Dynamic Programming

Master MoSIG — Algorithms and Program Design

Florent Bouchez Tichadou*

November 4, 2013

1 Course

Objectives:

- Efficiently compute a recursive formula.

Dynamic programming is achieved while following different steps.

1. Write down the recursive formula corresponding to your needs. The formula might contain some conditionals.
2. Check the computations are finishing when using this formula (proof by induction).
3. Compute the complexity for the raw recursive computation. Are any function calls taking place twice ?
4. Write the function in your favorite programming language.
5. Add a table storing results and modify your program to avoid duplicate computations.
6. Write the dependency graph of the calls. Find vectors describing these dependencies. Follow these vectors to write a *sequential* program computing your function.
7. Optimize your sequential program for space.

Example

Naive recursive version

We consider as example the following formula: $f(x,y) = f(x-1,y) + f(x,y-1)$ with $f(0,y) = 1$ and $f(x,0) = 1$ which enables us to compute elements from the pascal triangle.
A possible C program to compute this formula is displayed on the right.

```
int f (int x, int y) {
  int result;
  if (x==0 || y==0)
    result=1;
  else
    result= f(x−1, y) + f(x, y−1);
  return result;
}
```

Figure 1 displays the call tree of function f for parameters 3,3. We can see that some calls are redundant and the computing cost of f as it is written now is exponential.

*Based on previous work by Mohammed Slim Bouguerra and Frédéric Wagner.

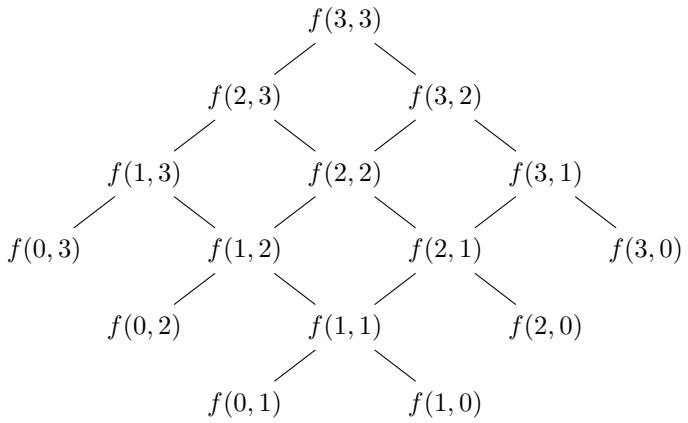


Figure 1: Call-tree for $f(3,3)$

```
1  int cache[N][N];
2  int f_optim (int x, int y) {
3    if (cache[x][y] != -1) return cache[x][y];
4    else {
5      int result;
6      if (x==0 || y==0)
7        result=1;
8      else
9        result= f(x-1, y) + f(x, y-1);
10     cache[x][y] = result;
11     return result;
12   }
13 }
```

Figure 2: Recursive version with caching

Optimized recursive version

In order to improve things we are going to remove redundant calculations by caching temporary results. To achieve this we allocate an array (the cache) big enough to contain all results for all different inputs of the f function. We then modify the function f in two steps: first we ensure that results are stored in the cache before all return points of the function and then we ensure that no duplicate computations take place by checking the content of the cache at entry point of the function. Of course the cache needs to be initialized with values which will never be mistaken with real results. In this example we can simply initialize all cache entries with -1 since any result will always be positive.

Figure 2 shows the modified f function with two modifications, at lines 3 and 10. Let us now we compute the cost of executing our algorithm with input (n_1, n_2). Clearly, this cost C is equal to $\sum_x \sum_y c(x,y)$ where $c(x,y)$ is the cost for computing the cache entry associated to x, y, for all reached values of x and y. In our case, $c(x,y) = O(1)$ since f contains no loop. This means that $C = O(1) \times \sum_x \sum_y 1 \leqslant (n_1+1) \times (n_2+1) \times O(1) = O(n_1 n_2)$.

