Tutorial 1 : Dynamic programming

Operations research, 3rd semester.

2024

Exercice 1 — *Starting exercise* Here is a recursive algorithm. Apply the dynamic programming method to this algorithm to produce a memoization version and an iterative version. What are the complexities of the 3 algorithms ?

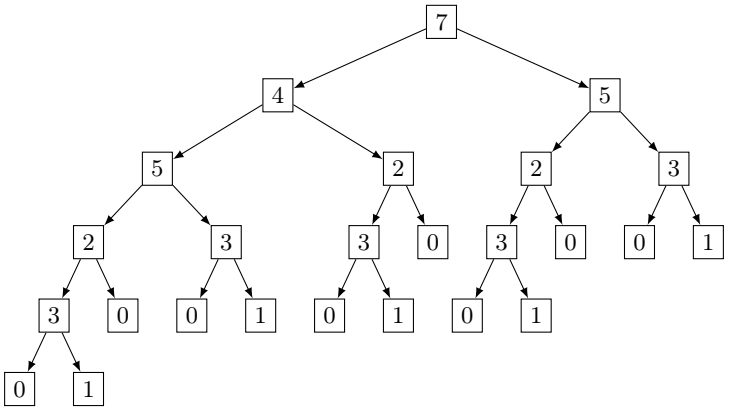
function F(n)
  Si $n \leq 1$ **Alors**
    Renvoyer 1
  Sinon Si n is even **Alors**
    Renvoyer F(n + 1) * F(n - 2)
  Sinon
    Renvoyer F(n - 3) + F(n - 2)

▶ **Correction**

The memoization version is

function FMEMO(n)
  Si $T[n]$ is empty **Alors**
    Si $n \leq 1$ **Alors**
      $T[n] \leftarrow 1$
    Sinon Si n is even **Alors**
      $T[n] \leftarrow$ FMEMO(n + 1) * FMEMO(n - 2)
    Sinon
      $T[n] \leftarrow$ FMEMO(n - 3) + FMEMO(n - 2)
  Renvoyer $T[n]$

For the iterative version, we need to find how to reverse the recursive calls. Let's do a little example with $n = 7$. We see that we would need to calculate the values of f in the following order : $1, 0, 3, 2, 5, 4, 7$ to obtain $f(7)$. More generally, we need to calculate them in the order $2p+1, 2p$ for all p.



The complexity of the algorithm given in the exercise is at worst of the order of $O(4^n)$. Each input will generate 2 recursive calls. And the value of n decreases by at least 1 every two recursive

```
function fIter(n)
    T[0] ← 1
    T[1] ← 1
    Pour p from 1 to ⌊n/2⌋ Faire
        T[2p + 1] ← T[2p − 2] + T[2p − 1]
        T[2p] ← T[2p + 1] ∗ T[2p − 2]
```

calls (in other words, for each value y such that $f(y)$ is called by $f(n)$, and for each z such that $f(z)$ is called by $f(y)$, we have $z \leq n-1$). Thus, the tree of recursive calls is a binary tree where the number written in each node decreases by at least 1 when descending 2 levels. Since f stops when n is less than 1, the height of the tree cannot exceed $2n$. Therefore, there are at worst $2^{2n} = 4^n$ nodes in the tree. The complexity of the memoization algorithm is $O(n)$ because there are at most $n + 1$ entries that can be used for f, and during each recursive call, there is at most a constant number of operations. The complexity of the iterative algorithm is also $O(n)$ because there is a loop with $\lfloor n/2 \rfloor$ iterations containing a constant number of operations per iteration.

Exercice 2 — *Pocket money*

A student wants to earn some money when she is not at the university. She can work at most T hours by week so that she has time left to study. She found n possible jobs and it is possible for her to divide her time into multiple jobs (for example, she can work two hours for the first job, ten for the third one, ...). The following table gives the reward $s(j, i)$ depending on how many hours j per week she decides to work for each job i, where $T = n = 4$.

