dp[n]$ contains the maximum number of pieces that can be obtained from the ribbon of length n. By effectively examining every possible cut combination and avoiding the exponential time complexity of brute-force approaches, this dynamic programming methodology ensures the optimal answer.

Equation

The equation calculates the maximum number of pieces a ribbon of a given length can be cut into using a specified set of cut lengths.

Let:

- n be the total length of the ribbon.
- a, b, and c be the possible lengths of the ribbon pieces you are allowed to cut.
- i represent a sub-length of the ribbon from 1 to n.
- $R[i]$ be the maximum number of ribbon pieces that can be obtained from a ribbon of length i.

The dynamic programming equation is defined as:

$$R[i] = \max(R[i-a],\ R[i-b],\ R[i-c]) + 1 \quad \text{for all } i \geq \min(a, b, c)$$

Approach

- Assuming we calculate the best way to cut a ribbon of length i by determining if it is possible to make one more cut of size a, b, or c.
  - To demonstrate the new cut we're making, we take the maximum of those previous sub-solutions and add 1.
  - A sub-length (e.g., $i - a$) is ignored or marked as invalid (usually initialized as negative infinity) if it is negative or not reachable (e.g., no prior cuts lead to it).

Initially:

- $R[0] = 0$ (Base case: zero-length ribbon requires zero cuts)
- All other $R[i]$ are initialized to negative infinity to represent invalid states.

The value stored in $R[n]$ will eventually hold the maximum number of pieces the ribbon of length n can be divided into using only cuts of lengths a, b, and c.

If $R[n]$ remains negative, it means the ribbon cannot be cut exactly using the provided lengths.

The Algorithm

The objective is to cut a ribbon of length n units into as many pieces as possible, given an array cuts[] that contains the permitted cut lengths you can make on the ribbon. If the ribbon is ideal, you can cut it into any number of pieces or leave it uncut.

To effectively address this issue, we build a function ribbonCutting that makes use of dynamic programming. The maximum number of pieces that can be obtained from a ribbon of length i is stored in dp[i], which is a dynamic array of size n + 1 that is created inside this method. Since a ribbon of length zero cannot be sliced into any sections, we initialise dp[0] = 0.

The approach is as follows:

1 to n, we iterate across each length i of the ribbon.

For every i, we examine each potential cut length from the `cuts[]` array.

We determine whether cutting at a length produces a greater (bigger) number of pieces than the existing best for length i, provided that the current cut length is less than or equal to i.

After cutting a piece of length `cutLen`, we examine $dp[i - cutLen]$, which is the maximum number of pieces that may be obtained from the remaining ribbon.

If $dp[i - cutLen] + 1$ is greater than the current $dp[i]$ and $dp[i - cutLen]$ is not -1 (indicating that the remaining ribbon can be cut into acceptable pieces), then:

- Since we created an additional piece by cutting at `cutLen`, we update
$$dp[i] = dp[i - cutLen] + 1.$$
- will assist us in reconstructing the precise cuts that resulted in the maximum number of pieces we can cut from the ribbon of length n, which will be stored in $dp[n]$ at the end of this operation.
  The series of cuts produced is then printed using a function call `printCuts(lastCut[], n)` that iterates through the `lastCut` array. Beginning with length n, this function prints each cut by continually subtracting the length of the last cut, as noted in `lastCut`, until it reaches zero. This demonstrates precisely how the ribbon was cut to obtain the maximum number of pieces and validates the dynamic programming method.

By the end, `dp[n]` will hold the maximum number of pieces for the ribbon of length n, and the array `lastCut[]` will help us reconstruct the cuts.

The following pseudocode shows this algorithm:

[H] [1] Length n, array `cuts[]` of allowed cut lengths Maximum number of pieces from ribbon of length n

Initialize array `dp` of size $n + 1$ with -1 Initialize array `lastCut` of size $n + 1$ with 0 `dp[0]` $= 0$

$i = 1$ to n each `cutLen` in `cuts[]` $cutLen \leq i$ and $dp[i-cutLen] \neq -1$ $dp[i-cutLen]+1 > dp[i]$ $dp[i] = dp[i-cutLen]+1$ $lastCut[i] = cutLen$

$dp[n]$

To print the sequence of cuts, we use the following procedure:

[H] [1] Array `lastCut[]`, length n Initialize $length \leftarrow n$ $length > 0$ Print `lastCut[length]` $length \leftarrow length - lastCut[length]$

This prints the lengths of the pieces that together form the optimal cutting solution.

