DYNAMIC PROGRAMMING

A CASE STUDY

Tony Hürlimann
Tony.huerlimann@unifr.ch

Working Paper

January 2005

DEPARTMENT D'INFORMATIQUE, UNIVERSITE DE FRIBOURG
DEPARTEMENT FÜR INFORMATIK, UNIVERSITÄT FREIBURG
Department of Informatics, University of Fribourg
Site Regina Mundi, rue de Faucigny 2, CH-1700 Fribourg / Switzerland

Dynamic Programming

A Case Study

Tony Hürlimann
Tony.huerlimann@unifr.ch

Summary

This paper exposes several modeling examples on how to implement efficiently problems which can be formulated as dynamic programming problems. Dynamic Programming is an algorithmic technique in which an *optimization problem* is solved by caching subproblem solutions (*memoization*) rather than recomputing them. Memoization is an algorithmic technique which saves (memoizes) a computed answer for later reuse, rather than recomputing the answer.

This two techniques together (breaking a problem in smaller subproblems and memoization) can be efficiently used to implement a large number of problems. By the means of several examples we show this ideas. We also show when the method breaks down and cannot be used anymore because of time or space complexity which cannot be overcome.

Introduction

If you don't do the best with what you have happened to have got, you will never do the best with what you should have had.
Aris Rutherford

Dynamic Programming refers to a very large class of algorithms. The idea is to break a large problem down (if possible) into incremental steps so that, at any given stage, optimal solutions are known to *sub-problems*. When the technique is applicable, this condition can be extended incrementally without having to alter previously computed optimal solutions to subproblems. Eventually the condition applies to all of the data and, if the formulation is correct, this together with the fact that nothing remains untreated gives the desired answer to the complete problem. Let us illustrate this by two examples.

The matrix-chain multiplication problem

Given a sequence of matrices such that any matrix may be multiplied by the previous matrix, find the best association such that the result is obtained with the minimum number of arithmetic operations.

The problem can be formulated as a recursive function in the following way: Let m(i,j) be the number of operations to compute a sequence of matrices at position i and j with i < j, then

$$m(i,j) = \begin{cases} 0 & i = j \\ \min\limits_{i \le k \le j} (m(i,k) + m(k+1,j) + p(i-1) \cdot p(k) \cdot p(j) & i < j \end{cases}$$

where p(i) is the row dimension of the (i+1)-th matrix in the sequence. (Note that the column dimension of the i-th matrix must be the same as the row dimension of the (i+1)-th matrix in order to be multipliable.) Multiplying two matrices with row/column dimensions p(1) x p(2) and p(2) x p(3) requires p(1) · p(2) · p(3) operations. Hence, the recursion says, that a sequence of matrices from position i to j can be calculated by splitting the sequence at any point k, with $i \le k \le j$. We then (1) multiply the sequence from i to k, which requires m(i,k) operations, and (2) multiply the sequence from k to j, which requires m(k,j) operations, and are then (3) left to multiply two matrices of size p(i-1) x p(k) and p(k) x p(j). We choose the k such that the number will be minimal.

Note that this recursion only calculates the number of minimal operations to be executed, not the actual sequence. However this can easily be done by tracing the k's that led to a minimal number.

Edit distance problem

Find the the minimum number of point mutations required to change a string s1 into another string s2, where a point mutation is: (1) changing a letter, (2) inserting a letter, or (3) deleting a letter.

The problem can be formulated as a recursive function in the following way. Let d(s1,s2) be the edit distance of two strings s1 and s2, then we have one of the three cases :

$$d('','') = 0$$
$$d(s,'') = d('',s) = |s|$$
$$d(s + c_1, t + c_2) = \min\left[d(s,t) + if\,(c_1 = c_2, 0, 1)\,,\, d(s + c_1, t) + 1\,,\, d(s, t + c_2) + 1\right]$$

The edit distance of two empty string is trivially zero.

The edit distance of a non-empty string *s* and the empty string is the length of the string *s*.

The edit distance between two non-empty strings *s* and *t* is either: (a) by adding/removing/changing a character (c1,c2) on both strings, if they are equal nothing changes, else the distance changes by one unit, or by adding/removing/changing a character from the first (b) or tzhe second (c) one of the two strings, then the distance changes by one unit. We take the minimium of the three possibilities.

[$if\,(a,b,c)$ means "if a is true then return b else return c".]

Note again that this function only returns the distance number, the actual transformation from one string to another is not computed. But this can easily be done by tracing the path following this computation.

The two problems have several elements in common:

1. The problem of a given size can be reduced to one or several problems of smaller size.
2. The smaller sized problems have the same "optimality" properties than the larger sized problem.
3. The problem of minimal size is trivially solvable.
4. The reduction step can be done by a "trivial" sequence of operations.

