Kyle Wetherald
CS 1501
Dr. John Aronis
March 22, 2013

The Cloth-Cutting Problem

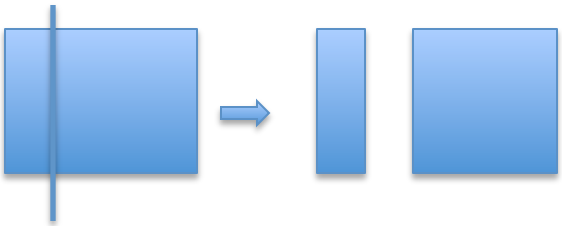
Dynamic programming is a method of solving a wide range of optimization problems by first solving a set of subproblems. Many optimization problems are very complex, but Dynamic Programming makes them easy to solve by cutting the problem up into smaller, simpler problems. Once the subproblems are solved, their solutions are used to find the solution to the original problem.

The cloth-cutting problem is one example of an optimization problem that can be solved using dynamic programming. A description of the problem is as follows:

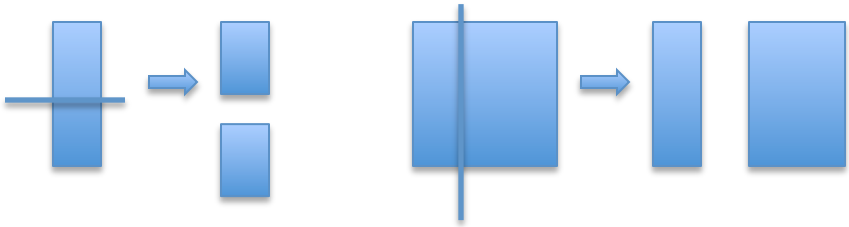
> Suppose you have been given a rectangular piece of cloth with a width and height of X and Y, which are both positive integers. Suppose you are also given a set of n products that could be made from fabric, each requiring a rectangular piece of the cloth. Each pattern has a width and height of w_i and h_i (both positive integers), and a selling price represented by v_i (again, a positive integer) that can be earned by making the product. Finally, suppose you have a machine that is capable of cutting any rectangular piece of cloth. What is the best way to cut the cloth so the optimal profit is made and the least amount of cloth is wasted?

To clarify, the cutting machine would only be able to make horizontal or vertical cuts- the cuts must run the entire length or width of the cloth. Once a cut is made, two smaller rectangles are made; a cut made to one of the rectangles would not affect the other. For instance, suppose we have the following piece of cloth:

and a vertical cut were made as follows, leaving two rectangular cloths:

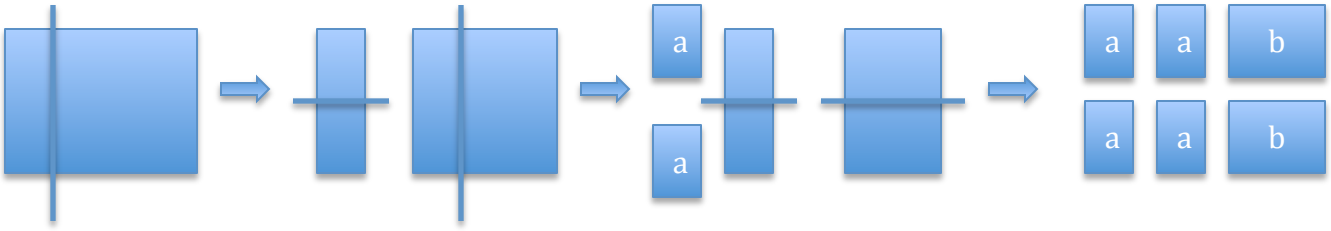


Each of the resulting cloths could then be cut independently:



It is easy to imagine how such a machine could cut a cloth to the size of certain patterns. For instance, suppose we are given the following patterns, with which garments can be made:

Then the original cloth could be cut as follows to make pieces of cloth that are useful for making garments:



The value of the cloth after cutting is equal to the sum of the values of all of the pieces of cloth cut from it:

$$Value = \sum_{i \in garments} value(i)$$

In our example, if using pattern "a" made garments with a value of two, and pattern "b" made garments with a value of three, then the value of the original cloth would be 4*2 + 2*3 = 14. The goal of the cloth-cutting problem is to find the best series of cuts to maximize profit. In this paper, I seek to show how I used dynamic programming to solve the cloth-cutting problem in a time and memory efficient manner.

