1 Word Clouds Revisited

View on Kattis
Tags: Dynamic Programming

1.1 Problem Description

We are given a list of text boxes, represented with a width and a height. We need to lay them out within a box of fixed width. An entry can either be placed horizontally to the right of the previous box (if it fits), or at the far left end of the next row. The height of each row is the sum of the heights of every entry in the row. Our goal is to minimize the total height of the word cloud.

As it turns out, a greedy solution, as shown in Figure 1 may not be the optimal solution, depicted in Figure 2

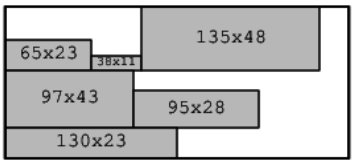

Figure 1: Greedy Strategy

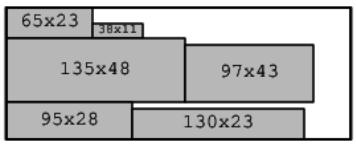

Figure 2: Optimal Solution

1.2 Algorithm

We can make this problem more tractable using dynamic programming. To approach this problem with that strategy, we need to define two things: an optimal substructure, and a data structure for memoizing the result.

Optimal Substructure
An optimal substructure simply represents some way of breaking the problem into smaller pieces, or some relation that we can use to build upon previous solutions to smaller portions of the problem. In this case, we can realize that we only need to track the smallest total height attainable for every possible width j of the final column after inserting the ith entry. Then, to calculate the $i+1$th entry's optimal values, we can iterate over all optimal solutions to the ith case and track the results of both inserting in that row (if possible) or inserting a new row.

Data Structure

If we are given N entries and a width of W, one option is to store everything in an $N \cdot (W+1)$ array. To compute the results of the ith row, we can iterate over all columns of the previous row. However, because many columns in the previous row may not be attainable, we can instead use a map of integers to optimal values, allowing us to cut down on the average case complexity. Additionally, because we only need the ith row to calculate the $i+1$th row, we only need to keep 2 maps, rather than N of them.

Armed with this, we can arrive at the following psuedocode:

Let *best* be a map of width to the optimal solution for that width.
for Rectangle r in the input **do**
  Let *nextBest* be a new map of width to the optimal solution.
  for *width*, *solution* in *best* **do**
    $nextBest(r.width) \leftarrow optimal(nextBest(r.width), addNewLine(best(width), r))$
    if $r.width + width \leq W$ **then**
      $newWidth \leftarrow r.width + width$
      $nextBest(newWidth) \leftarrow optimal(nextBest(newWidth), addSameLine(best(width), r))$
    end if
  end for
  $best \leftarrow nextBest$
end for
Return smallest height in *best*.

The optimal solution is whichever minimizes the total height, while $addNewLine$ and $addSameLine$ can be implemented by following the rules of the layout.

Time Complexity: $\mathcal{O}(N \cdot W)$
Space Complexity: $\mathcal{O}(W)$
Where N is the number of entries ($2 \leq N \leq 5000$) and W is the width of the word cloud ($150 \leq W \leq 1000$).

1.3 Implementation

```
import java.util.*;

public class WordCloudsRevisited {
  public static void main(String[] args) {
    Scanner in = new Scanner(System.in);
    int N = in.nextInt(), W = in.nextInt();

    Map<Integer, Size> memo = new HashMap<>();
    memo.put(0, new Size(0, 0));

    // Iterate over all text boxes.
    for (int i = 0; i < N; i++) {
      int w = in.nextInt(), h = in.nextInt();
      // Will hold the best values after inserting
      // this text box.
      Map<Integer, Size> nMemo = new HashMap<>();
      // Iterate over all trailing widths for the
      // previous insertion.
      for (int width: memo.keySet()) {
        Size prevSize = memo.get(width);
        // If we can insert this into the current
        // row, do so.
        if (width + w <= W) {
          int nCurrentRowHeight = Math.max(h,
              prevSize.currentRowHeight);
          Size nSize = new Size(nCurrentRowHeight,
            prevSize.previousRowsHeight);
          insert(nMemo, width + w, nSize);
        }
        // Always try creating a new row.
        Size nSize = new Size(h,
          prevSize.totalHeight);
        insert(nMemo, w, nSize);
      }
      // Copy over memoization.
      memo = nMemo;
    }
    // Find best at end.
    int best = Integer.MAX_VALUE;
    for (int width: memo.keySet()) {
      best = Math.min(best,
        memo.get(width).totalHeight);
    }
    System.out.println(best);
  }
```

```
  /**
   * Optimal Substucture - We want to store the
   * smallest total height (current row's current
   * height + height of any previous rows) attainable
   * for every possible width of the current row.
   */
  static void insert(Map<Integer, Size> m, int w,
      Size s) {

    if (!m.containsKey(w) || m.get(w).totalHeight >
        s.totalHeight) {
      m.put(w, s);
    }
  }
}

/**
 * Size represents the height of the word cloud
 * after a number of insertions.
 */
class Size {
  int currentRowHeight;
  int previousRowsHeight;
  int totalHeight;

  /**
   * Constructor for size.
   */
  public Size (int currentRowHeight,
    int previousRowsHeight) {

    this.currentRowHeight = currentRowHeight;
    this.previousRowsHeight = previousRowsHeight;
    totalHeight = currentRowHeight+previousRowsHeight;
  }
}
```