Jonathan Borghese

February 22, 2022

CS 5114

Line Breaking problem

Problem Description

The Line breaking problem is the problem of choosing where to put line breaks within a set of words to achieve an appealing look. The appealing look we are looking for is where the line endings all happen at around the same places. Long areas of space at the ends of the lines are discouraged.

For this problem, the input parameters are the words as well as the maximum line length in characters. The total cost is the sum of the squares of the remaining spaces at the end of each line. The goal is to minimize this cost function to achieve the most even line endings. The output of of this algorithm are the locations of the optimal line endings to achieve this minimal cost.

Dynamic Programming Advantage

The optimal structure for this solution consists of an optimal substructure of a smaller solution plus the cost of the remaining words left. Let C(n) denote the minimum possible cost for arranging 'n' words. Let LC(i, j) be the cost to place words 'i' through 'j' onto a single line. Let 'k' be an index greater than 0 and less than 'n'. The total cost for the next value (n+1) is:

$$C(n+1) = C(k) + LC(k+1, n+1)$$

for some value of 'k'. The value 'k' represents which sub-problem is part of the optimal solution and how many words go on the next line (n – k). This results in the optimal solution for C(n+1). Putting it all together, we have:

$$C(n) = \begin{cases} 0 & if\ n = 0 \\ \min\limits_{0 \le k \le n} (\ C(k) + LC(k+1, n)\ ) & if\ n > 0 \end{cases}$$

$$LC(i,j) = LineLength - \sum_{k=i}^{j} Word(k)$$

For the recursive formulation, C(0) = 0 because the cost for zero words should be zero. The algorithm works for any constant here but for simplicity zero is used. The function $LC(i,j)$ is an N x N array with each index needing to be calculated before the recursive function call. Word(i) is the length of word 'i'. If the result is negative, infinity is placed into the LineCost array.

An algorithm to calculate C(n) using a brute-force method takes exponential time which is not feasible for a large n. The dynamic programming complexity is $\Theta(n^2)$ making this solution orders of magnitude faster than its recursive counterpart.

Implementation

Top-down

Here is the pseudo code for the top-down approach:

```
TOP-DOWN-HELPER(index, Cost, Solution):
    if Cost[index] is set
        return Cost[index]
    min_cost <- INF
    For J = I to 0
        new_cost <- TOP-DOWN-HELPER(j, cost, solution)
        if new_cost + Line_cost[j, index] < min_cost
            min_cost <- new_cost
    Cost[index] = min_cost
    record solution in Solution array
    return min_cost

TOP-DOWN(N, Line_cost):
    Initialize Cost array
    Initialize Solution array
    Top-DOWN-HELPER(N, Cost, Solution)
    return Solution array
```

The main function simply initializes the cost and solutions arrays and then calls the recursive helper function 'TOP-DOWN-HELPER'. This function first checks to see if the cost value for the given index has already been calculated. If it has, it just returns the value. Otherwise, it attempts to find the cost value using an exhaustive search of previous cost values and line costs. Each time a previous cost value is needed, a recursive function call is used to get it. After the minimum cost is found, the answer is cached into the cost array and the solution is saved into the solution array. The minimum cost for the given index is then returned. The 'TOP-DOWN' function then returns the solution array. This algorithm is a top-down approach because it works its way down from largest problem to the smaller ones, calculating them accordingly.

Bottom-up

The bottom-up approach differs from the top-down approach in that it does not have any recursive function calls. Instead, it preemptively calculates all smaller subproblems and works its way up. This is done using a simple for loop looping from 0 to N. Here is the pseudo code for the bottom-up approach:

```
BOTTOM-UP(N, Line_cost):
    Initialize Cost array
    Initialize Solution array
    For I = 1 to N
        min_cost <- INF
        for J = 0 to I
            if Cost[I] + Line_cost[j][i] < min_cost
                min_cost <- Cost[I] + Line_cost[j][i]
        Cost[I] <- min_cost
        Save Answer into Solution array
    return Solution array
```

The algorithm starts by initializing the cost and solution array. The cost array is used to cache previous solution for later use and the solution array is used to store information regarding where the line breaks occur. After the arrays are initialized, the program loops from zero to N times. Each iteration of this loop, the program calculates the value C(I). It starts from C(1) and works its way up to C(N). The cost value is calculated using an exhaustive search each previous solution and the correlating line cost. The minimum cost is kept track of in a variable called 'min_cost'. At the end of this nested loop, this variable holds the minimum cost for C(I). This value is cached into the cost array and the solution array is updated. When the outside loop terminates, the cost array holds the minimum costs for each C(0) to C(N) and the solution array holds the optimal arrangement of line endings for C(N). The algorithm finally returns this optimal solution.