Sequential version

Current cost is rather nice but we would like to optimize a bit more by getting rid of recursive calls. We start by making a small drawing of the cache together with the dependencies between the different results.

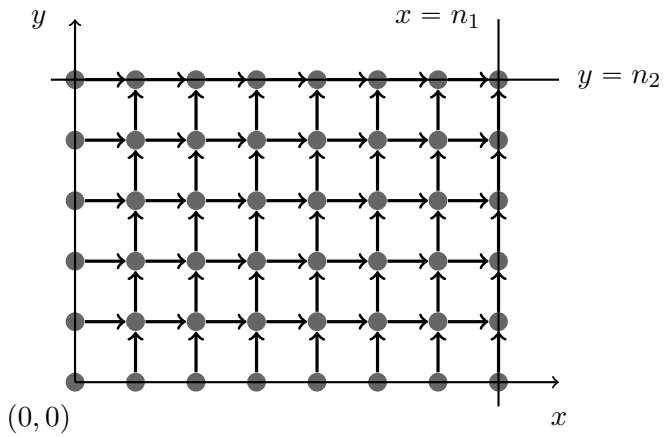


Figure 3: Dependence graph

```
10 int f_seq (int n1, int n2) {
11   for (int i = 0; i<=n1; i++)
12     for (int j = 0; j<=n2; j++) {
13       int result;
14       if (i==0 || j==0) result=1;
15       else result = cache[i-1][j] + cache[i][j-1];
16       cache[i][j] = result;
17     }
18   return cache[n1][n2];
19 }
```

Figure 4: Sequential code

As Figure 3 shows, dependencies follow two directions: we have vertical vectors and horizontal vectors. These dependencies mean that cache[i][j] cannot be computed before cache[i−1][j] and cache[i][j−1]. We choose as a basis of our space the two vectors $(0, 1)$ and $(1, 0)$ which translates into two nested loops, one loop on x and one loop on y. We have two possibilities because we can choose which one is the inner-most and which one is the outer-most loop. We choose here to iterate on y in the inner-loop.

Figure 4 shows the sequential code obtained. You can see that the body of the two nested loops is almost identical to the body of the recursive version of Figure 2. The only difference is that functions calls have been replaced by reading in the cache. Computing the cost of the sequential algorithm is direct ($O(n_1 n_2)$).

Memory optimization

The last step of optimisation which can be achieved is by reducing the amount of memory used. There is actually a simple way do do this, by using only one array of size the dimension of the inner loop (in our previous example, n_1, but we can chose the smallest dimension). For the sake of learning, we will impose the constraint of not iterating along the x or y axis.[1]

Figure 5 shows a decomposition of the cache into diagonal lines (12 in this particular example). It is clear than any cache value belonging to diagonal d_i can be computed if all values of diagonal d_{i-1} are available. Thus, if the outer-loop we choose iterates on the diagonals and the inner loop iterates inside each diagonal we can compute the final result while only storing two

[1] As a side note, what we are about to do is frequent when trying to parallelize a nested loop.

