CSE 101 - Winter 2014
Homework 4

February 26, 2014

Note 1: For all dynamic programming algorithms, you are required to (1) describe the subproblems in words and (2) provide a recurrence relation.

Note 2: The first six questions of this assignment, worth 10 points each, can give you up to 60/60 points. The seventh question is a bonus question that can give you up to 10 additional points.

1. (Dasgupta et al. textbook, Problem 6.8) Given two strings $x = x_1x_2 \dots x_n$ and $y = y_1y_2 \dots y_m$, we wish to find the length of their longest common *substring*, that is, the largest k for which there are indices i and j with $x_ix_{i+1} \dots x_{i+k-1} = y_jy_{j+1} \dots y_{j+k-1}$. Give a dynamic programming algorithm to find the largest k. Please provide a proof of correctness and running time of your algorithm.

Solution:

Consider the problem of finding the length of the longest common substring which is a suffix in both x and y. If $x_n = y_m$, then the length of the longest common suffix of x and y is one more than the length of the longest common suffix of $x' = x_1x_2 \dots x_{n-1}$ and $y' = y_1y_2 \dots y_{m-1}$. This optimal substructure property can be used to design an efficient DP algorithm to find the longest common suffix for each combination of strings $x_1x_2 \dots x_i$ and $y_1y_2 \dots y_j$ for $1 \leq i, j \leq n$.

The length of the longest common substring in x and y is thus the maximum of the length of the longest common suffix over all subproblems defined below.

Subproblem definition:

We define $L(i, j)$ to be the length of the longest common substring ending at position i in X and at position j in Y. $L(1, 1) = 1$ if $x_1 = y_1$, and $L(1, 1) = 0$ otherwise.

Recurrence relation:

The recurrence for $L(i, j)$ is

$$L(i,j) = \begin{cases} 1 + L(i-i, j-1) & \text{if } x_i = y_j \\ 0 & \text{if } x_i \neq y_j \end{cases}$$

Proof of Correctness:

Let the longest common substring, S, of x and y, end at position i in x and at position j in y. Then, S is the longest common suffix in $x_1x_2 \dots x_i$ and $y_1y_2 \dots y_j$. Also, no common suffix can be longer than the length of the longest common substring since the common suffix is also a common substring. Taking the length of the longest common suffix over all subproblems thus gives us the length of the longest common substring.

Proof by induction

The base case is $i = 1, j = 1$: In this case, if the two first characters are same then $L(1, 1) = 1$, else it is 0.

For the Induction Hypothesis, suppose that the recurrence is correct for all values of $i \leq k$ and all values of $j \leq l$.
Induction Step:
When computing $L(k+1, j), \forall j \leq l$, the following cases apply:

1. If $x_{k+1} = y_j$, then $L(k+1, j)$ is the same as $L(k, j-1)$, plus 1 to account for the matching characters at the end of x and y. The subproblem $L(k, j-1)$ gives the correct length of the longest common prefix by the Induction Hypothesis.

2. If $x_{k+1} \neq y_j$, there cannot be any common substring that ends at position $(k+1, j)$. Therefore, $L(k+1, j) = 0$ in this case.

   The proof for cases $L(i, l+1), \forall i \leq k$ and $L(k+1, l+1)$ is analogues to the one above. This completes the proof.

Pseudocode

```
L(n, m) //Matrix of size n × m to store values of L(i,j)

for i from 0 to n:
 L(i, 0) = 0
end for
for i from 0 to m:
 L(0, i) = 0
end for

for i from 1 to n:
 for j from 1 to m:
  if(X[i] == Y[j]):
   L(i, j) = 1 + L(i − 1, j − 1)
  else:
   L(i, j) = 0
  end if
 end for
end for

longest_substring = 0

for i from 1 to n:
 for j from 1 to m:
  if(L(i, j) > longest_substring):
   longest_substring = L(i, j)
  end if
 end for
end for

return longest_substring
```

Running time: As $L(i, j)$ is calculated for each combination of i and j, there are $O(n^2)$ total subproblems. Given the values of the preceding subproblem $L(i-1, j-1)$, $L(i, j)$ can be calculated in $O(1)$ time. Finally, the values of all the subproblems have to be scanned to find the largest

common substring, which takes an additional $O(n^2)$ time. Therefore, the overall running time of the algorithm is $O(n^2)$.

