




Chap 11 · Bestiaire des algorithmes

Prise de note par Léo BERNARD en MP2I au Lycée du Parc. (Année 2022-2023)

-  Chap 11 · Bestiaire des algorithmes
- I. Diviser pour régner
 - 1.1. Principe
 - 1.2. Exemple idiot : somme d'un tableau
 - 1.3. Exponentiation rapide
 - 1.4. Tri fusion
 - 1.5. Algorithme de Karatsuba
 - 1.6. Complexité
- II. Programmation dynamique
 - 2.1. Un exemple classique
 - 2.2. Principe général
 - 2.3. Top-down ou bottom-up ?
 - 2.4. Stockage des résultats
- III. Algorithmes gloutons

I. Diviser pour régner

1.1. Principe

La méthode diviser pour régner (divide and conquer) est fondamentalement récursive. On souhaite résoudre une instance de taille n d'un certain problème :

- si n est assez petit, c'est un cas de base.
- sinon, on sépare notre instance en un certain nombre d'instances strictement plus petites du même problème.
 - on résout récursivement chacune de ces instances.
 - on reconstruit la solution pour l'instance initiale

Si on divise le problème en k sous-problèmes de taille $\frac{n}{p}$ et que l'on note $T(n)$, le temps de calcul maximal sur une instance de taille n , on a :

$$T(n) \leq \underbrace{f_s(n)}_{\text{séparation}} + kT\left(\frac{n}{p}\right) + \underbrace{f_f(n)}_{\text{reconstruction}}$$

 Remarque :

En toute rigueur, il faudrait des parties entières (par exemple, $T(\lfloor \frac{n}{2} \rfloor) + T(\lceil \frac{n}{2} \rceil)$). Mais, ça complique énormément tout en n'apportant rien.

1.2. Exemple idiot : somme d'un tableau

On dispose d'un tableau de n flottants et on souhaite en déterminer la somme.

- si $n = 0$ ou $n = 1$, aucune addition à faire.
- sinon, on effectue un appel récursif sur la partie gauche et un sur la partie droite, et on additionne les deux résultats.

```
double somme(double* t, int n) {  
    if (n == 0) return 0;  
    if (n == 1) return t[0];  
    int a = somme(t, n/2);  
    int b = somme(&t[n/2], n - n/2);  
    return a + b;  
}
```

En notant $f(n)$ le nombre d'additions entre flottants pour un tableau de taille n , on a :


$$f(2n) = 1 + 2f(n)$$

En posant $u_k = f(2^k)$, on trouve

$$\begin{cases} u_0 &= 0 \\ u_{k+1} &= 1 + 2u_k \end{cases} \implies \text{donc } u_k = \boxed{2^k - 1 = f(2^k)}$$

C'est exactement la même chose que par l'approche naïve.

1.3. Exponentiation rapide

 **Rappel :** Algorithme naïf, pour calculer a^n , n ou $n - 1$ multiplications.

```
let rec expo a n =  
    if n = 0 then 1  
    else if n mod 2 = 0 then expo (a*a) (n/2)  
    else a * expo (a*a) (n/2)
```

Avec $f(n)$ le nombre de multiplications,

$$f(n) = \begin{cases} f(\lfloor \frac{n}{2} \rfloor) + 1 & \text{si } n \text{ est pair} \\ f(\lfloor \frac{n}{2} \rfloor) + 2 & \text{si } n \text{ est impair} \end{cases}$$

Pour une puissance de 2 :

$$\begin{cases} f(2^{k+1}) = f(2^k) + 1 \\ f(2^0) = 1 \end{cases}$$

donc $f(2^k) = k + 1$

Dans le cas général :

$$f(n) \leq f\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + 2$$

On définit \tilde{f} par

$$\tilde{f} : \begin{cases} \tilde{f}(n) = \tilde{f}\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + 2 \\ \tilde{f}(1) = 1 \end{cases}$$

Par récurrence forte

$$f(n) \leq \tilde{f}(n)$$

Or \tilde{f} est croissante et $\tilde{f}(2^k) = 2k + 1$

Donc $f(n) \leq \tilde{f}(n) \leq \tilde{f}(2^{\lceil \log n \rceil}) = 2\lceil \log n \rceil + 1$

Donc $f(n) = \mathcal{O}(\log n)$

1.4. Tri fusion

```
let rec tri u =
  match u with
  | [] | [_] -> u
  | _ ->
    let a, b = separate u in
    let a_trie, b_trie = tri a, tri b in
    fusionne a_trie b_trie
```

La fonction `separe` est de complexité $\mathcal{O}(|u|)$.