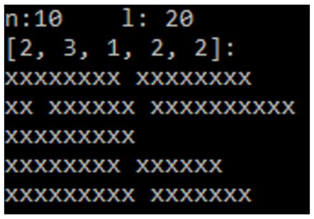
Recursive

Here is the recursive pseudo code:

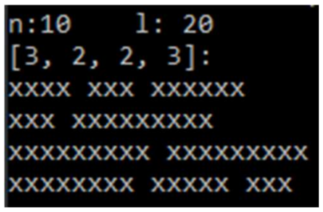
```
RECURSION-HELPER(index, Solution):
      if index == 0
            return 0   // cost[0] is always 0
      min_cost <- INF
      For J = 0 to I
            new_cost <- RECURSION-HELPER(j, solution)
            if new_cost + Line_cost[j, index] < min_cost
                  min_cost <- new_cost
      return min_cost

RECURSION(N, Line_cost):
      Initialize Solution array
      RECURSION-HELPER(N, Cost, Solution)
      return Solution array
```

The recursive algorithm works very similarly to the top-down approach. It also has a helper function that calculates and returns the cost value for a given index. The difference between this recursive solution and the top-down solution is that the helper function doesn't keep track of previous cost calculations. It instead calculates each cost value from scratch each time it needs it. This results in the same cost values being calculated many times over making this algorithm orders of magnitude less efficient than the top-down and bottom-up approaches.

Example Solutions



```
n:10    l: 20
[2, 3, 1, 2, 2]:
xxxxxxxx xxxxxxxx
xx xxxxxx xxxxxxxxxx
xxxxxxxxx
xxxxxxxx xxxxxx
xxxxxxxxx xxxxxxx
```



```
n:10    l: 20
[3, 2, 2, 3]:
xxxx xxx xxxxxx
xxx xxxxxxxxx
xxxxxxxxx xxxxxxxxx
xxxxxxxx xxxxx xxx
```

Here are two example solutions. N is the number of words and L is the line length in characters. The array is the solution array returned from the algorithms. Each index signifies how many words should lie on each line. For example, Sol[0] = 2 means that two words should be on the first line. Prints underneath shows what the text would look like given this configuration. The words used for the input are randomly generated and have lengths ranging from 2 to 10. As you can see, each line has a similar length resulting in an even look.

Time Complexity Analysis

$$LineCost: T(n) = n^2$$

Calculating the LineCost array takes time of n squared. This is because the LineCost array is size n by n and each value in this array has a constant cost to generate. Each algorithm relies on this array to be generated so this n^2 is added to each.

$$Bottom\ up: T(n) = \frac{n^2}{2} + n^2 = \frac{3n^2}{2} = O(n^2) = \Omega(n^2) = \Theta(n^2)$$

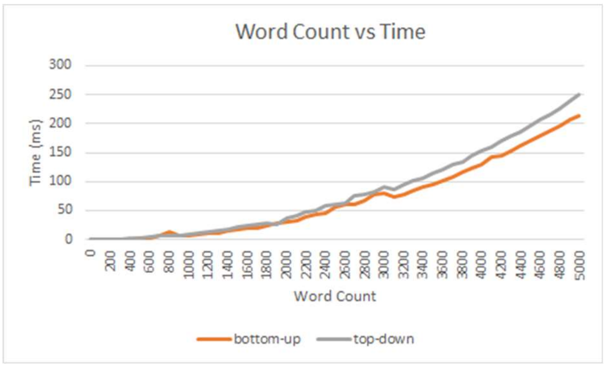
$$Top\ down: T(n) = \frac{n^2}{2} + n^2 = \frac{3n^2}{2} = O(n^2) = \Omega(n^2) = \Theta(n^2)$$