Factorial

As a small example, consider the calculation of the factorial. The calculation of the factorial of n can be (1) reduced to the calculation of the factorial of (n-1). (2) The smaller problem [(n-1)!] is calculated in the same way as is n! . (3) The problem of calculating the factorial of minimal size (n=1) is trivial, it is 1. (4) The reduction step is to multiply (n-1)! By n to obtain n!. Hence, one can write :

$$n! = \begin{cases} 1 & n = 1 \\ n \cdot (n-1)! & n > 1 \end{cases}$$

The nice property if such a function is that one can directly derive a recursive algorithm to compute the required quantities. A recursive function to calculate the factorial of a number is :

```
fact(n) {
    if n is 1 or less, then return 1
    else return n*fact(n-1);
 }
```

This function is efficient since it requires only n multiplication to calculate n!. It has even the special property to be tail-recursive. That is it does not require to store intermediate data between the recursive calls. This means that the function can be translated (even by a compiler) to a loop variant as follows :

```
fact1(n) {
    result is 1;
    for i=1 to n do result is i*result;
    return result;
 }
```

Fibonacci numbers

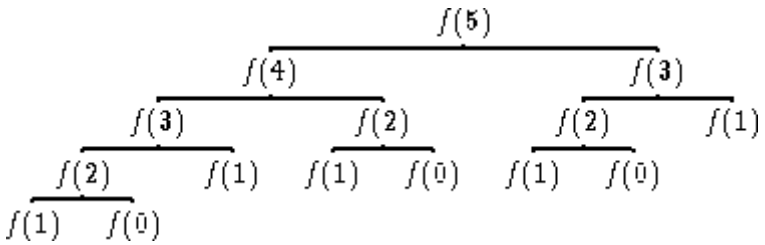
This is different with an other problem, the calculation of the n-th Fibonacci number. The Fibonacci number can be calculated by the following recursive function :

$$fib(n) = \begin{cases} 1 & n = 1,2 \\ fib(n-2) + fib(n-1) & n > 2 \end{cases}$$

A naive program to compute Fibonacci numbers is:

```
fib(n) {
    if n is 1 or 2 then, return 1
    else return fib(n-1) + fib(n-2);
 }
```

Because fib() is recomputed over and over for the same argument, run time for the above is $\Omega(1.6^n)$.



If instead we memoize (save) the value of fib(n) the first time we compute it, the run time is $\Theta(n)$.

```
 allocate array for memo;
 initialize memo[1] and memo[2] to 1, all others are 0;

fib_memo(n) {
   if memo[n] is zero then memo[n] = fib(n-1) + fib(n-2);
   return memo[n];
}
```

Of course, Fibonacci numbers can easily be computed in constant time (see [Fibonacci Site], an wonderful site), this only illustrates the technique.

This shows that the method of dynamic programming and its direct implementation as a recursive function is still useful for certain problems, even if the naïve recusive function has exponential complexity. With an additional (polynomial to the size of the problem) space, we can reduces the number of recursive calls to a polynomial number.

However, the scheme of memorizing of the recursive function requires quite a bit of reflection in general and it seems that it cannot be computed automatically. Often it is a difficult task to "distribute" the work between the *time* that the recursive function shall take to complete the computation and the *space* for memorizing. We have often to find a "good" balance between *computation time* and *memorizing space*.

Furthermore, in many real applications – we shall see one of them – either computation time or memory space or both grow very fast with the size of the problem, which means that only small instances can be solved using the method of dynamic programming even with memoization.

The 0-1 Knapsack Problem

The president of the Unites States was elected 1988 by 538 votes of lectors, who were distributed to the 51 states as following [Manber, ex 1.13]:

Alabama	9	Alaska	3	Arizona	7
Arkansas	6	California	47	Colorado	8
Connecticut	8	Delaware	3	Florida	21
Georgia	12	Hawaii	4	Idaho	4
Illinois	24	Indiana	12	Iowa	8
Kansas	7	Kentucky	9	Louisiana	10
Maine	4	Maryland	10	Massachusetts	13
Michigan	20	Minnesota	10	Mississippi	7
Missouri	11	Montana	4	Nebraska	5
Nevada	4	New Hampshire	4	New Jersey	16
New Mexico	5	New York	36	North Carolina	13
North Dakota	3	Ohio	23	Oklahoma	8
Oregon	7	Pennsylvania	25	Rhode Island	4
South Carolina	8	South Dakota	3	Tennessee	11
Texas	29	Utah	5	Vermont	3
Virginia	12	Washington	10	Washington D.C.	3
West Virginia	6	Wisconsin	11	Wyoming	3

Each state must give all votes a single canditate. Would it have been possible that both cantitates will get the same number of votes, namely 269?

To answer this question, we must make sure that there exists at least one subset of the 51 states for which the sum of all votes gives exactly 269. How to find this subset? We could enumerate all subsets and calculate the corresponding number for each subset. Unfortunately, there are 2^{51} subsets, too many to enumerate all.