La fonction `fusionne` est de complexité $\mathcal{O}(|a| + |b|)$.

Pour le tri fusion sur une liste de taille 2^i

- un appel à `separe` qui prend un temps majoré par $A2^i$
- 2 appels récursifs sur des listes de taille 2^{i-1}
- un appel à `fusionne` sur 2 listes de taille 2^{i-1} qui prend un temps majoré par $B2^i$

$$\begin{aligned} T(2^i) &\leq A2^i + 2T(2^{i-1}) + B2^i \\ &\leq 2T(2^{i-1}) + C2^i \quad (\text{où } C \text{ est une liste}) \end{aligned}$$

On divise par 2^i

$$\frac{T(2^i)}{2^i} - \frac{T(2^{i-1})}{2^{i-1}} \leq C$$

On somme pour i allant de 1 à n :

$$\frac{T(2^n)}{2^n} - \frac{T(1)}{1} \leq nC$$

$$\implies T(2^n) \leq Dn2^n$$

Donc $T(2^n) = \mathcal{O}(n2^n)$

Ici, T est croissante, pour n quelconque : $n \leq 2^{\lceil \log_2 n \rceil}$.

Donc $T(n) \leq T(2^{\lceil \log_2 n \rceil}) \leq D \lceil \log_2 n \rceil 2^{\lceil \log_2 n \rceil}$.

Ainsi, $\boxed{T(n) = \mathcal{O}(n \log n)}$.

1.5. Algorithme de Karatsuba

On souhaite multiplier deux entiers en base b fixée, de taille non bornée.

On considère que multiplier ou ajouter deux *chiffres* est une opération élémentaire.

Un entier « long » $x = \sum_{i=0}^{n-1} x_i b^i$ est représenté par un tableau $\llbracket x_0, x_1, \dots, x_{n-1} \rrbracket$ d'entiers machine et on dispose de 3 fonctions :

- `longueur(x)` qui renvoie n en temps constant,
- `chiffre(x, i)` qui renvoie x_i en temps constant (et 0 si $i \geq n$),
- `normalise(x)` qui s'exécute en $\mathcal{O}(n)$ et élimine les zéros à gauche du tableau.

On suppose que tous les nombres manipulés sont normalisés.

+ Addition

Ajouter deux nombres de taille n peut se faire en temps $\mathcal{O}(n)$

fonction Ajoute(x, y) :

$n \leftarrow \max(\text{longueur}(x), \text{longueur}(y))$

$z \leftarrow (0, \dots, 0)$ de taille $n + 1$

retenue $\leftarrow 0$

pour $i = 0$ à $n - 1$:

| $c \leftarrow \text{chiffre}(x, i) + \text{chiffre}(y, i) + \text{retenue}$

| $z_i \leftarrow c \bmod b$

| retenue $\leftarrow c/b$

$z_n \leftarrow \text{retenue}$

renvoyer `normalise(z)`

+ Multiplication par une puissance de b

Multiplier un nombre par b^k peut se faire un temps proportionnel à son nombre de chiffres.

$$\text{MulBase}\left((x_0, \dots, x_{n-1}), k\right) = (\underbrace{0, \dots, 0}_{k \text{ zéros}}, x_0, \dots, x_{n-1})$$

+ Multiplication par un chiffre

$\text{MulChiffre}(x, c)$ se fait en temps $\mathcal{O}(n)$.