The first step I took to solve the cloth problem was to create a brute-force, recursive algorithm that simply returned the value of an optimized series of cuts. In pondering the solution, I observed that there are three possibilities for what the optimal step could be for any piece of cloth:

1. No cut is made. The pattern with the most value that fits on the cloth is the optimal value of the cloth itself.
2. A vertical cut is made. The optimal value of the piece of cloth is the sum of the values of the pieces on the left and the right.
3. A horizontal cut is made. The optimal value of the piece of cloth is the sum of the values of the pieces on the top and the bottom.

From here it was easy to see how the recursion would work. First, the algorithm would cycle through the *n* patterns:

```
Optimize(x,y,width,height)
     INIT optimalValue to 0
     FOR each pattern in the set
          IF pattern fits and pattern.value > optimalValue
```

```
            SET optimalValue equal to pattern.value
        ENDIF
    ENDFOR
```

Then try each cut vertically, followed by each cut horizontally:

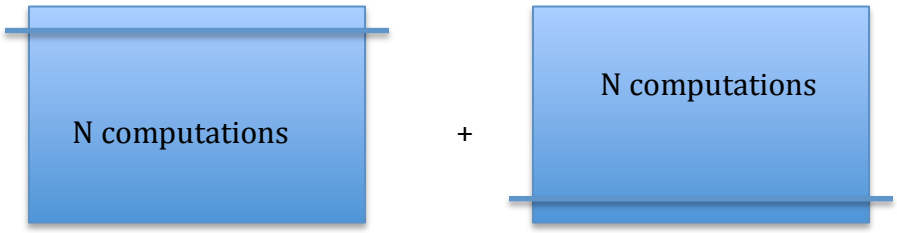
```
    FOR cuts 1 through width
        COMPUTE value as optimized(left)+optimized(right)
        IF value > optimalValue
            SET optimalValue equal to value
        ENDIF
    ENDFOR
    FOR cuts 1 through height
        COMPUTE value as optimized(top)+optimized(bottom)
        IF value > optimalValue
            SET optimalValue equal to value
        ENDIF
    ENDFOR
    RETURN optimalValue
```

The recurrence relation is as follows:

$$V_{w,h} = \max\left(\max_{0<i<w}\left(V_{i,h}\right), \max_{0<j<w}\left(V_{w,j}\right)\right)$$

$$V_{p_w,p_h\{p|p\in Patterns\}} = V_p$$

This algorithm worked, but very inefficiently. To understand the runtime, we can look at the number of computations needed to determine the optimal cut of a certain cloth and compare it to the number of computations needed for a cloth that is one unit taller. Suppose a cloth of height h makes *N* computations. If we were to increment the height by one, or optimize a cloth of height *(h+1),* we would end up making those *N* computations twice: once when optimizing a horizontal cut at position 1, and once when optimizing at position *h*, as seen in the following illustration:



Therefore, increasing *h* by 1, more than doubles the number of computations needed. This makes the number of computations roughly $2^h N$. Similarly, adding 1 to *w* makes an exponential number of computations, roughly $2^w N$. The number of computations for each increase in *w* would be:

$$C_w = 2^w N + w * h$$

The runtime of the algorithm, therefore is $O(2^n)$, or exponential runtime.

To make the algorithm much more efficient, we can add a memo of all of the cloth sizes that have been optimized already. To do this, we can set up a matrix of size *w*x*h*. Each time a cloth of size *X* by *Y* is optimized, the optimized value is added to the matrix at position *[X][Y]*. Then, at the beginning of each `optimize()` function call, we can check the matrix for a value for a cloth of size *X* by *Y*. This significantly reduces the number of optimization computations required.

