Abstract. This report will outline the dynamic programming solution to the Optimal Parenthesization Problem proposed by R. Bunescu, provide pseudocode for the algorithm that implements the dynamic programming solution, detail the time and space complexity of this solution, and, finally, give a linear time algorithm that achieves the same result. Additionally, we will show that the solution can be extended to $\mathbb{R}$, instead of being limited to only integral inputs.

Contents

1. Introduction 2
2. Characterizing the Structure of an Optimal Solution 2
3. Recursively Defining the Value of an Optimal Solution 3
4. Computing and Constructing the Optimal Solution 3
5. Time and Space Complexity Analysis 4
6. Finding the Optimal Value in Linear Time 5
7. Time and Space Complexity of the Linear Algorithm 6
8. General Remarks and a Theorem 6
1. Introduction

Before we proceed in discussing the problem, let us restate it for ease of the reader.

**Problem 1.** Given a vector of integers, \( A = [A_1, A_2, \ldots, A_n] \), give a dynamic programming solution for finding the parenthesization that maximizes

\[ A_1 - A_2 - \cdots - A_n. \]

For bonus, if it is possible to find the optimal parenthesization in linear time, please provide an algorithm and implementation. If it is not, justify.

We will first go through the dynamic programming approach (Sections 2 through 4). Then, in Section 6 we will analyze the time and space complexity of our dynamic programming solution. Finally, in Sections 6 through 8 we will discuss a linear solution to the problem and also present relevant and interesting remarks regarding this problem.

2. Characterizing the Structure of an Optimal Solution

When considering the problem of optimal parenthesization, it is intuitive that the optimal way to do this is to group the inputs such that we take as little as possible (possibly and preferably negative) away from the maximum possible grouping. Conversely, the way to achieve the minimum is to take as much as possible from as small a group as we can. That is:

\[
(1) \quad \text{OP}([A_1, \ldots, A_n]) = \text{OP}([A_1, A_2, \ldots, A_k]) - \text{OP}([A_{k+1}, \ldots, A_n])
\]

where \( \text{OP}([A]) \) denotes that we are grouping to minimize instead of maximizing—which is denoted simply by \( \text{OP}([A]) \). Now, we will see in the next section that there is a convenient and recursive way to define \( \text{OP}([A]) \). The final remark that we have is about the correctness of Eq. 1. Namely we have that if \( \text{OP}([A_1, \ldots, A_n]) \) is maximal, then \( k \) is such that \( \text{OP}([A_1, \ldots, A_k]) - \text{OP}([A_{k+1}, \ldots, A_n]) \) gives us that maximum difference. Notice that this is not to say that we have that maximum grouping of \([A_1, \ldots, A_k]\) or the minimum grouping of \([A_{k+1}, \ldots, A_n]\). This, in fact, is one of the tricky spots for implementing the algorithm we present: you want the *maximum* difference, not the minimum grouping of \([A_{k+1}, \ldots, A_n]\) once we found the maximum way to group \([A_1, \ldots, A_n]\).

To see this in practice, consider the array \([52 - 1]\). The optimal way to group these numbers is

\[ ((5 - 2) - (-1)) = 4. \]

If we first tried to maximize (here \( k \leq 2 \)) \([A_1, A_2]\) we would want to make \( k = 1 \) since \( 5 > 5 - 2 \). This would, however, leave us with only one way to group \([A_2, A_3]\): \( 2 - (-1) \). This solution yields

\[ (5 - (2 - (-1))) = 2 < 4 \]

and demonstrates why we want to maximize the difference instead of working with the groups individually. Let us formalize this notion in the next section.
3. Recursively Defining the Value of an Optimal Solution

As we saw in the example that ended Section 2, we must be explicit in what we are wanting to maximize. Thus, we introduce the following recursive formula:

\[
OP([A_i, \ldots, A_j]) = \begin{cases} 
A_i & i = j \\
\max_{i \leq k < j} \{ OP([A_i, \ldots, A_k]) - \overline{OP}([A_{k+1}, \ldots, A_j]) \} & i < j.
\end{cases}
\]

Notice that we can actually formulate a similar definition for \(\overline{OP}([A])\):

\[
\overline{OP}([A_i, \ldots, A_j]) = \begin{cases} 
A_i & i = j \\
\min_{i \leq k < j} \{ \overline{OP}([A_i, \ldots, A_k]) - OP([A_{k+1}, \ldots, A_j]) \} & i < j.
\end{cases}
\]

Thus, we will need to implement code that uses this double recursion within our program so that we can find the maximum way to group our input. Thus, for \(i < j\), we have the following single line formula for \(OP([A])\):

\[
OP([A_i, \ldots, A_j]) = \max_{i \leq k < j} \left\{ \{ OP([A_i, \ldots, A_k]) - \min_{k+1 \leq h < j} \{ \overline{OP}([A_{k+1}, \ldots, A_h]) - OP([A_{h+1}, \ldots, A_j]) \} \} \right\}.
\]