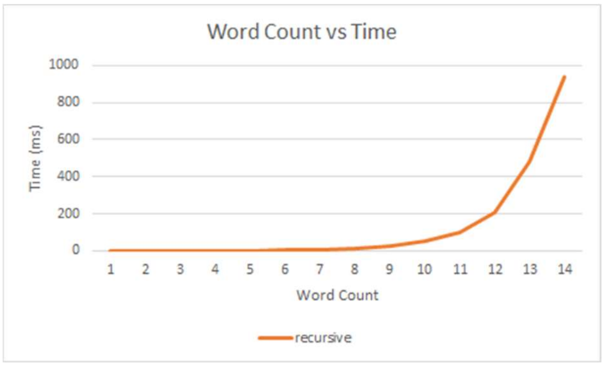
$$Recursive: T(n) = 2^n + n^2 = O(2^n) = \Omega(2^n) = \Theta(2^n)$$

Here are the time complexities for the bottom-up and the top-down algorithms. The $\frac{n^2}{2}$ comes from the nested for loop structure each algorithm has. The n^2 comes from the calculation of the LineCost array. The recursive algorithm has an exponential time complexity because of its recursive nature. This makes it much worse for larger input sizes compared to its bottom-up and top-down counterparts. This difference in time complexity is why dynamic programming works well for this problem.

Sensitivity Analysis

There are two input parameters for the algorithm, the words, and the line length. Each of these parameters affect the efficiency of the algorithm in differently. Here are the results from varying word counts:

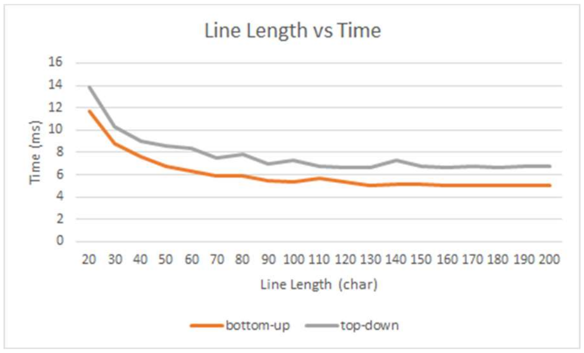


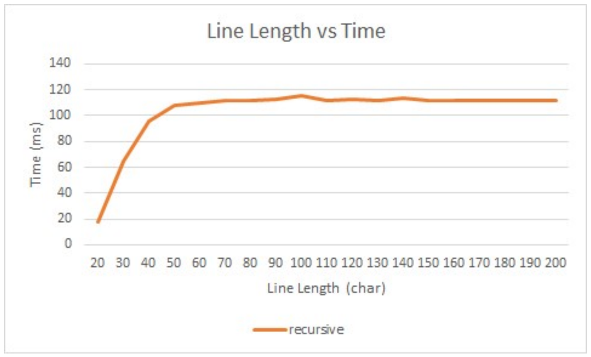


As you can see, both the top-down and the bottom-up show similar behavior. This is expected given their time complexities are the same. One interesting observation is that the top-down algorithm is a little bit slower than its bottom-up counterpart. This is due to the increased amount of function calls in this algorithm. While it takes the same number of commands to complete, the recursive function calls slow it down a bit more.

While the top down and bottom-up algorithms perform similarly, the recursive algorithm is much less efficient. This is shown in the x axis of the graph only reaching 14 words compared to the 5000 in the other graph. This is due to its exponential time complexity.

Here are the results from the varying line lengths:





These two graphs show differences in runtime when varying line length. If the line length is sufficiently long, the algorithm performs the same which makes sense because the line length is not incorporated in the time complexity functions. For the top down and bottom-up algorithms, they performed a bit slower when the line length was very small. I this this is due to having to create a larger solution array for the output. Because the line length is smaller, the output array is larger. Functions ‘deepcopy’ and ‘append’ are used for this solution array making it somewhat expensive to create larger arrays.

For the recursive algorithm, it ran faster with a small line length. This is due to the LineCost array being mostly empty. The reason only the recursive algorithm benefitted from this simple LineCost array is because it is much less efficient. A recursive function call is only done if the LineCost array holds a non-infinite value. This results in less recursive function calls the shorter the line length is resulting in a faster algorithm. This benefit of the LineCost array does not affect the bottom-up and top-down algorithms because they do not calculate the cost array each time it is needed so the time saved is minimal.