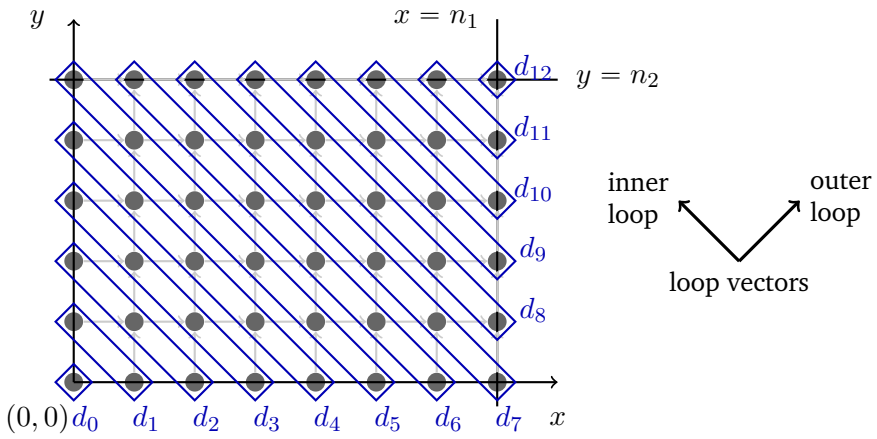


Figure 5: Iterating on diagonals

diagonals in memory. Thus, memory consumption can be reduced from $O(n^2)$ to $O(n)$. We can see from the figure that inner-loop vector will be $(-1, 1)$ while for outer loop vector it will be $(1, 1)$.

You can find below the source code for this last function. We decompose the iterations on the diagonals (on our example first from d_0 to d_7 and then from d_8 to d_{12}) into two loops. As you can see it is very difficult to write such a function without a drawing of the cache guiding your work.

```
int diagonal[2][N];
int f_horribilis (int n1, int n2) {
  int x, y, result;
  int cache = 0;          /* which diagonal to use: [0] or [1] */
  for (int d=0; d<=n1; d++, y=0) {
    for (int x=d; x>=0; x--, y++) {
      if (x==0 || y==0) result=1;
      else result = diagonal[1-cache][d-x-1] + diagonal[1-cache][d-x];
      diagonal[cache][d-x] = result;
    }
    cache = 1-cache;          /* switch cache between 0 and 1 */
  }
  for (int d=1; d<=n2; d++, x=n1) {
    for (int y=d; y<=n2; y++, x--) {
      if (x==0 || y==0) result=1;
      else result = diagonal[1-cache][y-d] + diagonal[1-cache][y-d+1];
      diagonal[cache][y-d] = result;
    }
    cache = 1-cache;
  }
  return diagonal[1-cache][0];
}
```

Exercises

Exercise 1 (Longest Common Subsequence (LCS))

This paragraph taken from the Wikipedia page:

> The longest common subsequence (LCS) problem is to find the longest subsequence common to all sequences in a set of sequences (often just two). (Note that a subsequence is different from a substring, for the terms of the former need not be consecutive terms of the original sequence.) It is a classic computer science problem, the basis of file comparison programs such as diff, and has applications in bioinformatics.

We will study here the problem of LCS on two sequences. Formally, the problem is defined as follows: Given two sequences $X = \langle x_1, x_2, \ldots, x_m \rangle$ and $Y = \langle y_1, y_2, \ldots, y_n \rangle$; then $Z = \langle z_1, z_2, \ldots, z_k \rangle$ is a common subsequence of X and Y if there exists two sequences of stricly increasing indices $\langle i_1, i_2, \ldots, i_k \rangle$ and $\langle j_1, j_2, \ldots, j_k \rangle$ such that for all $1 \leqslant l \leqslant k$, $z_l = x_{i_l} = y_{j_l}$. For instance $Z = \langle A, C, B, C \rangle$ is a common subsequence of $X = \langle \underline{A}, B, D, \underline{C}, A, \underline{B}, \underline{C} \rangle$ and $Y = \langle B, \underline{A}, B, \underline{C}, \underline{B}, D, \underline{C}, B \rangle$.

Given a common subsequence Z of X and Y, it is a **longest common subsequence** (LCS) if, for every common subsequence Z_α of X and Y, the length of Z is greater or equal to the length of Z_α. For instance, in the above example, Z is not a LCS as $\langle A, B, C, B, C \rangle$ is also a common subsequence of X and Y and of greater length.

Question 1.1 Given two sequences Z and X, write an algorithm that checks whether Z is a subsequence of X or not. What is its complexity?