2. (Dasgupta et al. textbook, Problem 6.3) Yuckdonald's is considering opening a series of restaurants along Quaint Valley Highway (QVH). The n possible locations are along a straight line, and the distances of these locations from the start of QVH are, in miles and in increasing order, $m_1, m_2, \ldots, m_n$. The constraints are as follows:

   - At each location, Yuckdonald's may open at most one restaurant. The expected profit from opening a restaurant at location i is p_i, where $p_i > 0$ and $i = 1, 2, \ldots, n$.
   - Any two restaurants should be at least k miles apart, where k is a positive integer.

   Give a dynamic programming algorithm to compute the maximum expected total profit subject to the given constraints. Please provide a proof of correctness and running time of your algorithm.

Solution:

Consider an optimal set of locations to open the restaurants. In this optimal set, if the n^{th} location is not chosen, then the optimal set is same as the optimal set for first $n-1$ locations.

If the n^{th} location is chosen, then the optimal set consists of the n^{th} location and the optimal set using only the first i location, where i is maximum such that $m_n - m_i \geq k$. This is an optimal substructure property that can be used to design a DP algorithm for this problem.

Subproblem definition:

Define $P(i)$ to be the maximum total profit achievable using only the first i locations. The maximum total profit is then given by $P(n)$.

Recurrence relation:

In figuring out $P(j)$, we have two choices: either place a restaurant at location j or not. In the former case, we must skip all locations which are closer than k miles to the left of location j. To help with this, define $prev(i)$ to be the largest index i with $m_j \leq m_i - k$, or 0 if there is no such index. $P(0) = 0$.

Then:

$$P(i) = \max\{p_i + P(prev(i)), P(i-1)\}$$

Proof of correctness:

The proof of correctness follows from the correctness of the recurrence relationship as shown above.

Proof by induction

The base case of $P(0) = 0$ is true because if there are no locations, then there is no profit.
Assume now that $P(i)$ gives the correct value of profit for all $i \leq k$.
To prove that $P(k+1)$ give the correct value of maximum profit from first $k+1$ locations, we have to consider two cases.
Case 1: Location $k+1$ is not chosen.
In this case, only first k locations are considered. However, in this case, we already know optimal profit to be $P(k)$ from the induction hypothesis.
Case 2: Location $k+1$ is chosen.
In this case, since location $k+1$ is chosen, we cannot chose any location after $prev(k+1)$. Note, however, that $P(i)$ is an increasing series. Thus the optimal profit is obtained by choosing maximum profit using first $prev(k+1)$ locations and adding profit from location $k+1$.

Pseudocode

$P(1, \ldots, n)$ `//Stores maximum profit obtainable by using first` i `locations`
$prev(1, \ldots, n)$ `//` $prev(i) = j$ `Stores the rightmost location` j `for which` $m_i - m_j \geq k$

$prev(1) = 0$
$P(0) = 0$

`//Calculate values for` $prev$
`for` i `from 2 to` n`:`
 `if(`$m_i - m_{i-1} > k$`):`
  $prev(i) = i - 1$
 `else:`
  $prev(i) = prev(i - 1)$
 `end if`
`end for`

`for` i `from 2 to` n`:`
 $P(i) = \max\{p_i + P(prev(i)), P(i - 1)\}$
`end for`

`return` $P(n)$

Running time:

$P(0)$ takes $O(1)$ time to calculate. Each successive subproblem $P(j)$ takes $O(1)$ time to calculate given the values of the preceding subproblems. Therefore, the total running time of the algorithm is $O(n)$.

3. (Kleinberg and Tardos, Algorithm Design, Problem 6.7) We wish to study the price of a given stock over n consecutive days, numbered $i = 1, 2, \ldots, n$. For each day i, $p(i)$ is the price per share for the stock on that day. (Assume that the price of a share does not change during each day.) How should we choose a day i on which to buy the stock and a later day $j > i$ on which to sell it, if we want to maximize the profit per share, i.e., $p(j) - p(i)$? If there is no way to make money during the n days, we should calculate the profit to be zero. Describe a dynamic programming algorithm to find the optimal numbers i and j in $O(n)$ time. Please provide a proof of correctness and running time of your algorithm.