Nb hours of work \ Job	Job I	Job II	Job III	Job IV
0	0	0	0	0
1	26	23	16	19
2	39	36	32	36
3	48	44	48	47
4	54	49	64	56

1. Using a naive recursive algorithm, compute how much money she can earn at most.
2. Deduce a memoization dynamic programming algorithm to solve the problem. Use this algorithm on the example.
3. Write the iterative version of the dynamic programming algorithm algorithm. Use this algorithm on the example.
4. What are the complexities of the three algorithms ?
5. How could we compute the number of hours the student has to work for each job to obtain the optimal reward ?

▶ **Correction**

An optimal solution of this instance should be 85 : 2 hours IV, 1 hour II and 1 hour I.

1. A naive algorithm could try to brute force enumerate the possible solutions but we are asked for a recursive algorithm.

   We set $\textsc{Pocket}(h, i)$ as the maximum reward that can be own by working h hours for the jobs 1 to i. The initial problem consists in finding $\textsc{Pocket}(T, n)$. We can initialize $\textsc{Pocket}(h, 1) = s(h, 1)$ for every h. We can also set $\textsc{Pocket}(0, i) = 0$, but this is optional. Pocket can be computed with the following formula :

   $$\textsc{Pocket}(h, i) = \max_{k=0..h} \textsc{Pocket}(k, i-1) + s(h-k, i)$$

   We may interpret it this way : if I work h hours for the jobs 1 to i. Then, there necessarily exists $k \leq h$ such that I work k hours for the jobs 1 to $i-1$ and $(h-k)$ hours for the job i. I just need to find the optimal k.

   ```
   1: function Pocket(h, i)
   ```

```
2:    Si i = 1 Alors
3:        Renvoyer s(h, i)
4:    Si h = 0 Alors
5:        Renvoyer 0
6:    Renvoyer max_{k=0..h} Pocket(k, i − 1) + s(h − k, i)
```

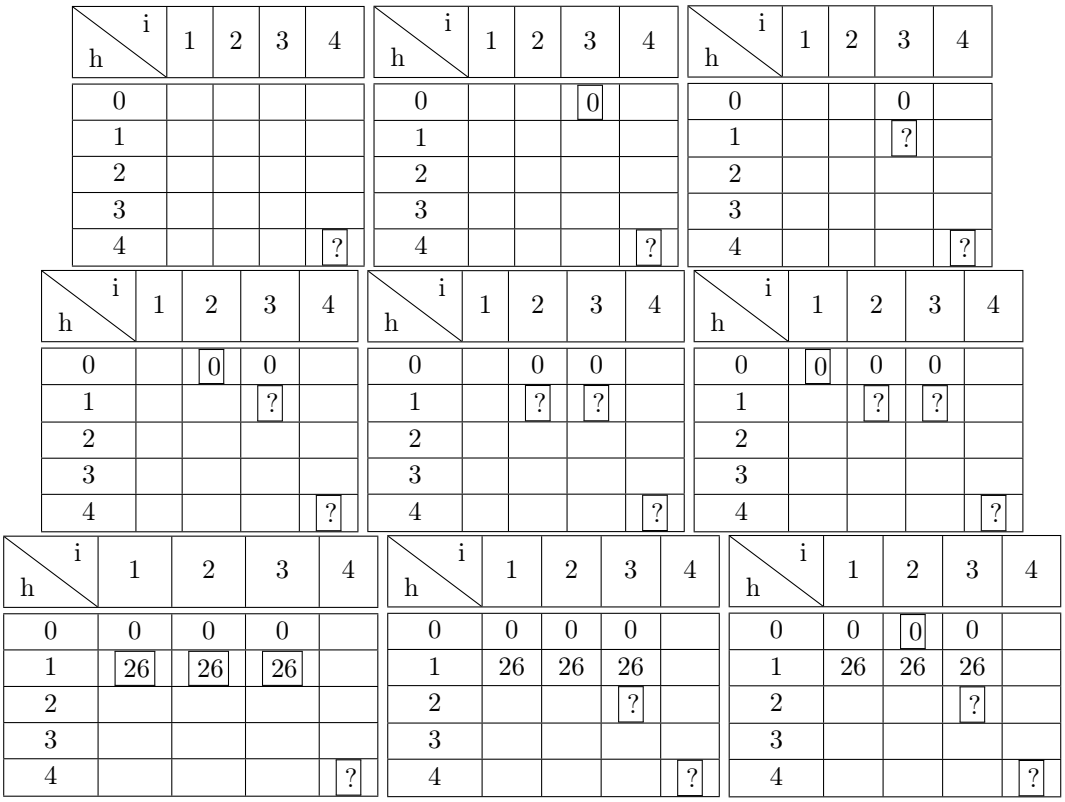
2. We can then use the memoization technique by using an auxilliary table $Ar : [\![0; T]\!] \times [\![1; n]\!] \to \mathbb{N}$. (I did not call this table T as the variable T is already used.)