This problem is an instance of the 0-1 Knapsack: Given n items with a weight w each, fill a bag in such a way that the total weight capacity K is exactly obtained. This problem has many applications, for example in the investment policy: Given an amount of money (the capacity), invest it in a subset of potential shares with a given price.

The problem could be formulated as an optimization problem: Let I be a set of items, let x_i be a 0-1 variable, which is 1 if the item i **is** in the Knapsack, and which is 0 if it **is not** in the Knapsack, let w_i be the weight of item i , and let K the required capacity, the the problem is :

$$\min \sum_{i \in I} x_i \quad \text{subject to} \quad \sum_{i \in I} w_i x_i = K \quad , x_i \in \{0,1\}$$

If the problem has a solution then it returns the required subset, otherwise it is infeasible. Solving it with the data above gives:

```
California        47
Illinois          24
Indiana           12
Massachusetts     13
Michigan          20
New Jersey        16
New York          36
North Carolina    13
Ohio              23
Pennsylvania      25
Texas             29
Wisconsin         11
--------------------
TOTAL            269
```

This is a solution with a minimal number of states (12). There are many other solutions with 13, 14, 15, 16, 17, … states. This can be seen by imposing an additional constraint: $\sum_{i \in I} x_i \geq a \;\; with \;\; a = 12, 13, 14, 15, ...$ showing the flexibility of such an approach. In LPL [virtual-optima] the 0-1 Knapsack problem can be coded as following :

```
model Knapsack01;
  set i;
  parameter w{i} := / … data are cut … /;
            K := 269; a:=12;
  binary variable x{i};
  minimize obj: SUM{i} x;
  constraint R: SUM{i} w*x = K;
              S: sum{i} x >= a;
end
```

The 0-1 Knapsack problem can also be formulated as a dynamic programming functuion. Let P(n,K) be a Boolean predicate that says, P(n,k) is true is there exists a Knapsack with n items and a capacity K, other wise it is false. We can consider exhaustively three cases as following:

$$P(n,K) = \begin{cases} true & \text{if } n = 0, K = 0 \\ false & \text{if } n = 0, K > 0 \\ P(n-1,K) \vee P(n-1,K-w_n) & \text{otherwise} \end{cases}$$

The Predicate P(0,0) says that there exists a Knapsack with zero items and capacity zero is trivially true.
The prediacte P(0,K) with K > 0, saying that there exists a Knapsack of capacity K > 0 with no item filled in is trivially false.
Considering the third case (with n > 0, K > 0), there exists a Knapsack P(n,K) (with n items and capacity K), if and only if there exists a Knapsack P(n-1,K) (with n-1 items and still capacity K) or there exists a Knapsack P(n-1,K-w_n) (that is, a Knapsack with n-1 items (by removing the n-th item) and capacity of K-w_n). In other words, if P(n-1,K) is true then also P(n,K) must be true. Why? Because we can take an n-th item and mark it "does not belong to the Knapsack". On the other hand, let's consider the case when P(n-1,K-w_n) is true, then we take an n-th item add it to the sample and mark it as "do belong to the Knapsack", this shows that if P(n-1,K-w_n) is true then also P(n,k) must be true.

A naïve implementation if the Knapsack predicate is as follows :

```
Knapsack(n,K) {
    if n is 0
       if K is 0 then return true else return false;
    else return Knapsack(n-1,K) or Knapsack(n-1,K-w[n]);
 }
```

The function is very similar to the recursive function for calculating Fibonacci numbers. It also has exponential complexity in the worst case. Again using a memo table of space |n·K| it is possible to reduce the worst case time complexity to a polynomial function in O(n·K). The corresponding program is as follows:

```
  allocate array T : size is nxK of type char;
  initialize all T[i,j] to '-';

Knapsack2(n,K) of boolean {
      Local vars are  sol1,sol2 of boolean;
  if T[n,K]=='-' then {
    if n==0 then {
      if K==0 then T[n,K]:='O' else T[n,K]:='x';
    } else {
      sol1=Knapsack2(n-1,K-w[n]);
      sol2=Knapsack2(n-1,K);
      if sol1 and sol2 then T[n,K]='B'
      else if sol1 and not sol2 then T[n,K]='I'
      else if not sol1 and sol2 then T[n,K]='O'
      else T[n,K]='x';
    }
  }
  return (T[n,K] is in ['O','I','B']);
}
```

This function finds *all* solutions. The table can have the following entries:

```
'-' : initialisation
'x' : Knapsack not possible
'I' : Knapsack possible when including the corresponding item
'O' : Knapsack possible by excluding the corresponding item
'B' : Knapsack possible with or without the corresponding item
```

The resulting table with K=8, n=4 and w=(2,3,5,6) is as follows:

```
K  0 1 2 3 4 = n
0  O O - - -
1  x - - - -
2  x I O O -
3  x x I - -
4  - - - - -
5  x x - - -
6  x - - - -
7  - - - - -
8  x x x I B
```