Solution:

We observe that the maximum profit by selling on some day j is achieved if we bought stock on a prior day i, such that the price of the stock on day i is the minimum of all the days in the range $1 \leq i \leq j$. We also notice that if we sell stock on day j, the best day to buy stock is either j itself (if it is the minimum of all the days 1 to j), or the same as the best day to buy stock when selling on day $j - 1$ (as the day with the minimum price of stock prior to j is the same day to buy stock if we sell on day $j - 1$). This shows that the problem has an optimal substructure property that we use to design a DP algorithm.

Subproblem definition: Let $P(j)$ be the maximum profit achievable by selling stock on the j^{th} day, having bought stock on some day i, such that $i \leq j$. We have $P(1) = 0$.

As we need to record the day i that we bought the stock on, let $buy(j) = i$ record the day i when the stock was bought. We have $buy(1) = 1$.

The optimal day to sell can then be found by looking through the array P and finding the index j such that $P(j)$ is maximum. The optimal value of i is then $buy(j)$.

Recurrence relation:

We get the following recurrence

Then:

$$P(j) = \max\{P(j-1) - p(j-1) + p(j), 0\}$$

$$buy(j) = \begin{cases} j & \text{if } P(j) = 0 \\ buy(j-1) & \text{otherwise} \end{cases}$$

Proof of correctness: The proof of correctness follows from the correctness of the recurrence.

Proof by induction

The base case of $P(1) = 0$ is true because if we buy and sell on the 1^{st} day itself, then the maximum profit obtainable is 0.

For the Induction Hypothesis, assume that $P(j)$ correctly find the maximum profit obtainable by selling on day j for all $j \leq k$.

The maximum profit achievable by selling on day $k+1$ is 0 if the price of the stock on day $k+1$ is the minimum of all the days from 1 to $k+1$, and correspondingly, $buy(k+1) = k+1$. Otherwise, the maximum profit on selling stock on day j is achieved by buying stock on the same day that enabled us to achieve maximum profit on day k, i.e., $buy(k)$. The profit thus achieved on day $k+1$ can be calculated from the maximum profit achieved on day k (true by Induction Hypothesis), plus the difference between the price of stock on days $k+1$ and k.

Pseudocode

```
P(1,...,n) //Stores maximum profit obtainable by selling on day i
buy(1,...,n) //Best day to buy if selling on day i  P(1) = 0
buy(1) = 1

for i from 2 to n:
 P(i) = max{P(i-1) - p_{i-1} + p_i, 0}
 if(P(i) = 0):
  buy(i) = i
 else:
  buy(i) = buy(i-1)
 end if
end for

best_buy = 1 //best day to buy
best_sell = 1 //best day to sell

for i from 1 to n:
 if (P(i) > P(best_sell)):
  best_sell = i
  best_buy = buy(i)
 end if
end for

return best_buy, best_sell
```

Running time:

Each subproblem $P(j)$ can be solved in $O(1)$ time given the values of the preceding subproblems. Filling in the value of $buy(j)$ also takes $O(1)$ time. Therefore, filling in all the n subproblems takes $O(n)$ time. An additional pass over all the profits achieved on each of the days has to be made to find that day j that gives us the maximum achievable profit, which takes $O(n)$ time. Looking up its corresponding $buy(j)$ value is then again $O(1)$ time. Therefore, the overall running time of the algorithm is $O(n)$.

4. (Dasgupta et al. textbook, Problem 6.7) A subsequence is palindromic if it is the same whether read left to right or right to left. For instance, the sequence

$$A, C, G, T, G, T, C, A, A, A, A, T, C, G$$

has many palindromic subsequences, including A, C, G, C, A and A, A, A, A (on the other hand, the subsequence A, C, T is not palindromic). Design a dynamic programming algorithm that takes a sequence $x[1, \ldots, n]$ and returns the (length of the) longest palindromic subsequence. Please provide a proof of correctness and running time of your algorithm.

