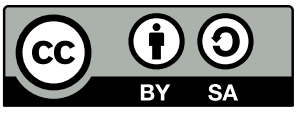
Dynamic Programming

Florent Bouchez Tichadou

October 2020



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 many calls are redundant. For instance $f(2,2)$ will be called twice, and $f(1,1)$ will be called 6 times! The computing cost of f as it is written now is exponential.

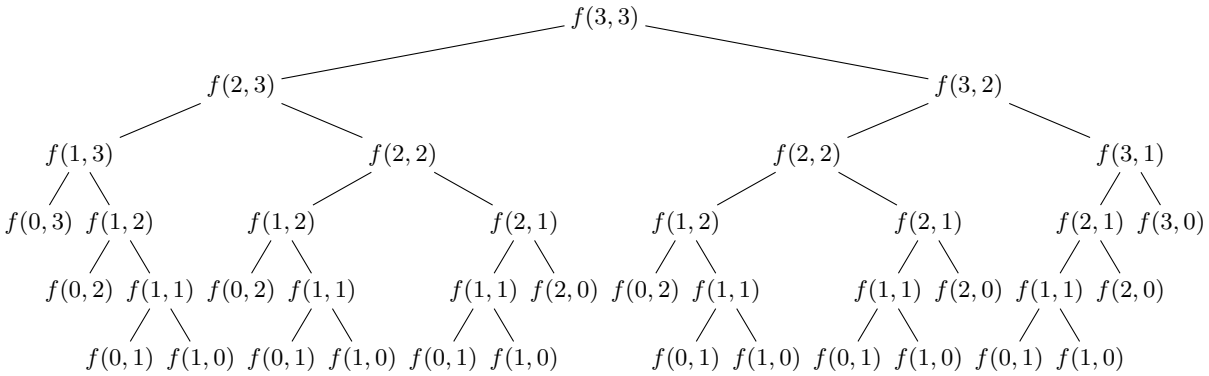


Figure 1: Full call-tree for $f(3,3)$

If we could magically "fold" the call tree, such that the same function call could be reached from two different places, we would obtain the representation on Figure 2, which is much more compact and now only has a polynomial amount of nodes. This means that if we manage to pass the "information" (the return value) of a function call to two (or more) different callers, we would now have a polynomial number of recursive calls.

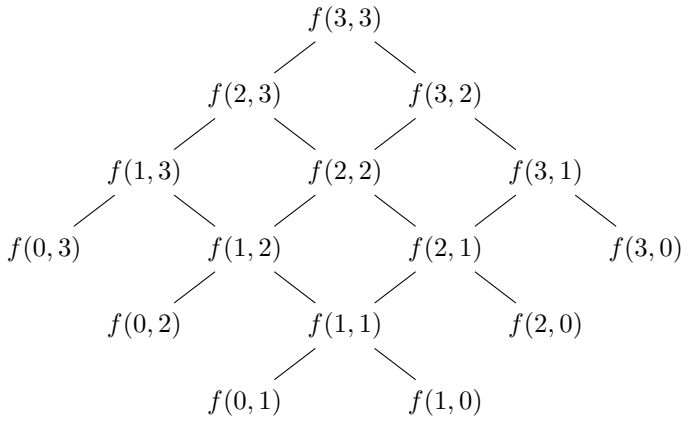


Figure 2: Folded call-tree for $f(3,3)$

Optimized recursive version

In order to simulate returning from a function call to multiple places, we will cache the results of the calls, hence remove redundant calculations: The first time a particular call is made, we actually compute the result, but before returning it we store it for later use. The next time a function call is made with the same parameters, we can retrieve the value already computed.

This method is applicable every time a big problem uses multiple times the results of **sub-problems**. In our case, the problem $f(x,y)$ has sub-problems $f(x-1,y)$ and $f(x,y-1)$.

To achieve this we allocate a cache (memory location) big enough to contain all results for all different inputs of the f function. In our case, since we have two parameters x and y bounded by 0 and n, a 2D-array of size $n+1 \times n+1$ is the logical choice. 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 computation takes place by checking the contents of the cache at entry point of the function. Of course the cache needs to be initialized

```
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 3: Recursive version with caching

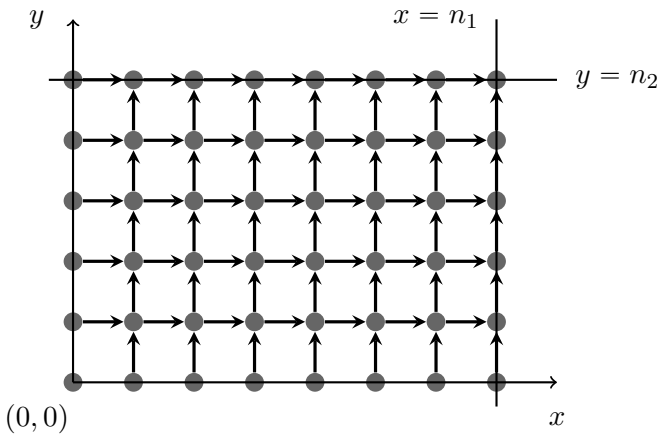


Figure 4: Dependence graph

with values which will never be mistaken with real results. In this example we can simply initialize all cache entries with -1 since all results will always be positive.

Figure 3 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). 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. We do not count in f the cost of the (eventual) recursive calls since it is already counted in the corresponding cache entries. This means that $C = O(1) \times \sum_x \sum_y O(1) \leqslant (n_1 + 1) \times (n_2 + 1) \times O(1) = O(n_1 n_2)$.

Note that, here, the same result is used at most 2 times, i.e., a constant number of times. For some algorithms, it can happen that the same function call is made n times; In that case, it should be taken into account when computing the complexity.

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.

As Figure 4 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 $(1, 0)$

```
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 5: Sequential code

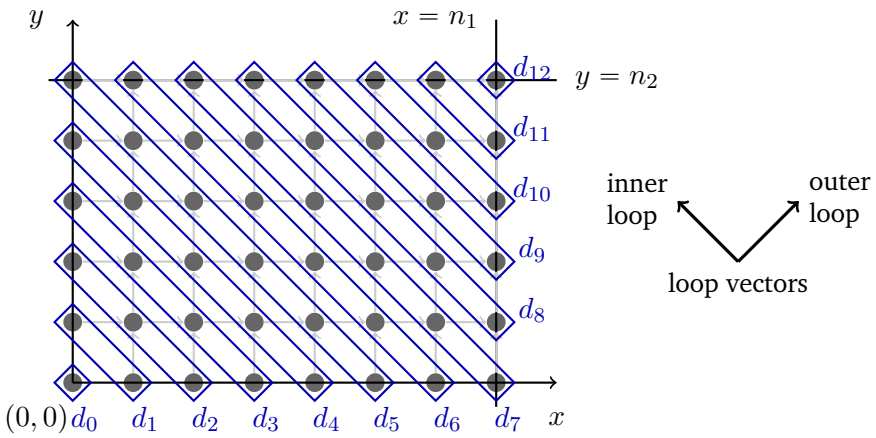


Figure 6: Iterating on diagonals

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 5 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 3. 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, which is currently in $O(n^2)$. 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 choose the smallest dimension). For the sake of learning, we will impose the constraint of not iterating along the x or y axis.[1]

Figure 6 shows a decomposition of the cache into diagonal lines (12 in this particular example). Seeing the dependency arrows, 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 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 $(-1, 1)$

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

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?

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?

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)).

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

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.

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

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*.

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 $\langle 0, 1, 1, 1, 1, 0, 1, 0\rangle$.

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

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.

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?

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.

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 ?