From the table it is possible to construct all solutions, in the following way: We begin with the Knapsack (n=4,K=8), the bottom right corner of the table. The entry says that two solutions are possible. One with the fourth item included and one without it. Let's follow the first path now: Item 4 with weight 6 is included. This leads us to a Knapsack with (n=3,K=2). The corresponding entry says that a Knapsack is only possible without the third item, so we remove it. This leads us to the Knapsack (n=2,K=2), again the entry says that a Knapsack is only possible without the second item, so we remove it too. We get the Knapsack (n=1,K=2), the entry says that a Knapsack is possible by including the first item. We get the Knapsack (n=0,K=0) and we are done with the solution: items 1 and 4: (2+6=8).

Following the second path now: Not including the fourth item leads us to the Knapsack (n=3,K=8). The entry says that a Knapsack is possible by adding the third item. We got the Knapsack (n=2,K=3). The entry says that an Knapsack only exists by adding the second item. We get to the Knapsack (n=1,K=0). The entry is 'O' so we remove the first item are are done since we got to (n=0,K=0). The solution is: items 2 and 3: (3+5=8).

We have constructed two solutions. There are no others as you can also say by inspection. At each entry in the table with a 'B' one has to follow two pathes, with 'O' and 'I' we only need to follow one path. This enumerates all possible Knapsack. We see from the table that "many" entries are still in a initialization state with '-'. This leads us to the idea that a smaller table would do it also. How small could it be? We could use a hash table. However statistical analysis of n<100 and K<1000 show that the table is filled up with a *constant* fraction of $|n \cdot K|$ entries, hence the space complexity is in the order of $O(|n \cdot K|)$.

However, if we are satisfied to find a single Knapsack solution then using a hash table of space $O(|n|)$ would do it. The corresponding function is as follows :

```
Knapsack21(n,K; var found:boolean):boolean {
      local vars are sol1,sol2 of boolean;
  if found then { return true; }
  if T[n,K]=='-' then {
    if n==0 then {
```

```
      if K==0 then { T[n,K]='O'; found=true; } else T[n,K]='x';
    } else {
      sol1=Knapsack21(n-1,K-w[n],found);
      sol2=Knapsack21(n-1,K,found);
      if sol1 and sol2 then T[n,K]='B'
      else if sol1 and not sol2 then T[n,K]='I'
      else if not sol1 and sol2 then T[n,K]='O'
      else T[n,K]='x';
    }
  }
  return (T[n,K] is in ['O','I','B']);
}
```

As before we initialize all entries of table T to '-' and set the global variable to found to false. Now the function return every time is found is true, no other Knapsack call are executed. The resulting table T now is as follows:

```
K  0 1 2 3 4 = n
0  O - - - -
1  - - - - -
2  x I O O -
3  - - - - -
4  - - - - -
5  - - - - -
6  - - - - -
7  - - - - -
8  - - - - B
```

An entry 'B' has to be interpreted as "follow both pathes but if you are stoke then try the other".

The Unconstrained Two-dimensional Guillotine Cutting Problem

The unconstrainted, two-dimensional cutting problem is the problem of cutting from a single plane rectangular piece a number of smaller rectangular pieces, of a given size and each with a given value, so as to maximize the value of the pieces cut (there being no constraint on the number of pieces of each size that result from the cutting) [Beasley 1985].

The problem appears in the cutting of steel or glass plates into required sizes, in the cutting of wood sheets to make furniture and in cutting of cardboard into boxes. Defining the value of a piece as its area converts the problem into the probem of minimizing the waste of material.

The restriction on guillotine cuts means that a cut on a rectangle must be a cut from one edge of the rctangle to the opposite edge which is parallel to the remaining edges. Hence the cuts must be done in stages, in which we have two cases:

1. First make cuts of the original plate horizontally into a certain number of smaller plates (first stage), then cut the remaining plates vertically into a certain number of still smaller plates (second stage), the remaining plates can now be cut horizontally again (third stage), etc. until *k*-stages cuts.
2. First make cuts of the original plate vertically into a certain number of smaller plates (first stage), then cut the remaining plates horizontally into a certain number of still smaller plates (second stage), the remaining plates can now be cut vertically again (third stage), etc. until *k*-stages cuts.

The two cases are displayed in Figure 1. No every cut can be done with guillotine staged cuts even if the numbr of stages *k* are extended to infinity. This can be shown by a simple example displayed in Figure 2. It is not possible to partition the rectagle in Figure 2 in its smaller plates by only using guillotine cuts.
However the guillotine cuts have important applications in which the material do not permit to cut-out rectangles from a larger one (glass for example). Guillotine cuts are essential in cutting such materials.