Solution to 1.1: We use two pointers i and j that start respectively at x_1 and z_1. We scan Z with j, incrementing i whenever $z_i = x_j$. If i goes beyond the end of X, Z is not a subsequence of X, otherwise, it is.

```
1 i ← 1;
2 for j ← 1 to k do
3 |   while i ⩽ m and x_i ≠ z_j do
4 |   |   i ← i + 1;
5 |   if i > m then
6 |   |   return FALSE;
7 return TRUE;
```

Complexity: $O(\text{length}(X) + \text{length}(Z)) = O(\text{length}(X))$.

Question 1.2 How many subsequences of X are there? What would be the complexity of a LCS algorithms that checks for every subsequence of X if it is a subsequence of Y?

Solution to 1.2: If m is the length of X, there are 2^m subsequences as, for each element, you can choose whether to include it or not (note that the empty sequence and X itself are also subsequences). If n is the length of Y, then a brute-force algorithm would run in $O(n \times 2^m)$ complexity.

Optimal sub-structure (Warning: difficult questions)

A condition for dynamic programming to work is that the problem to solve has an **optimal sub-structure**, i.e., an optimal solution to the problem includes optimal solutions to its sub-problems.

Question 1.3 We want to characterize a LCS in terms of LCS of subproblems. Given a solution Z, state the conditions under which Z is a LCS of X and Y (based on the relations between Z and subproblems of LCS(X, Y)).

Solution to 1.3: We define $X_i = \langle x_1, x_2, \ldots, x_i \rangle$ and $Y_j = \langle y_1, y_2, \ldots, y_j \rangle$ for $0 \leqslant i \leqslant m$ and $0 \leqslant j \leqslant n$. Then Z is a LCS if:

1. if $x_m = y_n$, then $z_k = x_m = y_n$ and Z_{k-1} is a LCS of X_{m-1} and Y_{n-1};
2. if $x_m \neq y_n$, then $z_k \neq x_m \Rightarrow Z$ is a LCS of X_{m-1} and Y;
3. if $x_m \neq y_n$, then $z_k \neq y_n \Rightarrow Z$ is a LCS of X and Y_{n-1}.

Question 1.4 Prove the LCS has an optimal sub-structure (i.e., prove the conditions found in the previous question).

Solution to 1.4: We want to prove that an optimal solution contains optimal solutions to sub-problems. Let us consider an optimal solution $Z = \langle z_1, \ldots, z_k \rangle$.

- *Proof of case 1.* If $x_m = y_n$, then $z_k = x_m$, otherwise, we could build a longer solution $Z \cdot x_n$ (concatenation of Z and x_n). Now let us prove that Z_{k-1} is an optimal solution to the subproblem with X_{m-1} and Y_{n-1}. By the absurdity, suppose there is a solution S of greater length than Z_{k-1}, then $S \cdot x_m$ is a common subsequence of X and Y of greater length than Z, which contradicts the optimality of Z. □
- *Proof of case 2.* If $x_m \neq y_n$, then if $z_k \neq x_m$, Z is also a common subsequence of X_{m-1} and Y. Let S be an optimal solution to the LCS of X_{m-1} and Y. Since S is also a common subsequence of X and Y, it cannot be of greater length that the length of Z as the latter is optimal. Hence, Z is a LCS of X_{m-1} and Y. □
- *Proof of case 3.* This case is symetrical to case 2. □

Sub-problem superposition

A second condition for dynamic programming to work is that the sub-problems to a problem have **superposition**, i.e., a recursive algorithm will solve the same sub-problems again and again.[2] If the number of *different* sub-problems is polynomial in the size of the input, then dynamic programming can solve the problem in polynomial time, even though a direct recursive solution would require exponential time.

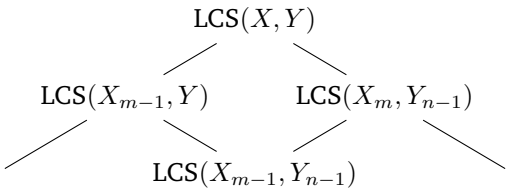
Question 1.5 Using the characterization of question 1.3, what would be the complexity of a direct recursive implementation for finding the LCS of X and Y? Prove that the LCS problem has *sub-problem superposition*.