   This array is initially empty.

```
1: function Pocket_Memo(h, i)
2:    Si Ar(h, i) is empty Alors
3:        Si i = 1 Alors
4:            Ar(h, i) ← s(h, i)
5:        Sinon Si h = 0 Alors
6:            Ar(h, i) ← 0
7:        Sinon
8:            Ar(h, i) ← max_{k=0..h} Pocket_Memo(k, i − 1) + s(h − k, i)
9:    Renvoyer Ar(h, i)
```

   The next illustration is the execution of the algorithm on the example. We indicate the values of Ar, h and i for every recursive call.

   The beggining is detailed, but not the end. Do not copy 15 times the array on your sheet of paper, you can edit dynamically the array.



h \ i	1	2	3	4
0	0	0	0	
1	26	[26]	26	
2			[?]	
3				
4				[?]

h \ i	1	2	3	4
0	0	0	0	
1	26	26	26	
2		[?]	[?]	
3				
4				[?]

h \ i	1	2	3	4
0	[0]	0	0	
1	26	26	26	
2		[?]	[?]	
3				
4				[?]

h \ i	1	2	3	4
0	0	0	0	
1	[26]	26	26	
2		[?]	[?]	
3				
4				[?]

h \ i	1	2	3	4
0	0	0	0	
1	26	26	26	
2	[39]	[49]	[49]	
3				
4				[?]

h \ i	1	2	3	4
0	0	0	0	
1	26	26	26	
2	39	49	49	
3			[?]	
4				[?]

After some more iterations we get the following result.

h \ i	1	2	3	4
0	0	0	0	
1	26	26	26	
2	39	49	49	
3	48	62	65	
4	54	75	81	85

We see that there is no need to compute the fourth column entirely.

3. To trace back the recursive calls, we can observe that : POCKET(h, i) calls POCKET$(k, i-1)$ for all $k \leq h$.

   So if we have computed POCKET$(k, i-1)$ for all $k \leq T$, then we can compute POCKET(h, i) for all $h \leq T$.

   The base cases are when $h = 0$ and when $i = 1$.

   1: **function** POCKET$_{Iter}$(T, n)
   2: **Pour** i from 1 to n **Faire**
   3: $Ar(0, i) \leftarrow 0$
   4: **Pour** h from 0 to T **Faire**
   5: $Ar(h, 1) \leftarrow s(h, 1)$
   6: **Pour** i from 1 to n **Faire**
   7: **Pour** h from 1 to T **Faire**
   8: $Ar(h, i) \leftarrow \max_{k=0..h} Ar(k, i-1) + s(h-k, i)$
   9: **Renvoyer** $Ar(T, n)$

   On the example we get the following results

   After lines 2 and 3 :

h \ i	1	2	3	4
0	0	0	0	0
1				
2				
3				
4				

After lines 4 and 5

h \ i	1	2	3	4
0	0	0	0	0
1	26			
2	39			
3	48			
4	54			

After lines 6, 7 and 8

h \ i	1	2	3	4
0	0	0	0	0
1	26	26	26	26
2	39	49	49	45
3	48	62	65	68
4	54	75	81	85

4. Let's denote the time complexity of the algorithm from question 1 by $t_1(T, n)$.
We know that $t_1(h, i) = \sum_{k \leq h} t_1(k, i-1)$ and that $t_1(h, 1)$ takes constant time. We will set $t_1(h, 1) = 1$.
Thus, we have, for example :