Solution:

Consider any substring $x[i, \ldots, j]$ of $x[1, \ldots, n]$. If $x[i] = x[j]$, then the longest palindromic subsequence of $x[i, \ldots, j]$ is the longest palindromic subsequence of $x[i+1, \ldots, j-1]$ appended with the character $x[i]$ on both sides. If $x[i] \neq x[j]$, then the longest palindromic subsequence of $x[i, \ldots, j]$ cannot have both $x[i]$ and $x[j]$. In this case the longest palindromic sequence of $x[i, \ldots, j]$ is found either in $x[i+1, \ldots, j]$ or $x[i, \ldots, j-1]$. This observation provides us with the required substructure for formulating a DP algorithm.

Subproblem definition:

Let $T(i, j)$ be the length of the longest palindromic subsequence of the substring $x[i \ldots j]$. $T(i, j) = 0 \ \forall i, j, \ j < i$. T(1,n) gives the length of the longest palindromic sequence in $x[1, \ldots, n]$.

Recurrence relation:

We have the following recurrence.

$$T(i, i) = \begin{cases} 1 & \\ 2 + T(i+1, j-1) & \text{if } x[i] = x[j] \\ \max\{T(i+1, j), T(i, j-1)\} & \text{otherwise} \end{cases} \quad (1)$$

Proof of correctness:

The proof of correctness follows from the correctness of the recurrence.

We prove that $T(i, j)$ correctly finds the length of the longest palindromic subsequence in $x[i, \ldots, j]$.

Proof by induction

The base case is when the $i = j$. In this case, the character $x[i]$ is a palindrome of length 1. Thus, the base case of $T(i, i) = 1$ is correct.

For the Induction Hypothesis, we assume that $T(i, j)$ correctly finds the length of the longest palindromic subsequence in $x[i, \ldots, j]$ for all i, j such that $j - i \leq k$.

For the Induction Step, there are two cases:

1. $T(i, i+k+1) = 2 + T(i+1, i+k)$ if the characters on both ends of the subsequence are equal, i.e., $x[i] = x[i+k+1]$.

2. Otherwise, we consider the maximum of the lengths of the two subproblems that are constructed by extending the sequence on either side: $T(i, i+k+1) = \max(T(i+1, i+k+1), T(i, i+k))$

Both of these cases give the correct result by the Induction Hypothesis.

Pseudocode

```
T(n, n) //Matrix to store length of longest palindrome subsequence in substring x[i, ..., j]

for i from 1 to n:
 T(i, i) = 1
 T(i, i-1) = 0
end for

for k from 1 to n-1:
 for i from 1 to n-k:
  if (x[i] == x[i+k]):
   T(i, i+k) = 2 + T(i+1, i+k-1)
  else:
   T(i, i+k) = max{T(i+1, i+k), T(i, i+k-1)}
  end if
 end for
end for

return T(1, n)
```

Running time:

As there are n^2 subproblems formed by all combinations of i and j in $x[i \dots j]$, each one taking $O(1)$ time to compute, the overall running time of the algorithm is $O(n^2)$.

5. (CLRS textbook, Problem 16.2) Consider the problem of neatly printing a paragraph on a printer. The input text is a sequence of n words of lengths $l_1, l_2, \dots, l_n$, measured in characters. We want to print this paragraph neatly on a number of lines such that each line holds a maximum of M characters. The criterion of "neatness" is as follows: if a given line contains words i through j and we leave exactly one space between words, the number of extra spaces at the end of the line is $M - j + i - \sum_{k=i}^{j} l_k$.

   We wish to minimize the sum, over all lines except the last, of the number of extra spaces at the end of lines. Give a dynamic programming algorithm to print a paragraph of n words neatly on a printer. Please provide a proof of correctness of your algorithm. Analyze the running time and space requirements of your algorithm.

Solution:

We consider printing all the words 1 through n on one line. If all the words fit on one line, then we have obtained the solution. However, if all the words cannot fit on a single line, we consider all possibilities of neatly printing the first i words, followed by the remaining $n - i$ words on one

line. The fact that the first i words are printed neatly gives us the desired optimal substructure. We can thus define a subproblem as follows.