Solution to 1.5: Let us define $C(m, n)$, the cost of finding a LCS two sequences of size m and n. In the worst case, at each step, the last elements of both sequences are different and we need to compute two subproblems:

$$\begin{aligned} C(0, n) &= O(1) \\ C(m, 0) &= O(1) \\ C(m, n) &= C(m-1, n) + C(m, n-1) + O(1) \\ &= O(2^{\max(m,n)}) \end{aligned}$$

[2] *A contrario*, in a divide-and-conquer algorithm, generated sub-problems are different.

As an example of subproblem superposition, we see that LCS(X_{m-1}, Y_{n-1}) can be called twice when computing LCS(X, Y) in the recursive call tree:



Note that we could also count the number of actually *different* subproblems, which are all the LCS(X_i, Y_j) for all $0 \leqslant i \leqslant m$ and $0 \leqslant j \leqslant n$. There are only a polynomial number of those (exactly $(m+1) \times (n+1)$) so many of the $O(2^{\max(m,n)})$ subproblems called recursively are equal.

Question 1.6 Propose a way to store the length of the LCS of sub-problems so that we do not need to re-compute them. In which order do you need to solve the sub-problems? Test it by computing the LCS of $\langle 1, 0, 1, 0, 0, 1, 0, 1\rangle$ and $\langle 0, 1, 1, 1, 1, 0, 1, 0\rangle$.

Solution to 1.6: We store the optimal length of the subproblems in an $(m + 1) \times (n + 1)$ matrix C indexed by i and j. Then, the cell $C_{i,j}$ contains the length of an LCS of X_i and Y_j. Note that this means $C_{m,n}$ contains the length of an optimal solution for X and Y.

Our dependency vectors are $(1, 1)$ (when the last elements are equal) and $(0, 1)$ and $(1, 0)$ (when we need to take the maximum of two subproblems). So we need to perform the computation in an order that respects those three vectors. We can solve the subproblems row by row or column by column, as long as the iteration variables are increasing (i.e., going left to right and top to bottom).

Here is now the computed matrix for the sequences in example. The arrows and grey and blue cells display information used for later questions.

	j	0	1	2	3	4	5	6	7	8
i		y_j	0	1	1	1	1	0	1	0
0	x_i	0	0	0	0	0	0	0	0	0
1	1	0	↓ 0	↘ 1	↘ 1	↘ 1	↘ 1	↓ 0	↘ 1	↓ 0
2	0	0	↘ 1	↓ 1	↓ 1	↓ 1	↓ 1	↘ 2	→ 2	↘ 2
3	1	0	↓ 1	↘ 2	↘ 2	↘ 2	↘ 2	↓ 2	↘ 3	→ 3
4	0	0	↘ 1	↓ 2	↓ 2	↓ 2	↓ 2	↘ 3	↓ 3	↘ 4
5	0	0	↘ 1	↓ 2	↓ 2	↓ 2	↓ 2	↘ 3	↓ 3	↘ 4
6	1	0	↓ 1	↘ 2	↘ 3	↘ 3	↘ 3	↓ 3	↘ 4	↓ 4
7	0	0	↘ 1	↓ 2	↓ 3	↓ 3	↓ 3	↘ 4	↓ 4	↘ 5
8	1	0	↓ 1	↘ 2	↘ 3	↘ 4	↘ 4	↓ 4	↘ 5	↓ 5

Solution:
$\langle 1, 1, 0, 1, 0\rangle$

Question 1.7 Write the corresponding algorithm for finding the LCS of two sequences.