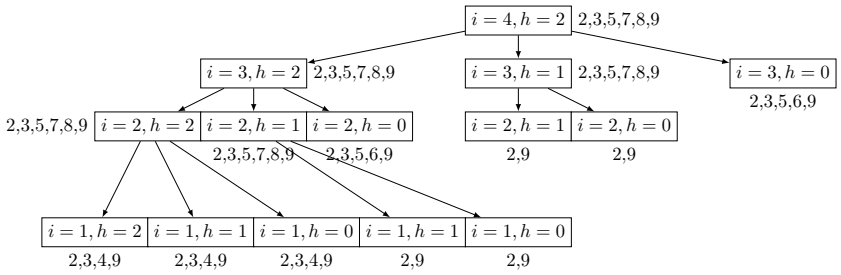
$$t_1(h,1) = 1$$
$$t_1(h,2) = h$$
$$t_1(h,3) = \frac{h \cdot (h+1)}{2}$$
$$\dots$$

We will show that $t_1(h, i) = O(h^{i-1})$ by induction. This is true for $i = 1$, and if it is true for $i - 1$, then by induction, $t_1(h, i) = \sum_{k \leq h} O(k^{i-2}) = O(h^{i-1})$.
(Strictly speaking, we should be a bit careful with these O notations and develop them with their constants, but the proof is somewhat complex (without being horrible) ; I invite you to check)
`http://cm2.ens.fr/sites/default/files/summae_potestatum.pdf`

For the memoization version :
We execute lines 3 to 8 only once for each (h, i), for $2 \leq i \leq n$ and $1 \leq h \leq T$, because on subsequent calls $Ar(h, i)$ is non-empty and the **If** at line 2 is false. Thus, we execute lines 3 to 8 at most $T \cdot n$ times. Therefore, we execute line 8 at most $T \cdot n$ times with its $0 \leq h \leq T$ recursive calls. There are thus at most $T^2 \cdot n$ recursive calls to POCKET_{Memo}.
Short Proof : Among these calls, there are at most $T \cdot n$ calls executing lines 3 to 8, taking time $O(T)$, and at most $T^2 \cdot n$ calls that do not execute them, taking time $O(1)$. Thus, we have a complexity of the order of $O(T^2 \cdot n + T^2 \cdot n) = O(T^2 \cdot n)$.
Less Short Proof : One might wonder in the above version if we forget to account for some computational time. Why would the calls executing lines 3 to 8 take $O(T)$ time ? Indeed, at

line 8 we make recursive calls. So it takes more time than $O(T)$ to compute the max. This is quite true, but this time is taken into account with the other recursive calls.

Here is a tree representing the different calls when $n = 4$ and $T = 2$.



In this example, there are 13 recursive calls. Next to each call, we have written the lines of the algorithm that are executed during the algorithm. The complexity is the number of instructions in the program.

Note : We could count the number of lines of the algorithm in the drawing, which is 60, as the complexity but that would overlook the fact that, once we have all the results from the recursive calls, line 8 is not executed in constant time but in time $h + 1$ (there are $h + 1$ instructions during the call to line 8 due to the max). Therefore, we need to count the number of instructions other than line 8 plus the number of line 8 times $h + 1$. Since $h \leq T$, in the worst case, that's $55 + 5(T + 1) = 70$ instructions.

How can we calculate this value for any value of T and n ? We can define two values associated with each node u of the tree.

$T(u)$ is the number of instructions executed during the call of u, that is to say, the set of instructions associated with the entire subtree rooted at u. For example, for the root node, it is 70 as explained above. For the leaf $i = 1, h = 0$ at the bottom beneath the root, it is 2. For the node $i = 2, h = 2$ on the left, it is 17 instructions other than line 8, plus $h+1$, which is 20.