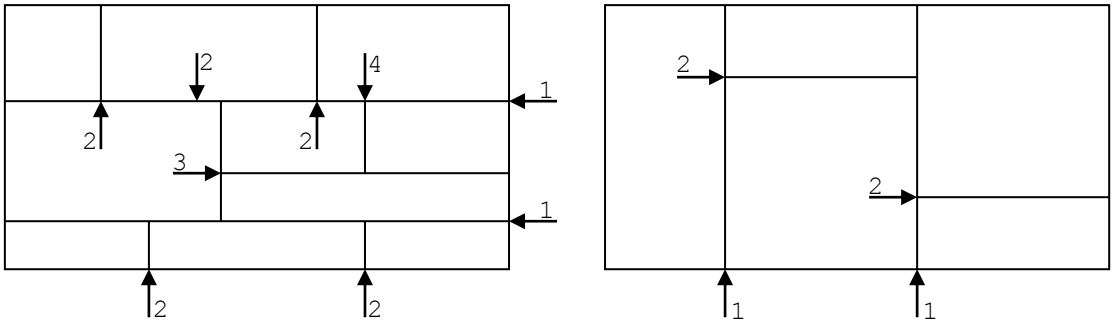


Figure 1: Stage guillotine cutting

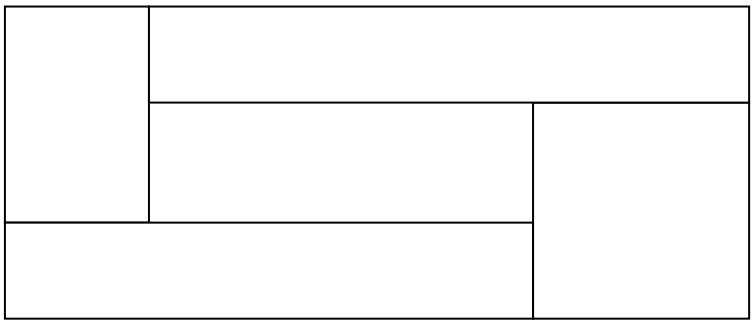


Figure 2: Guillotine cutting is not possible

To formulate a dynamic programming which can solve this problem, we introduce first some conventions:

(1) Let be the original plate of width w_0 and height h_0, and the m smaller plates to be cut of dimension w_i and height h_i with i = {1…m}.
(2) We note the dimension of rectangle i as (w_i , h_i).
(3) w is always meant to be the dimension in the horizontal direction (the x-axis) and h in the vertical direction (the y-axis).
(4) All number are integer and the cuts are infinitely thin.
(5) Let v_i the value of the i-th rectangle (we have $v_i = w_i \cdot h_i$ for minimizing the waste).
(6) We define the sets W = {1,…w_0-1} and H = {1,…h_0-1}, the sets of all possible lengths for any cuts parallel to the x-axis and the y-axis.

Let ***F(k,x,y)*** be the value of an optimal k-stage cut of a rectangle of size (x.y) where the first-stage cut direction is parallel to the x-axis; and similary, let ***G(k,x,y)*** be the value of an optimal k-stage cut of a rectangle of size (x.y) where the first-stage cut direction is parallel to the y-axis. Then we have the recursive functions

$$F(k,x,y)=\begin{cases}\max(0,v_i|w_i\le x,h_i\le y\,,i=1,\text{K}\,,m) & ,k=0\\ \max\left[F(0,x,y)\,,F(k,x,y-y_1)+F(k,x,y_1)\,,G(k-1,x,y)\right] & ,k>0\end{cases}$$
$$with\quad y_1\in H\,,\ y_1\le K_1(y)\quad and\quad K_1(y)=\lfloor y/2\rfloor$$

Similar for the function *G(k,x,y)* we have :

$$G(k,x,y)=\begin{cases}\max(0,v_i|w_i\le x,h_i\le y\,,i=1,\text{K}\,,m) & ,k=0\\ \max\left[G(0,x,y)\,,G(k,x_1,y)+G(k,x-x_1,y)\,,F(k-1,x,y)\right] & ,k>0\end{cases}$$
$$with\quad x_1\in W\,,\ x_1\le K_2(x)\quad and\quad K_2(x)=\lfloor x/2\rfloor$$

From the recursive functions we conclude that :

$$F(0,x,y)=G(0,x,y)$$

To find the optimal k-stage cutting independently of the beginning direction, we calculate:

$$v=\max\left[F(k,x,y)\,,G(k,x,y)\right]$$

An important reduction in the recursive calls can be done be redefining x_1 and y_1 as follows:

$$y_1,y_1^{'}\in HH,\quad HH=\left\{\sum_{i=1}^{m}h_ib_i,\quad 1\le y_1\le H-\min_{i\in\{1..m\}}h_i,\quad y_1+y_1^{'}\ne H,\quad b_i\ge 0\,(i\in\{1..m\})\right\}$$

and

$$x_1,x_1^{'}\in WW,\quad WW=\left\{\sum_{i=1}^{m}w_ia_i,\quad 1\le x_1\le W-\min w_i,\quad x_1+x_1^{'}\ne W,\quad a_i\ge 0\,(i\in\{1..m\})\right\}$$

To implement the function efficiently, we need a memorizing table of size O(k·W·H). With the reduction done, the table size can be reduced to O(k·|WW|·|HH|).