Solution to 1.7: The algorithm is direct with the above answers, taking sub-solution $C_{i-1,j-1} + 1$ if $x_i = y_j$, or the max of $C_{i-1,j}$ and $C_{i,j-1}$ otherwise. Lines 7, 11, and 14 are answers to the next question.

```
1  for i ← 0 to m do C_{i,0} ← 0;
2  for j ← 1 to m do C_{j,0} ← 0;
3  for j ← 1 to m do
4      for i ← 1 to n do
5          if x_i = y_j then
6              C_{i,j} ← C_{i−1,j−1} + 1;
7              choice_{i,j} ← " ↘ ";
8          else
9              if C_{i−1,j} > C_{i,j−1} then
10                 C_{i,j} ← C_{i−1,j};
11                 choice_{i,j} ← " ↓ ";
12             else
13                 C_{i,j} ← C_{i,j−1};
14                 choice_{i,j} ← " → ";
15 return C_{m,n};
```

Question 1.8 With the previous question, we only know the maximum length of a common subsequence of X and Y. What else do you need to store in order to actually exhibit a LCS? Modify your algorithm so that it returns a LCS of X and Y.

Solution to 1.8: We need a second matrix, "choice," to store for each LCS problem the sub-problem that we "come from," i.e., the subsolution that we based our length calculation on. As there are only three possibilities, we need three different numbers, for instance 0, 1, and 2. Here, since we are writting pseudo-code we have the liberty to use arrows which are more graphical: $\downarrow$, $\rightarrow$, and $\searrow$. We store the direction we come from in lines 7, 11, and 14 of our algorithm. After all the computation are done, we can start at $i = m$ and $j = n$ and follow the arrows in the reverse direction to building the LCS (starting from the last element): each time we encounter $\searrow$, this means the element is part of the subsequence.

```
1  i ← m;
2  j ← n;
3  S ← ∅;
4  while i > 0 and j > 0 do
5      if choice_{i,j} =↘ then
6          S ← x_i · S;
7          i ← i − 1;
8          j ← j − 1;
9      else if choice_{i,j} =↓ then
10         i ← i − 1;
11     else
12         j ← j − 1;
13 return S;
```

The path taken by this algorithm is shown on the matrix: each blue cell means an element is added to S, and the grey cells are walked through without modifying S.

The complexity of building the subsequence is linear in $O(m + n)$ as i and j are only decreasing.

Question 1.9 Does every sub-problem needs to be computed? Propose a solution to compute only the required sub-problems. Does it change the complexity?

Try to use less memory. What can you still do and what can't you?

Solution to 1.9: By looking at the direction matrix, we can see that every ↖ "cuts" some part of the matrix (above or left) that is useless, i.e., recursing from LCS(X, Y), we would never compute these subproblems. These useless areas are marked in grey in the matrix below. Note however that with → or ↓, we *had* to compute the values in the cells from those two directions to know which one is the maximum.

With a recursive implementation, we compute only the subproblems that we need, but we still cache the results in the C and choice matrices. However, it doesn't change the worst-case complexity that stays in $O(m \times n)$.

	j	0	1	2	3	4	5	6	7	8
i		y_j	0	1	1	1	1	0	1	0
0	x_i	0	0	0	0	0	0	0	0	0
1	1	0	↓ 0	↘ 1	↘ 1	↘ 1	↘ 1	↓ 0	↘ 1	↓ 0
2	0	0	↘ 1	↓ 1	↓ 1	↓ 1	↓ 1	↘ 2	→ 2	↘ 2
3	1	0	↓ 1	↘ 2	↘ 2	↘ 2	↘ 2	↓ 2	↘ 3	→ 3
4	0	0	↘ 1	↓ 2	↓ 2	↓ 2	↓ 2	↘ 3	↓ 3	↘ 4
5	0	0	↘ 1	↓ 2	↓ 2	↓ 2	↓ 2	↘ 3	↓ 3	↘ 4
6	1	0	↓ 1	↘ 2	↘ 3	↘ 3	↘ 3	↓ 3	↘ 4	↓ 4
7	0	0	↘ 1	↓ 2	↓ 3	↓ 3	↓ 3	↘ 4	↓ 4	↘ 5
8	1	0	↓ 1	↘ 2	↘ 3	↘ 4	↘ 4	↓ 4	↘ 5	↓ 5

With the current implementation, we need two matrices of size $(m + 1) \times (n + 1)$. It is possible to not use the choice matrix, as C already has enough information to reconstruct the path: at point (i, j) we come from a diagonal if $x_i = y_j$, otherwise we can compare $C_{i-1,j}$ and $C_{i,j-1}$ and go to the bigger one. This adds a little more computation but does not change the complexity.