4. Computing and Constructing the Optimal Solution

The following pseudocode computes the optimal value in a bottom up fashion (the pseudocode is based off of Matlab code in the sense that \(M(i, i)\) represents the \((i, i)\)th entry in the matrix \(M\), and all arrays/loops initialize (by default) at 1 (not 0 like some languages).

```
1: Dynamic_Parenth(A)
2: for i = 1 : n do
3: MaxVal(i, i) = A(i)
4: MinVal(i, i) = A(i)
5: MinInd(i, i) = 0
6: MaxInd(i, i) = 0
7: end for
8: for l = 2 : n do
9: for i = 1 : (n - l + 1) do
10: j = i + l - 1
11: MaxVal(i, j) = max_{i \leq k < j} \{ Max(i, k) - min_{k+1 \leq h < j} \{ Min(k+1, h) - Max(h + 1, j) \} \}
12: end for
13: MinInd(k + 1, j) = h /*Store argmin of the minimum part of the Max equation*/
14: MaxInd(i, j) = k /*Store argmax of the maximum part of the Max equation*/
15: end for
16: return MaxVal(1, n)
17: return display_max_parens(MinInd, MaxInd, A, 1, n) /*Display the optimal parenthesization*/
```

Lines 2 through 16 compute the maximum value and store the indices related to the optimal grouping, while Line 17 uses the following programs to display the optimal solution:
1: display_max_parens(MinInd,MaxInd,A,i,j)
2: if i == j then
3:    print num2str(A(i))
4: else
5:    k = MaxInd(i, j)
6:    l = k + 1
7:    if l ≤ j then
8:       print '('
9:       print display_max_parens(MinInd,MaxInd,A,i,k) - display_min_parens(MinInd,MaxInd,A,l,j)
10:      print ')'
11: else
12:    print '('
13:    print display_max_parens(MinInd,MaxInd,A,i,j-1) - num2str(A(j))
14:    print ')'
15: end if
16: end if

1: display_min_parens(MinInd,MaxInd,A,i,j)
2: if i == j then
3:    print num2str(A(i))
4: else
5:    k = MinInd(i, j)
6:    l = k + 1
7:    if l ≤ j then
8:       print '('
9:       print display_min_parens(MinInd,MaxInd,A,i,k) - display_max_parens(MinInd,MaxInd,A,l,j)
10:      print ')'
11: else
12:    print '('
13:    print display_min_parens(MinInd,MaxInd,A,i,j-1) - num2str(A(j))
14:    print ')'
15: end if
16: end if

When we make the call to the function display_max_parens(...) the recursion takes place until it is called on (1, 1), the output is the optimal parenthesization which yields the optimal value returned on Line 16 of Dynamic_Parenth(A).

5. Time and Space Complexity Analysis

In Dynamic_Parenths(A) we have to maintain four $nxn$ matrices, even though we only update half of the $n^2$ entries. We also have to maintain temporary arrays for the $\max_{i \leq k < j}$ and the $\min_{k+1 \leq h < j}$ operations, but these arrays are at most length $n$. Finally, we store a constant number of variables for use throughout the loops, but they are rewritten each iteration since the necessary values will be stored in the arrays. Thus, we have the following polynomial space complexity:

$$S_{DP}(n) = 4n^2 + 3n + c = \Theta(n^2).$$
The last analysis we will do is a time complexity analysis for Dynamic_Parenth(A). Lines 2 through 7 take $n$ time to initialize our matrices, but Lines 8 through 15 are not as obvious at first. Notice that we have at least three nested loops—one for $l$, one for $i$ within the $l$ loop, and one within the $i$ loop to find the maximum. Notice, though, that we are also finding the minimum within finding the maximum. Thus, we have a third nested loop. The only question is the time complexity for the MAX and MIN loop, since the $l$ loop runs $n - 1$ times and the $i$ loop runs $n - l + 1$ times. The MAX loop runs $j - i \leq n$, while the MIN loop runs $j - (k + 1) \leq n - 1$. Finally, when printing the optimal solution we call the recursive program at most $n$ times (if the optimal way to group the array $A$ is to put parentheses around $A_2$ through $A_n$ individually. Thus, we have as our time complexity

$$T_{DP}(n) = n + n^4 + n = n^4 + 2n = \Theta(n^4).$$

It might be possible to reduce the time complexity to $\Theta(n^3)$ with clever computation of $\text{Min}(k + 1, j)$, but our implementation embeds the MIN loop within the MAX loop.

6. Finding the Optimal Value in Linear Time

It is in fact possible to find both the optimal value and the optimal parenthesization in linear time—we can actually find both at the same time. We will first provide the pseudocode, then the idea behind it.