Now the implementation is as follows:

```
Let define four integer : kk,nn,ww,hh (kk: number of stages, nn: number
    of items, ww: width W, hh: height H
Let w and h be a array of nn integer to store width and height of the
    smaller rectangles
Let FF be an array of (2*kk+1)*ww*hh) integers to store the values

function F0(x,y:integer):integer; {
    define local integers  k,maxv,max,po;
```

```
  po=hh*(0*ww+x-1)+y-1;
  if FF[po]<>-1 then { max=FF[po]; maxv=h[max]*w[max]; }
  else {
    maxv:=0; max:=0; for k:=1 to nn do
      if (w[k]<=x) and (h[k]<=y) and (maxv<w[k]*h[k]) then {
        maxv=w[k]*h[k]; max:=k; }
    FF[po]=max;
  }
  p=max;
  return maxv;
}

function F(k,x,y:integer):integer; {
    define local integers i,v0,v1,v11,v12,v2,po
  po=hh*(k*ww+x-1)+y-1;
  if (k>0) and (FF[po]<>-1) then return FF[po];
  v0=F0(x,y,p00);
  if k=0 then return v0;
  v2:=G(k-1,x,y,p2);
  v1:=0;
  for i=1 to trunc(y/2) do {
    v11=F(k,x,i,p11);  v12=F(k,x,y-i,p12);
    if v11+v12>v1 then v1=v11+v12;
   }
   FF[po]=max(v0,v1,v2);
   Return v0;
}

function G(k,x,y:integer):integer; {
    define local integers i, v0,v1,v11,v12,v2,po
  po=kk*ww*hh + hh*(k*ww+x-1)+y-1;
  if (k>0) and (FF[po]<>-1) then return FF[po];
  v0=F0(x,y,p00);
  if k=0 then return v0;
  v2=F(k-1,x,y,p2);
  v1=0;
  for i=1 to trunc(x/2) do {
    v11=G(k,i,y,p11);  v12=G(k,x-i,y,p12);
    if v11+v12>v1 then v1=v11+v12;
  }
  FF[po]=max(v0,v1,v2);
  return v0;
}

  function FG(k,x,y:integer):integer; {
    return max(F(k,x,y) , G(k,x,y,p1));
  }
```

This implementation solves the unconstrained two-dimensional guillotine cutting problem. The actual cutting pattern is only a matter of an additional parameter to return. In this way, F(k,x,y) not only returns the optimal value but also the actual pattern. We define the cutting pattern as a string, expressing the structure of the pattern. The string has the format of a nested expression in the following way:

- [x,y,z] : the items x,y,a nd z are horizontally arranged from left to right.
- (a,b,c) : the items a, b, and c are vertically arranged from bottom to top.
- Each item can itself contain a nested list of rectangles.

Using this convention, we can now define how F(k,x,y) should return the format in string p:

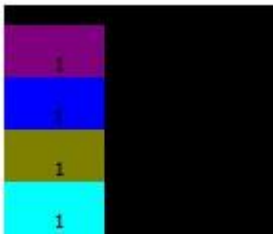
```
Let PP be an array of (2*kk+1)*ww*hh) strings to store the formats
```

```
function F(k,x,y:integer; inout p:string;):integer {
    define locals i,v0,v1,v11,v12,v2,p00,po:integer;
    and  p0,p1,p11,p12,p2:string;
  po=hh*(k*ww+x-1)+y-1;
  if (k>0) and (FF[po]<>-1) then { p=PP[po]; return FF[po]; }
  v0:=F0(x,y,p00);
  if k=0 then { p=IntToStr(p00); if p='0' then p=''; return v0; }
  v2=G(k-1,x,y,p2);
  v1=0;
  for i=1 to trunc(y/2) do {
    v11=F(k,x,i,p11);  v12=F(k,x,y-i,p12);
    if v11+v12>v1 then {
      v1=v11+v12;
      if (p11<>'') and (p11[1]=='(') then p11=copy(p11,2,length(p11)-2);
      if (p12<>'') and (p12[1]=='(') then p12=copy(p12,2,length(p12)-2);
      if p11=='' then p1:=p12 else
         if p12=='' then p1=p11 else p1=p11+','+p12;
    }
  }
  if v0>v1 then p=IntToStr(p00) else { v0=v1; p='('+p1+')'; }
  if v2>=v0 then { p=p2; v0=v2; }
  if p='0' then p='';  FF[po]=v0; PP[po]=p;
  return v0;
end;
```

The function F is almost the same as before. From the implementation point of view, this is an essential point in dynamic programming: It gives us a natural way to refine the program. This results in a very clear a verifiable code.