In a similar fashion as before, if we look at the increase in computations for a single increment to the height or width, we can determine the runtime of the algorithm. Suppose that *N* comparisons are required to optimize a cloth of width *w* and height *h*. A cloth of width *w*+1 would only require an additional *w***h* computations, because all cloths from size 1 by 1 up to *w* by *h* have already been computed. The only computations left to compute would be the optimizations of cloths of sizes *w*+1 by 1, *w*+1 by 2, …, *w*+1 by *h*-1, and *w*+1 by *h*. For each increase in *w*, the number of computations would be: $C_w = N + w * h$, making the runtime polynomial rather than exponential.

To implement the memorized cloth-cutting algorithm in java, I created several new java classes, most of which are storage data structures. The main workhorse was the ClothCutter class. This class took three things:

1. The width of the piece of cloth to be optimized
2. The height of the cloth
3. An ArrayList<Pattern> of patterns that can be cut from the cloth

The patterns were implemented within a data structure I named "Pattern" that housed the dimensions, value, and name of each pattern. The memo was comprised of an array matrix that held instances of a data structure I called "Cloth." Each Cloth instance held an `int` that represented the optimal type of cut (horizontal cut, vertical cut, or not cut at all), and an `int` that represented the position of the cut. In the case of no cut, the 2nd `int` represents the position in the ArrayList of Patterns that the optimal Pattern could be found. One of the snags along the way involved the memo. When adding a value to the memo, I needed to subtract 1 from the height and width, as it is impossible to have a cloth with dimensions 0 by 0. Also, because all of the memo values were initialized to zero, I

needed to find a way to keep track of pieces that were too small to fit any of the patterns, as their value would be zero. Instead of setting their memo value to zero, I set it to -1 so that the optimized value would not be computed every time.

After the optimal cutting scheme was found, I implemented a function called makeCuts() that called the recursive optimization function in order to access the memo. Each time an instance of Cloth was obtained from the memo, the dimensions and the cut information contained in the Cloth instance were used to calculate the coordinates of cuts and patterns. Each time a horizontal or vertical cut was encountered in the Cloth, a new instance of a class, called Cut, was added to an ArrayList<Cut>. The Cut class held information including coordinates of the starting and ending points of a cut to be made in relation to the original bolt of cloth. These coordinates could then later be translated into a line on a JPanel to visually represent the optimal cuts. Each time the makeCuts() function encountered a no-cut value, an instance of a class, called Garment, was added to a separate ArrayList<Garment>. The Garment class held information about the starting coordinates, and height and width of the optimal pattern to be used on the piece of cloth. This information could later be translated into a rectangle to be placed on a JPanel to visually represent the optimal patterns.

In order to create the visual representation of the optimal cloth-cutting strategy, I created a JPanel extension that allows for the painting of garments and cuts by placing lines and rectangles in the appropriate places based on the dimensions calculated in the makeCuts() function in the ClothCutter class. I implemented a "magicNumber" that is used to multiply the dimensions and coordinates so that they show up as an appropriate size on the screen. For instance, if the original cloth were of size 15x30, I did not want

the JPanel to be 15 pixels by 30 pixels as it would be nearly impossible to see where each of the cuts should be made. Instead, 15 and 30 could be multiplied by my magicNumber to make the JPanel closer to 800x600 pixels, which would be an appropriate size on most personal computer monitors.

The cloth-cutting problem is a practical real life example of how dynamic programming can be used to simply and efficiently come up with solutions to complex problems. As my analyses of the runtimes of both a simple recursive and dynamic solution show, a dynamic programming solution to the cloth-cutting program is significantly faster than other solutions. Because less recursive calls are needed to complete the problem, it is also significantly more memory efficient. This shows that dynamic programming is not only a viable way to solve optimization problems, but is a desirable method of problem solving.