$S(u)$ is the set of instructions associated with a node (and not its descendants) : 2 for the node at the bottom right, 5 for the node $i = 3, h = 0$ on the right, 8 for the root node (5 instructions excluding line 8 + 3 instructions for line 8). We verify that $T(u) = \sum_{v \geq u} S(v)$ where $v \geq u$ means that v is a descendant of u in the tree (including u).

How can we count $S(u)$ in practice ? We simply need to calculate the complexity of the algorithm assuming that the recursive calls are made in constant time. This complexity therefore does not take into account the complexity of the descendant recursive calls.

The complexity of the algorithm is $T(R)$ where R is the root of the tree. Therefore, we are looking for the sum of the $S(v)$ for $v \geq R$. We just need to count the number of nodes in the tree and upper bound $S(u)$ for each node.

As explained above, there are at most $T^2 \cdot n$ recursive calls, so as many nodes in the tree. Among these nodes, there are at most $O(T \cdot n)$ that will execute lines 3 to 8. All other nodes will only execute lines 2 and 9. For the latter, we have $S(u) = O(1)$. For the former, we have $S(u) = O(T)$.

Thus, $T(R) = O(T^2 \cdot n + T^2 \cdot n)$.

For the Iterative version :

The loop on lines 2-3 runs in time $O(n)$. The loop on lines 4-5 runs in time $O(n)$. The loop on lines 6-8 runs in time $O(T^2 \cdot n)$.

So the complexity is also $O(T^2 \cdot n)$.

5. In addition to calculating the remuneration, we will keep track of information in a second table regarding the solution associated with the optimal gain.

   We create a second table Ar_2. In the cell $Ar_2(h, i)$, we record the entire solution associated with $Ar(h, i)$ (though there are other methods that are less memory-intensive).

We initialize Ar_2 trivially with $Ar_2(h,1) = (h \to 1)$ for all h (which means we work h hours in job 1) and $Ar_2(0,i) = (0 \to 1, 0 \to 2, \ldots, 0 \to i)$ for all i (which means we work 0 hours in all jobs).

If $k^* = \arg\max_{k=0..i} Ar(k, i-1) + s(h-k, i)$, then k^* is the number of hours to work in jobs 1 to $i-1$ to maximize the money earned when working h hours in jobs 1 to i. Therefore, $Ar_2(h,i) = Ar_2(k^*, i-1) \cup (h - k^* \to i)$; which means that the solution associated with $Ar(h,i)$ consists of assigning $h - k^*$ hours to job i and then assigning the same hours to the other jobs as in the solution associated with $Ar(k^*, i-1)$.

We have then the following algorithms

```
1: function POCKET_Memo(h, i)
2:     Si Ar(h,i) est vide Alors
3:         Si i = 1 Alors
4:             Ar(h,i) ← s(h,i)
5:             Ar2(h,i) ← (h → i)
6:         Sinon Si h = 0 Alors
7:             Ar(h,i) ← 0
8:             Ar2(h,i) ← (0 → 1, 0 → 2, ..., 0 → i)
9:         Sinon
10:            k* ← arg max_{k=0..h} POCKET_Memo(k, i-1) + s(h-k, i)
11:            Ar(h,i) ← Ar(k*, i-1) + s(h-k*, i)
12:            Ar2(h,i) ← Ar2(k*, i-1) ∪ (h-k* → i)
13:    Renvoyer Ar(h,i)
```

```
1: function POCKET_Iter(T, n)
2:     Pour i de 1 à n Faire
3:         Ar(0,i) ← 0
4:         Ar2(h,i) ← (0 → 1, 0 → 2, ..., 0 → i)
5:     Pour h de 0 à T Faire
6:         Ar(h,1) ← s(h,1)
7:         Ar2(h,i) ← (h → i)
8:     Pour i de 1 à n Faire
9:         Pour h de 1 à T Faire
10:            k* ← arg max_{k=0..h} Ar(k, i-1) + s(h-k, i)
11:            Ar(h,i) ← Ar(k*, i-1) + s(h-k*, i)
12:            Ar2(h,i) ← Ar2(k*, i-1) ∪ (h-k* → i)
13:    Renvoyer Ar(T,n)
```

Exercice 3 — *The knapsack problem*

This is a classical problem of Operations Research.