If we only need the length of the LCS, we do not need to store all the C matrix: we just need to keep in memory enough to make the next computations. We can keep in memory only two rows or two columns (using the smallest dimension). By being careful, we can even use only one row or column and an extra variable.

Exercise 2 (Selling apples)

An apple producer is selling its production of N tons to some K different resellers. The producer can decide the quantity of apples he is selling to each reseller but the resellers decide how much they pay for each ton they buy. The producer has some information on the price paid for each ton and for each reseller represented in an array P.

Write an optimized program computing the best choices for the producer.

Solution to 2: The idea is to recursively compute, for each $0 \leqslant i \leqslant N$ tons sold to the last reseller, the value of the best solution to the problem with $N - i$ tons sold to the first $K - 1$ resellers. The recursive function writes as follows:

$$
\begin{aligned}
&C(0, n) = 0 \\
&C(k, 0) = 0 \\
&C(k, n) = max_{0 \leqslant i \leqslant n} \left(C(k-1, n-i) + \sum_{j=1}^{i} P(k, j) \right)
\end{aligned}
$$

So we need as cache a matrix of size $(K + 1) \times (N + 1)$ to store the best values. To reconstruct the solution whenever we have $C(K, N)$, we need another matrix S of the same size and such that $S(k, i)$ is the number of tons sold to resellor k whenever we have i tons to sell to the k first resellers.

```
/* Iterative version, as every cell of C needs to be computed */
int sell_best (int K, int N)
{
  int i, n, k;

  for (k=0; k<=K; k++)
    C[k][0] = 0;
  for (n=0; n<=N; n++)
    C[0][n] = 0;

  for (k=1; k<=K; k++)
    for (n=1; n<=N; n++) {
      int i_best = 0;
      int v_best = 0;
      int current = 0; /* store the price paid by reseller k */

      /* Compute the max */
      for (i=0; i<=N; i++) {
        /* Update current total value sold to k */
        current += P[k][i];

        if (current + C[k-1][n-i] >= v_best) {
          v_best = current + C[k-1][n-i];
          i_best = i;
        }
      }

      /* Store the max */
      C[k][n] = v_best;
      S[k][n] = i_best;
    }
  return S[K][N];
}
```

```
/* Show the best solution based on C and S */
void _print_solution (int k, int n)
{
  print_solution (k-1, n-S[k][n]);
  printf ("%d ");
}

void print_solutIon (int K, int N)
{
  fpucts ("Solution: ", stdout);
  _print_solution (K, N);
  puts ("");
}
```

Exercise 3 (Pretty text printing)

You have a very old printer printing text with fixed width. You need to print some text currently encoded in only one line. You do not allow hyphenation, i.e., a word cannot be broken and printed on two different lines; You can however choose where you break each line. As such it is possible to carefully choose where to break the lines in order to maximize the beauty of the final document. We say a document is beautiful if all lines end very close from the end of the page and ugly if some lines end very far from the edge.

Question 3.1 Devise a greedy algorithm for text printing. Does it produces pretty printed text? Show that greedy is optimal in the number of lines used.

We need a metric on the notion of beauty. In TEX, D. Knuth chose the sum of the squared space left over at the end of every line. The goal is to minimize this sum.

Question 3.2 Why choose this metric over the simpler sum of space left?

We call C the number of characters that can fit in a line.

Question 3.3 Propose a formula that computes the optimal distance given a sentence to print.

Question 3.4 Use your formula to write a program that computes where to place the line breaks.