1: Linear_Parenth(A)
2: Group(1) = A(1)
3: Group(2) = A(2)
4: $c = 2$
5: $str = \text{print}('num2str(A(1)) - (num2str(A(2)))')$
6: for $i = 3 : \text{length}(A)$ do
7: if $A(i) \geq 0$ then
8: $str = \text{string_concatenate}(str, \text{print}('-num2str(A(i))'))$
9: $Group(c) = Group(c) - A(i)$
10: else
11: $str = \text{string_concatenate}(str, \text{print}('-num2str(A(i))'))$
12: $c = c + 1$
13: $Group(c) = A(i)$
14: end if
15: end for
16: $str = \text{string_concatenate}(str, \text{print}(''))$
17: return $str$
18: return $Group(1) - \text{sum}(Group(2 : \text{end}))$

The algorithm breaks the input $A(2 : \text{end})$ up into minimum groups. Since this notion will be key to a theorem we state later, let us define it now.

**Definition 1.** Let $A = [A_1, \ldots, A_n]$ be an array of integers. We say that $[A_i, A_{i+1}, \ldots, A_j] \subseteq [A_1, \ldots, A_n]$ (with $i \leq i + 1 \leq \cdots \leq j$) is a **minimal group** if

$$(A_i - A_{i+1} - \cdots - A_j) < (A_i - A_{i+1} - \cdots - A_j - A_{j+1}).$$

Notice that our algorithm makes $A_1$ into its own group, then subtracts from it minimal groups. This ensures that, when a minimal group is negative, we are adding to $A_1$, and
when it is positive, we are subtracting as little as possible from $A_1$.

7. Time and Space Complexity of the Linear Algorithm

We begin by look at the amount of space that must be maintained for an input of size $n$. Unlike the dynamic programming solution there are no matrices to maintain. We must maintain the string that will be displayed, but it contains only the $n$ integers of $A$, $n - 1$ subtraction signs, and at most $2n - 2$ parentheses. Thus, the string requires at most $n + (n-1) + 2n - 2 = 4n - 3$ space. The array $Group$ has at most length $n$, which happens only when $A_i < 0$ for all $i = 1, 2, \ldots, n$. Thus, the space complexity for the linear time algorithm is

$$S_{LP}(n) = 4n - 3 + n = 5n - 3 = \Theta(n).$$

The time complexity is rather straightforward since all that we do is iterate over a loop of length $n - 2$ once. Thus, the time complexity is

$$T_{LP}(n) = n + c = \Theta(n).$$

Therefore, it is possible to solve the optimal parenthesization problem in linear time.

Finally, as a comparison, we ran the dynamic implementation and the linear time on a randomly generated array with length 200. The dynamic implementation took 94.975181 seconds while the linear implementation terminated in 0.038889 seconds (no, that is not a typo). Clearly it is better to use the linear solution when the input array is very large.

8. General Remarks and a Theorem

The first remark that we make is that this problem does not need to be constrained to only the integers, we can extend it to all of $\mathbb{R}$. This is because the crux of the linear time algorithm relies only on checking whether $A_i$ is positive or negative. The last thing we will present in this report is a theorem about the input $A$ and the optimal value returned.

**Theorem 2.** Let $A = [A_1, A_2, \ldots, A_n]$ be a vector of real numbers such that $n \geq 3$, and define $OP(A)$ to be the optimal value attained from the optimal parenthesization of $A$. Moreover, let $B = [A_1, A_2, |A_3|, |A_4|, \ldots, |A_n|]$. Then, $OP(A) = OP(B)$.

We will not go through all of the details of the proof, but we will demonstrate and discuss the keys to a rigorous proof. The easiest way to see why the theorem is true is to consider the optimal parenthesization of $A$ as given by the linear time algorithm. If there exists a negative entry after $A_2$, then there will be at least three 'groups' in the array $Group$ within the algorithm. Now, to gain insight, suppose that there exists only one negative entry—call it $A_j$—between the second and last entries of $A$ (exclusive of the endpoints), and that $A_n \geq 0$. Thus, as mentioned before, we will have three 'groups', and so our optimal value will be of the form

$$OP(A) = A_1 - (A_2 - C_1) - (A_j - C_2)$$

where $C_1 = A_3 - A_4 - \cdots - A_{j-1}$ and $C_2 = A_{j+1} - \cdots - A_n$. Since we know that all of the terms that make up both $C_1$ and $C_2$ are positive and that $A_j < 0$, then the above equation becomes
\[ OP(A) = A_1 - A_2 + \sum_{i=3}^{j-1} A_i + \sum_{i=j+1}^{n} A_i + |A_j|. \]

Notice, though, that the only reason that we had \( C_1 \) and \( C_2 \) was that \( A_j < 0 \). Thus, we could have achieved the same result if we had started with \(|A_j|\) to begin with.

The same procedure used here could be used for every occurrence of a negative entry between \( A_3 \) and \( A_n \), inclusively. Thus, we see that the optimal value of \( A \) is the same as for concatenating \( A_1 \) and \( A_2 \) with the absolute values of \( A_3 \) through \( A_n \).