Given :

- a bag of volume V,
- n type of items, numbered from 1 to n
- there are m_i items of type i
- each item of type i has a volume v_i
- each item of type i has a value u_i

we want to fill the bag maximizing the total value of the objects we put inside.

We are going to solve this problem using dynamic programming. Let x_i be the number of times the object i is put in the bag.

1. What is the value $U(x_1, x_2, \ldots, x_n)$ of the bag ?

2. Rewrite the volume constraint as $V(x_1, x_2, \ldots, x_n) \leq V$.
   Let $f(j, b)$ be the maximum value of a bag of volume b using only the first j types of objects, for $b \in [\![0, V]\!]$ and $i \in [\![1; n]\!]$, *i.e.* the maximum value of $U(x_1, \ldots, x_j)$ such that $V(x_1, \ldots, x_j) \leq b$.
3. How computing $f(j, b)$ for all j and b help us to solve the knapsack problem ?
4. What is the value of $f(1, b)$? (clue : it depends on the value of b)
5. Give a recursive formula for f.
6. Write an algorithm to compute f for all j and b. What is its complexity ? What would be the complexity of a brute force algorithm to solve the knapsack problem ?
7. Apply one of your dynamic programming algorithm to solve, in the following example, the knapsack problem such that $V = 10$

objet	1	2	3
v_i	3	5	2
m_i	2	2	4
u_i	10	15	8

▶ **Correction**

1. The utility is $U(x_1, x_2, \ldots, x_n) = \sum_{i=1}^{n} x_i \cdot u_i$.
2. For the volume $V(x_1, x_2, \ldots, x_n) = \sum_{i=1}^{n} x_i \cdot v_i \leq V$.
3. We have to compute $f(n, V)$.
4. $f(1, b)$ consists of placing object 1 as many times as possible in volume b :

$$f(1, b) = \min(\left\lfloor \frac{b}{v_1} \right\rfloor, m_1) \cdot u_1$$

   .
5. If we want to put objects 1 to j in a bag of volume b, we can try to fit item j as many times as possible, then make a recursive call with the remaining objects and volume :

$$f(j, b) = \max_{k=0..\min(\left\lfloor \frac{b}{v_j} \right\rfloor, m_j)} f(j-1, b - k \cdot v_j) + k \cdot u_j$$

6. We can then deduce the two following algorithms

```
1: function f(j, b)
2:     Si j = 1 Alors
3:         Renvoyer min(⌊b/v_1⌋, m_1) · u_1
4:     Renvoyer max_{k=0..min(⌊b/v_j⌋, m_j)} f(j − 1, b − k · v_j) + k · u_j
```

We write $T : [\![1; n]\!] \times [\![0; V]\!] \to \mathbb{N}$, the table T is empty at the beginning.

```
1: function f_Memo(j, b)
2:     Si T(j, b) is empty Alors
3:         Si j = 1 Alors
4:             T(j, b) ← min(⌊b/v_1⌋, m_1) · u_1
5:         Sinon
6:             T(j, b) ← max_{k=0..min(⌊b/v_j⌋, m_j)} f(j − 1, b − k · v_j) + k · u_j
7:     Renvoyer T(j, b)
```

We perform lines 3 to 6 only once for each (j, b), because subsequent times $T(j, b)$ is non-empty and the **If** is false. Thus, we execute at most $V \cdot n$ times lines 3 to 6. Therefore, we perform line 6 at most $V \cdot n$ times with its $\min(\left\lfloor \frac{b}{v_j} \right\rfloor, m_j) \leq \min(V, \max m_j)$ recursive calls. Hence, there are at most $V \cdot n \cdot \min(V, \max m_j)$ recursive calls to F_{Memo}.
Among these calls, there are at most $V \cdot n$ calls executing lines 3 to 6, taking time $O(\min(V, \max m_j))$, and at most $V \cdot n \cdot \min(V, \max m_j)$ calls that do not execute them in time $O(1)$.
Thus, we have a complexity of the order of $O(V \cdot n \cdot \min(V, \max m_j))$.