Exercise 4 (A caching problem)

We are interested in the optimization of a web proxy. The proxy works like a cache of html pages enabling to speed up the loading times when several requests to the same page take place. However, the proxy only has at his disposal a limited amount of space and only a small subset of all requested pages can be put into the cache. We consider the case where all requests by the clients are known in advance.

Simple case

We consider initially that all html pages take the same space and that the proxy is able to store 2 pages simultaneously. Pages are represented by letters $A, B, C, \ldots$ and the list of all pages requested is encoded as a string of characters. For example the string $ABCBB$ means that the proxy starts by delivering (and eventually caching) page A and then continues with page B and so on...

The behaviour of the proxy is the following:

- when a request is processed and the corresponding page is in cache, the proxy returns the page for free (no cost).
- when a request is processed and the corresponding page is not in cache the proxy downloads the page and stores it in the cache, eventually removing an other page from the cache. In this case, we count one cache miss.

The objective of this exercise is to design an algorithm enabling the proxy to choose which page to store or remove from the cache in order to minimize the total amount of cache misses.

We define a function f counting the minimal number of cache misses:

- $f([x_1, \ldots x_i][y, z])$ takes two arguments: the list of remaining requests : $[x_1, \ldots x_i]$ and the actual state of the cache: $[y, z]$
- f is a recursive function
- for example, with the cache containing pages A and C and requests BCB remaining, we have:
$$f(BCB, AC) = 1 + \min(f(CB, AC), f(CB, AB), f(CB, BC))$$

We name n the total number of requests and p the number of different pages.

1. give the definition of $f([x_1, \ldots x_i], [y, z])$.
2. prove that the recursive calls to f are ending.
3. draw the call-graph of f for the following case: $f(ABCBA, \varnothing)$. What is the result obtained ?
4. write a recursive program computing f. Your program will have a polynomial cost.
5. write a sequential program computing f.
6. what is the cost of your function ?
7. what is the cost in space of your function ?
8. is it possible to have a cost in space independant from n ?

Towards a more realistic problem

We note by l the size of the cache (l can now be more than 2).

1. what modifications should you apply to your algorithm ? (describe the main idea)
2. what become the costs in space and time of your algorithm ?
3. we suppose the pages can be of different sizes.
   (a) explain the modifications to apply to your algorithm.
   (b) give an upper bound on the time of execution.

Unused exercises

Exercise 5 (Distance between two strings)

In order to compare strings efficiently, we define a notion of distance between two strings as the minimum number of modifications needed to apply to string X to transform it into string Y. 3 types of modifications are possible:

- adding a letter α at beginning of string X (with cost add(α))
- removing a letter α from beginning of string X (with cost rem(α))
- changing the first letter of string Y (with cost chg(α, β)))

We define the distance function as follows, with α and β being different letters and ϵ denoting the empty string:

$$\text{dist}(\alpha X, \alpha Y) = \text{dist}(X, Y)$$

$$\text{dist}(\alpha X, \beta Y) = \min \begin{cases} \text{add}(\beta) & + & \text{dist}(\alpha X, Y) \\ \text{rem}(\alpha) & + & \text{dist}(X, \beta Y) \\ \text{chg}(\alpha, \beta) & + & \text{dist}(X, Y) \end{cases}$$

$$\text{dist}(\epsilon, \epsilon) = 0$$

1. write a recursive function computing dist
2. optimize this recursive function by adding a cache
3. write a sequential version
4. is it possible to optimize the memory usage ?

Exercise 6 (Backpack)

You have a backpack able to hold up to K kilograms without breaking. While coming back from holidays you intend to fill your bag with some items. Each item i has a weight w_i and some value v_i. You need to choose which items you take with you. You can take several times the same item but of course the total weight of the chosen items should be lower or equal to K.

Propose two different recursive formulas (recursing on the volume left and recursing on the number of items considered) solving this problem. Write the corresponding recursive and sequential algorithms. How do all of these algorithms compare ?