Subproblem definition:

Let $s(i,j) = M - j + i - \sum_{k=i}^{j} l_k$. A positive value of $s(i,j)$ will indicate the the words i through j can fit on one line. Let $c(i,j)$ be the cost of placing the words i through j on a single line. Let $C[j]$ be the optimal cost of printing the first j words, such that word j is printed on the last line. We have $C[1] = 0$. $C[n]$ gives the minimum possible sum of the number of spaces at the end of non-terminal lines.

Recurrence relation:

We have the following cases:

1. If $s(i,j) < 0$, we are placing too many words on this line, and we set $c(i,j) = \infty$ in this case.
2. If $j = n$, i.e., the last word is printed on this line, and $s(i,j) \geq 0$, then the cost of this line is 0.
3. In all other cases, $c(i,j) = s(i,j)$.

Then, $c(i,j) = \infty$ if $s(i,j) < 0$, $c(i,j) = 0$ if $j = n$ and $s(i,j) \geq 0$, and $c(i,j) = s(i,j)$ otherwise.

The recurrence for $C[j]$ is

$$C[j] = \min_i \{C[i-1] + c(i,j)\}, \forall 1 \leq i \leq j$$

Proof of correctness:

The proof of correctness follows from the correctness of the recurrence.

Proof by induction

The base case is when there are no words. In this case $C[0] = 0$ is the correct value.

For the Induction Hypothesis, assume that $C[i]$ gives the correct cost of neatly printing the first i words for all $i \leq k$.

For the Induction Step, we observe that to print the first $k+1$ words, we can print the first $i-1$ words neatly and then print the remaining words on a single line. The cost of this is $C[i-1]+c(i,j)$. From the Induction Hypothesis, we see that $c[i-1]$ is the optimal cost of neatly printing the first $i-1$ words. The remaining cost comes from the definition of the cost given in the problem. Thus $C[k+1]$ correctly represents the optimal cost of printing the first $k+1$ words.

Pseudocode

```
C[1,...,n] //Array to store printing cost
C[0] = 0

//Calculating c(i,j)
subsum[1,...,n] //subsum[i] will store sum of lengths of first i words
subsum[0] = 0
for i from 1 to n:
 subsum[i] = subsum[i-1] + l_i
end for

for i from 1 to n:
 for j from i to n:
```

```
    if (M − j + i − (subsum[j] − subsum[i − 1]) < 0):
     c(i, j) = ∞
    else if (j == n):
     c(i, j) = 0
    else:
     c(i, j) = M − j + i − (subsum[j] − subsum[i − 1])
    end if
   end for
  end for

  for j from 1 to n:
   C[j] = ∞
   for i from j to 1:
    if(C[j] > C[i − 1] + c(i, j)):
     C[j] = C[i − 1] + c(i, j)
    end if
   end for
  end for

  return C[n]
```

Running time:

To compute all possible values of $c(i, j)$, $O(n^2)$ time is required as i and j each vary from 1 to n. Each value of $C[j]$ takes $O(n)$ time to compute. To compute all values of $C[j]$ to obtain the final value of $C[n]$, $O(n^2)$ time is required. Therefore, the overall running time of the algorithm is $O(n^2)$.

6. (Dasgupta et al. textbook, Problem 6.6) Let us define a multiplication operation on three symbols a, b, c according to the following table; thus $ab = b, ba = c$, and so on. Notice that the multiplication operation defined by the table is neither associative nor commutative. Give a dynamic programming

	a	b	c
a	b	b	a
b	c	b	a
c	a	c	c

Table 1

algorithm that examines a string of these symbols, say *bbbbac*, and decides whether or not it is possible to parenthesize the string in such a way that the value of the resulting expression is a. For example, on input *bbbbac* your algorithm should return "Yes" because $((b(bb))(ba))c = a$. Please provide a proof of correctness and running time of your algorithm.

Solution:

A string $s[1, \ldots, n]$ can can be parenthesized to result in a value x if and only if it can be split into two parts $s[1, \ldots, k]$ and $s[k + 1, \ldots, n]$ such that $s[1, \ldots, k]$ can be parenthesized to give a value y and $s[k + 1, \ldots, n]$ can be parenthesized to give a value z such that $yz = x$. This observation shows that the problem has optimal substructure property. We use this property to design a DP algorithm to solve this problem.