Example 1

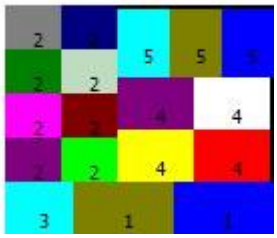
The data are : $m = 6, W = 67, H = 57, w = (25,14,17,19,13,15), h = (13,11,13,13,17,11)$



F(1,67,57) = 1300

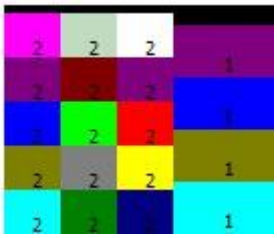
F(2,67,57) = 3718

F(3,67,57) = 3718



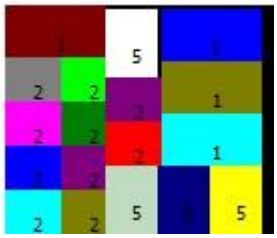
F(4,67,57) = 3754

G(1,67,57) = 1105



G(2,67,57) = 3610

G(3,67,57) = 3721



G(4,67,57) = 3724

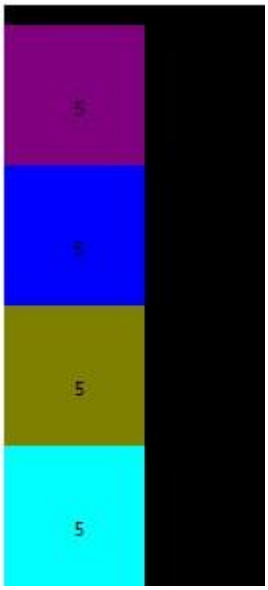
The resulting patterns (p) are:

```
F1 (1,1,1,1)
G1 [5,5,5,5,5]
F2 ([3,1,1],[3,1,1],[3,1,1],[5,5,5,5,5])
G2 [(2,2,2,2,2),(2,2,2,2,2),(2,2,2,2,2),(1,1,1,1)]
F3 ([3,1,1],[3,1,1],[3,1,1],[5,5,5,5,5])
G3 [(2,2,2,2,2),([3,3,4],[3,3,4],[3,3,4],[5,5,5,5])]
F4 ([3,1,1],[(2,2,2,2),(2,2,2,2),([4,4],[4,4],[5,5,5])])
G4 [([2,2],[2,2],[2,2],[2,2],1),([5,5,5],[(2,2,5),(1,1,1)])]
F5 ([3,1,1],[(2,2,2,2),(2,2,2,2),([4,4],[4,4],[5,5,5])])
G5 ([3,1,1],[(2,2,2,2),(2,2,2,2),([4,4],[4,4],[5,5,5])])
F6 ([3,1,1],[(2,2,2,2),(2,2,2,2),([4,4],[4,4],[5,5,5])])
G6 ([3,1,1],[(2,2,2,2),(2,2,2,2),([4,4],[4,4],[5,5,5])])
```

The best solution is generated by F(4,67,57) which is 3754. F(5,67,57) and F(6,67,57) produces exactly the same solution, and this is probably the optimal solution.