7. To trace back the recursive calls, we can notice that : $\text{F}(j, b)$ can call $\text{F}(j-1, b')$ for any $b' \leq b$. Therefore, if we have computed $\text{F}(j-1, b')$ for all $b' \leq V$, then we can compute $\text{F}(j, b)$ for all $b \leq V$.
The terminal cases are the cases where $j = 1$.

1: **function** $f_{Iter}(n, V)$
2: **Pour** $b \in [\![0; V]\!]$ **Faire**
3: $T(1, b) \leftarrow \min(\left\lfloor \frac{b}{v_1} \right\rfloor, m_1) \cdot u_1$
4: **Pour** $j \in [\![2; n]\!]$ **Faire**
5: **Pour** $b \in [\![0; V]\!]$ **Faire**
6: $T(j, b) \leftarrow \max_{k=0..\min(\left\lfloor \frac{b}{v_j} \right\rfloor, m_j)} f(j-1, b - k \cdot v_j) + k \cdot u_j$
7: **Renvoyer** $T(n, V)$

The complexities on the loops 2-3 and 4-6 are $O(V)$ and $O(n \cdot V \cdot \min(V, \max m_j))$.

8.

j \ b	1	2	3	4	5	6	7	8	9	10
1	0	0	10	10	10	20	20	20	20	20
2	0	0	10	10	15	20	20	25	25	30
3	0	8	10	16	18	24	26	32	34	36

The answer is 36 : twice the object 3 and twice the object 1.

Exercice 4 — *Square patch*

Let A be a $n \times n$ square binary matrix (only 1 and 0 entries). Describe an algorithm to find the largest square submatrix of A containing only 1 entries ; and the coordinates of its upper left corner. What is its complexity ?

▶ **Correction**

A good answer involves defining $f(i, j)$ as the size of the largest square of 1s whose top-left corner is (i, j). In this case, to find the correct answer, one must return the maximum of the $f(i, j)$.
Here is a recurrence relation for this function :

— $f(i, j) = 0$ if $A_{i,j} = 0$
— $f(i, j) = 1 + \min(f_{i+1,j+1}, f_{i,j+1}, f_{i+1,j})$ otherwise

Exercice 5 — *Climbing stairs*

1. You are in front of a stair with m steps. You can climb the steps one by one and/or two by two. How many ways of climbing this stair are there ?
   We now consider that you can climb α steps at a time, for $\alpha \in (\alpha_1, \alpha_2, \ldots, \alpha_p)$. For example, in the previous question, $p = 2$, $\alpha_1 = 1$ and $\alpha_2 = 2$. Wa assume that the numbers α_i are distinct and sorted. Let $N(m)$ be the number of ways of climbing an m steps stair.
2. Give a recursive formula for $N(m)$.
3. By using pseudo-code, describe a dynamic programming algorithm to solve this problem. What is the complexity of this algorithm ?
4. We assume $p = 3, \alpha_1 = 2, \alpha_2 = 3, \alpha_3 = 5$, give $N(m)$ for $0 \leq m \leq 12$.

▶ **Correction**

The answer to question 1 is the m-th Fibonacci number $f(m)$ with $f(0) = 1$ and $f(1) = 1$. Indeed, if we climb one step, there remain $m-1$ steps, and if we take 2 steps at once, there remain $m-2$. Thus, we have the formula $f(m-1) + f(m-2) = f(m)$ for $m \geq 2$. If the staircase has 0 steps, there is one way to climb it, which is to do nothing. If the staircase has 1 step, there is one way to climb it.

The generalization is almost identical : $N(m) = \sum_{\alpha_i \leq m} (N(m - \alpha_i))$.

We find in order for the last question : 1, 0, 1, 1, 1, 3, 2, 5, 6, 8, 14, 16, 27.