Main function

- Prompt user to enter total ribbon length
- Read `totalLength`
- Prompt user to enter number of cut sizes
- Read `numberOfCuts`
- Declare array `cutSizes` of size `numberOfCuts`
- Prompt user to enter cut sizes
- For $i = 0$ to `numberOfCuts` $- 1$:
  - Read `cutSizes[i]`
- Declare `maxCuts`, `minCuts` as arrays
- Declare `maxPieces`, `minPieces` as integers
- CallribbonCutting(cutSizes, totalLength, maxCuts, minCuts, maxPieces, minPieces)
- Print "Maximum pieces: ", `maxPieces`
- Call `printCutSequence(maxCuts, totalLength)`
- Print "Minimum pieces: ", `minPieces`
- Call `printCutSequence(minCuts, totalLength)`

Explanation:
In the main function, the user enters the total ribbon length and the range of allowed cut sizes. The ribbonCutting() function then receives these values and determines the maximum and minimum number of ribbon pieces. The printCutSequence() function is used to display the matching cut sequences.

ribbonCutting Function

- Function `ribbonCutting(cutSizes, totalLength, maxCuts, minCuts, maxPieces, minPieces)`:
- Declare `Maxpieces` as an array of size `totalLength + 1`, initialized with `-1`
- Declare `Minpieces` as an array of size `totalLength + 1`, initialized with a large number (e.g., `1,000,000`)
- Set `Maxpieces[0]` $= 0$
- Set `Minpieces[0]` $= 0$
- Initialize `maxCuts` and `minCuts` arrays with zeros

Explanation:
This function initializes two arrays: Maxpieces and Minpieces. For every length between 0 and totalLength, these arrays hold the maximum and least number of ribbon pieces that can be obtained. With the exception of length 0, which needs 0 cuts, the values are initialized to indicate no solution.

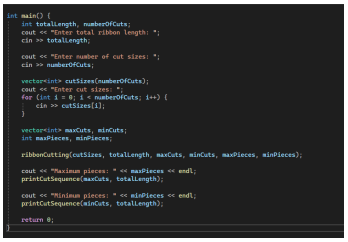

Figure 1. main function

Loop Logic for ribbonCutting Function

- For `i = 1` to `totalLength`:
  - For each `cut` in `cutSizes`:
    - If `i >= cut`:
      - If `Maxpieces[i - cut] -1` and `Maxpieces[i - cut] + 1 > Maxpieces[i]`:
      - Set `Maxpieces[i] = Maxpieces[i - cut] + 1`
      - Set `maxCuts[i] = cut`
      - If `Minpieces[i - cut]` large number and `Minpieces[i - cut] + 1 < Minpieces[i]`:
      - Set `Minpieces[i] = Minpieces[i - cut] + 1`
      - Set `minCuts[i] = cut`
- If `Maxpieces[totalLength] == -1`:
  - Set `maxPieces = 0`
- Else:
  - Set `maxPieces = Maxpieces[totalLength]`
- If `Minpieces[totalLength] == large number`:
  - Set `minPieces = 0`
- Else:
  - Set `minPieces = Minpieces[totalLength]`

Explanation:
This loop tries every cut size at every length to get the perfect number of ribbon pieces (max and min). The value is updated if an accurate cut provides better results. Following the loop, maxPieces and minPieces are given the best results.

Complexity Analysis

Our dynamic programming method for cutting ribbons has a time complexity of $O(n \times m)$, where n is the ribbon's length and m is the maximum number of permitted cut lengths in the `cuts[]` array. This is because, in order to determine the maximum number of pieces, we cycle through all m possible cut lengths for any length i between 1 and n. Our main data structures are arrays, which we fill iteratively. At each stage, the maximum number of pieces is modified using values that have already been calculated.

Since we only need one array dp of size $n+1$ to store the maximum number of pieces for each ribbon length, the space complexity is $O(n)$. Furthermore, in order to reconstruct the solution, we trace the locations where cuts are done using an optional array called `lastCut[]`.