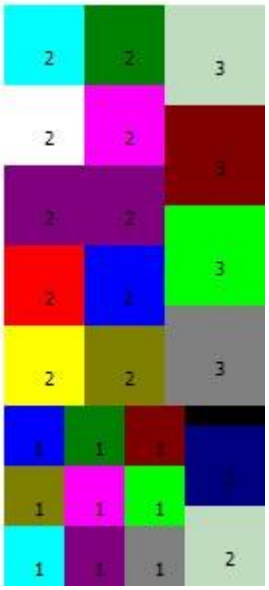
Example 2

The data are : $m = 5, W = 13, H = 29, w = (3,4,5,6,7), h = (3,4,5,6,7)$



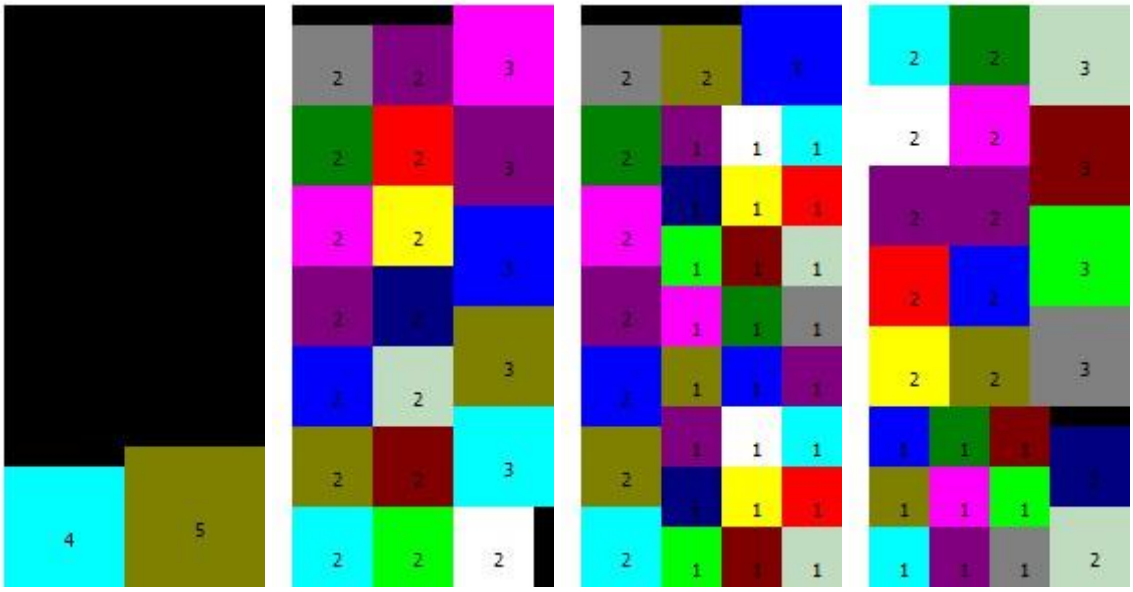
F(1,13,29) = 196

F(2,13,29) = 351



F(3,13,29) = 373

F(4,13,29) = 373



G(1,13,29) = 85 G(2,13,29) = 365 G(3,13,29) = 369 G(4,13,29) = 373

The resulting patterns (p) are:

```
F1 (5,5,5,5)
G1 [4,5]
F2 ([2,2,2],[2,2,2],[4,5],[4,5],[4,5])
G2 [(2,2,2,2,2,2,2),(2,2,2,2,2,2,2),(2,3,3,3,3,3)]
F3 ([(1,1,1),(1,1,1),(1,1,1),(2,2)],[(2,2,2,2,2),(2,2,2,2,2),(3,3,3,3)])
G3 [(2,2,2,2,2,2,2),
    ([1,1,1],[1,1,1],[1,1,1],[1,1,1],[1,1,1],[1,1,1],[1,1,1],[1,1,1],[2,3])]
F4 ([(2,2),([1,1,1],[2,3])],[(1,1,1,1,1,1,1),(1,1,1,1,1,1,1),(5,5,5)])
G4 ([(1,1,1),(1,1,1),(1,1,1),(2,2)],[(2,2,2,2,2),(2,2,2,2,2),(3,3,3,3)])
F5 ([(1,1,1),(1,1,1),(1,1,1),(2,2)],[(2,2,2,2,2),(2,2,2,2,2),(3,3,3,3)])
G5 ([(2,2),([1,1,1],[2,3])],[(1,1,1,1,1,1,1),(1,1,1,1,1,1,1),(5,5,5)])
F6 ([(2,2),([1,1,1],[2,3])],[(1,1,1,1,1,1,1),(1,1,1,1,1,1,1),(5,5,5)])
G6 ([(1,1,1),(1,1,1),(1,1,1),(2,2)],[(2,2,2,2,2),(2,2,2,2,2),(3,3,3,3)])
```

One meant think that the unconstraint problem is not so useful in practice. However, very often one can use this unconstraint problem as a sub-problem of real problems. An example is given in paper [Huer05a].

The Constrained Two-dimensional Guillotine Cutting Problem

The Constrained Two-dimensional Guillotine Cutting Problem is exactly the same problem as before, but with the additional restriction of a limited number b_i of each smaller rectangle. This problem is much more difficult to solve. In fact it cannot be solved using dynamic programming. At each table entry, we would need to store 2^n patterns and values which would explode the table. Hence, this problem must be solved using entirely different methods.

Conclusion

Dynamic programming is a powerful method to solve a large number of problems that can be formulated straightforward as recursive functions. Using memoization can limit the time complexity sometimes by using additional space. However to find a "good" mix between the time complexity and the additional space complexity requires fine tuning that cannot be left to an automatic procedure.

On the other hand, a large number of problems could be formulated using dynamic programming, but even adding memorizing does not remove the exploding complexity. Either the time or the space complexity become so large that they cannot be handled since there complexity is no longer polynomial in which case the method is nice to formulate the problem but it can not be used to be translated into a manageable program.

References

Beasley J.E., [1985], Algorithms for the Unconstrainted Two-Dimensional Guillotine Cutting, Journal of the Operational Research Society. Vol. 36, No. 4, pp. 297-306.

[Fibonacci Site] : http://www.mcs.surrey.ac.uk/Personal/R.Knott/Fibonacci/.

Hürlimann T., [2005], Two Cutting Stock Problems, Working paper, Departement of Informatics, University of Fribourg.

Manber U., [1989], Introduction to Algorithms: A Creative Approach, Wesley.

[virtual-optima]: http://www.virtual-optima.com