Subproblem definition:

For all i, j such that $1 \leq i \leq j \leq n$, and all $x \in \{a, b, c\}$, let $P(i, j, x) = true$ if it is possible to parenthesize $S[i \ldots j]$ so that the value of the resulting expression is x. Otherwise, $P(i, j, x) = false$. We want $P(1, n, a)$.

Recurrence relation:

We obtain the following recurrence.

(a) If $i = j$: $P(i, j, x) = true$ if $S[i] = x$.

(b) If $i < j$: $P(i, j, x) = true$ if there exist k, y, z such that $i \leq k < j$ and $y, z \in a, b, c$ such that $P(i, k, y) = true$, $P(k + 1, j, z) = true$ and $yz = x$.

Proof of correctness:

The proof of correctness follows from the correctness of the recurrence.

Proof by induction

We have the following cases for calculating $P(i, j, x)$:

1. Base case: If $i = j$, then there is only one character, $S[i]$ to be considered. Therefore, $P(i, j, x) = true$ if $S[i] = x$, and $false$ otherwise. In general, $P(i, i, x) = true$ if $S[i] = x$ and $false$ otherwise for all values of i and x.

2. If $i < j$, i.e., if the substring $S[i \ldots j]$ is longer than 1 character, then $P(i, j, x) = true$ if $S[i \ldots j]$ can be split into two parts $S[i \ldots k]$ and $S[k + 1 \ldots j]$ such that $i \leq k < j$ and by multiplying the values of the expressions of the two parts, it is possible to get x, i.e., $P(i, k, y) = true$, $P(k + 1, j, z) = true$ and $yz = x$ for some values of y and z. Since the smaller subproblems are true by the Induction Hypothesis, our proof is complete.

Pseudocode

```
P(n, n, 3) //Matrix of size n × n × 3 to store length of longest palindrome
           //subsequence in substring x[i, ..., j]

for i from 1 to n:
 for j from 1 to n:
  for k from {a, b, c} :
   if(i == j and s[i] == k):
    P(i, j, k) = 1
   else:
    P(i, j, k) = −1    //-1 indicates Values are not set
   end if
  end for
 end for
end for

function true(i, j, x) //returns 1 if and only if the string s[i, ..., j] can be
 if(P(i, j, x) ≠ −1):  //parenthesized to give x
  return P(i, j, x)
 else:
  for k from i to j:
   for l from {a, b, c}:
```

```
        for m from {a, b, c}:
         if(true(i, k, l) × true(k + 1, j, m) == 1 and lm == a):
          P(i, j, x) = 1
         else:
          P(i, j, x) = 0
         end if
        end for
       end for
      end for
     end if

    return P(1, n, a)
```

Running time:

We calculate subproblems $P(i, j, x)$ for all values of i, j and x. The values of i and j can range from 1 to n, and there are 3 values of x which is a constant. As each subproblem can be calculated in $O(n)$ time, which is the maximum difference between the values of i and j, the overall running time of the algorithm is $O(n^3)$.

7. **Bonus (extra credit) question**
(Dasgupta et al. textbook, 6.14) *Cutting cloth.* You are given a rectangular piece of cloth with dimensions $X \times Y$, where X and Y are positive integers, and a list of n products that can be made using the cloth. For each product $i \in [1, \ldots, n]$ you know that a rectangle of cloth of dimensions $a_i \times b_i$ is needed and that the final selling price of the product is c_i. Assume the a_i, b_i, and c_i are all positive integers. You have a machine that can cut any rectangular piece of cloth into two pieces either horizontally or vertically. Design an algorithm that determines the best return on the $X \times Y$ piece of cloth, that is, a strategy for cutting the cloth so that the products made from the resulting pieces give the maximum sum of selling prices. You are free to make as many copies of a given product as you wish, or none if desired. Please provide a proof of correctness and running time of your algorithm.

Solution:

In the optimal solution to cut the cloth of dimensions $X \times Y$, we either make a vertical or horizontal cut initially or take the maximum profit from a product of dimensions $X \times Y$, if one exists. If a cut is made in the optimal solution, then the best return on the original cloth is the sum of the best returns from the two resulting pieces. Using this optimal substructure property, we can design a DP algorithm as follows. Given a cloth of size $X \times Y$, we can make a product of size $X \times Y$ if such a product exists, or we can make one of $X - 1$ possible vertical cuts, or one of $Y - 1$ possible vertical cuts. For each of the possible sizes of cloth thus derived, we need to repeat this procedure to find that series of cuts, and hence products to be manufactured, that gives us the maximum selling price of the overall cloth.

If multiple products exist with the same dimensions $i \times j$ or $j \times i$, then we only consider that product of these dimensions which has the maximum selling price.

Subproblem definition:

Let $P(i, j)$ indicate the maximum selling price of products that can be made from a piece of cloth of dimensions $i \times j$. We want $P(X, Y)$.

Let $p(i, j)$ indicate the price of the product with the highest selling price that can be made from a piece of cloth of dimensions $i \times j$ or $j \times i$, if it exists. Otherwise, $p(i, j) = 0$.

Recurrence relation:

We obtain the following recurrence.

$P(i,j) = \max\{\max_{1\leq a<i}\{P(a,j)+P(i-a,j)\}, \max_{1\leq b<j}\{P(i,j-b)+P(i,b)\}, p(i,j)\}$.

Proof of correctness:

The proof of correctness follows from the correctness of the recurrence.

Proof by induction

The base case is $P(0,0) = 0$.

For the Induction Hypothesis, assume that $P(i,j)$ gives the maximum profit obtainable from a cloth of dimensions $i \times j$ for all $i \leq k$ and for all $j \leq l$.
We then have the following options for computing $P(k+1,j), \forall j \leq l$:

1. $P(k+1,j)$ can be the price of the product with the highest selling price that can be made from a cloth of dimensions $k+1 \times j$ or $j \times k+1$, if such a product exists. Otherwise, $P(k+1,j)$ can be 0. This gives us the option $P(k+1,j) = p(k+1,j)$.

2. The cloth can be cut vertically in one of k places. We consider all possible cuts made at some position a, $1 \leq a < k+1$, to find the vertical cut that gives us the maximum selling prices for the two pieces obtained, $P(a,j)+P(k+1-a,j)$. These cases give the maximum possible value based on the Induction Hypothesis.

3. The cloth can be cut horizontally in one of $j-1$ places. We consider all possible cuts made at some position b, $1 \leq b < j$, to find the horizontal cut that gives us the maximum selling prices for the two pieces obtained, $P(k+1,j-b)+P(k+1,b)$. These cases give the maximum possible value based on the Induction Hypothesis.

Of these three options, we choose the one that results in the maximum selling price, i.e., the maximum value of $P(i,j)$.

The optimality of cases $P(i,l+1), \forall i \leq k$ and $P(k+1,l+1)$ is analogues to the one above.

This completes the proof.

Pseudocode

```
p(X,Y) //Matrix to store price of single cloth of dimensions i × j

P(X,Y) //Matrix to store maximum return from cloth of dimensions i × j

for i from 1 to X:
 for j from 1 to Y:
  p(i,j) = -1
 end for
end for

//filling in value of p(i,j)

for i from 1 to n:
 if (p(a_i,b_i) < c_i):
  p(a_i,b_i) = c_i
end for
```

```
for i from 1 to X:
 for j from 1 to Y:
  P(i, j) = -1
 end for
end for

P(0, 0) = 0

function maxreturn(i, j) //function to calculate maximum return
 if(P(i, j) ≠ −1):       //from a cloth of dimension i × j
  return P(i, j)
 else:
  P(i, j) = max{ max_{1≤a<i} {P(a, j) + P(i − a, j)}, max_{1≤b<j} {P(i, j − b) + P(i, b)}, p(i, j)}.
  return P(i, j)
 end if
return P(X, Y)
```

Running time:

There exist $O(X \cdot Y)$ subproblems of the type $P(i, j)$, as i and j range from 1 to X and 1 to Y respectively. Calculating all the values of $p(i, j)$ initially takes $O(X \cdot Y + n)$ time. Calculating the value of each subproblem $P(i, j)$ takes $O(X + Y)$ to find the maximum selling price over all possible cuts. Therefore, the overall running time of the algorithm is $O(X \cdot Y \cdot (X + Y) + n)$.