+ Multiplication naïve

fonction $\text{MulNaïve}(x, y)$:

$n \leftarrow \text{longueur}(y)$

$s \leftarrow ()$ // Tableau vide représentant 0

pour $i = 0 \rightarrow n - 1$:

$s \leftarrow \text{Ajoute}(s, \text{MulBase}(\text{MulChiffre}(x, \text{chiffre}(y, i))))$

renvoyer s

Complexité :

$\text{MulNaïve}(x, y)$

$m \leftarrow \text{longueur}(y)$

$s \leftarrow ()$ tableau vide représentant 0

Pour $i = 0 \rightarrow m - 1$:

$\mathcal{O}(m^2) \left\{ \begin{array}{l} \mathcal{O}(m) \left(s \leftarrow \text{Ajoute}(s, \text{MulBase}(\text{MulChiffre}(x, \text{chiffre}(y, i)))) \right) \right. \\ \text{renvoyer } s \end{array} \right.$

Diagram illustrating the complexity analysis of the naive multiplication algorithm. The diagram shows the function $\text{MulNaïve}(x, y)$ and its steps. The complexity is analyzed as $\mathcal{O}(m^2)$ for the entire function, with $\mathcal{O}(m)$ for the inner loop. The inner loop consists of $\text{Ajoute}(s, \text{MulBase}(\text{MulChiffre}(x, \text{chiffre}(y, i))))$, which is annotated with $\mathcal{O}(m)$ for the Ajoute operation and $\mathcal{O}(m)$ for the MulBase operation. The MulChiffre operation is annotated with $\mathcal{O}(1)$. The final result is $\text{renvoyer } s$.

✂ Diviser pour régner

1e idée :

On considère que x et y possèdent tous les deux $2n$ chiffres.

On les décompose

$$x = x_{lo} + b^n x_{hi}$$

$$y = y_{lo} + b^n y_{hi}$$

$$x = [\underbrace{x_0, x_1, \dots, x_{n-1}}_{x_{lo}}, \underbrace{x_n, \dots, x_{2n-1}}_{x_{hi}}]$$

On a alors :

$$xy = x_l y_l + (x_l y_h + x_h y_l) b^n + x_h y_h b^{2n}$$

Analyse des opérations :

- 3 additions dont les opérandes sont au plus $2n$ chiffres : $\mathcal{O}(n)$
- 2 décalages : $\mathcal{O}(n)$
- 4 multiplications d'entiers de taille n .

$$\implies T(2n) = 4T(n) + \mathcal{O}(n)$$

$$T(2^{n+1}) = 4T(2^n) + \mathcal{O}(2^n)$$

$$\frac{T(2^{n+1})}{2^{n+1}} = 2 \frac{T(2^n)}{2^n} + \mathcal{O}(1)$$

$$\frac{T(2^{n+2})}{2^{n+1}} \geq 2 \frac{T(2^n)}{2^n}$$

$$\implies \frac{T(2^n)}{2^n} \geq 2^n \frac{T(1)}{1}$$

$$\implies T(2^n) \geq 4^n \times C$$

On trouve $T(2^n) \geq 4T(n)$

Donc $T(2^k) \geq A4^k$, ce qui est quadratique.

Algorithme de Karatsuma

On remplace une multiplication par des additions

On pose :

$$u = x_l y_l$$

$$v = x_h y_h$$

$$w = (x_l + x_h)(y_l + y_h)$$

On a alors

$$xy = u + (w - u - v)b^n + vb^{2n}$$

Analyse des opérations :

- 6 additions dont les opérandes ont au plus $2n$ chiffres : $\mathcal{O}(n)$
- 2 décalages : $\mathcal{O}(n)$
- 3 multiplications d'entiers de taille n .

Ainsi :

$$T(2^i) \leq 3T(2^{i-1}) + A2^i$$

On divise par 3^i :

$$\frac{T(2^i)}{3^i} - \frac{T(2^{i-1})}{3^{i-1}} \leq A(2/3)^i$$

En sommant pour $i = 1$ à k

$$\frac{T(2^k)}{3^k} - B \leq A \underbrace{\sum_{i=1}^k \left(\frac{2}{3}\right)^i}_{\text{bornée} \rightarrow \frac{2}{3} \frac{1}{1-\frac{2}{3}} = 2}$$

Donc $T(2^k) \leq C3^k$

Or $3^k = (2^{\log 3})^k = (2^k)^{\log 3}$

Si $n = 2^k$, $T(n) = O(n^{\log 3}) \approx O(n^{1.59})$

Fun fact, en 2019 il y a eu un algo en $O(n \log n)$ mais qui nécessiterait des nombres avec un nombre de chiffres plus grand que le nombre d'atomes dans l'univers.

1.6. Complexité

Supplément de cours, adapté du poly de M. Rebout

On rappelle le principe général de la méthode, et l'on fixe les notations pour cette partie :

- si l'instance à traiter est un cas de base, c'est direct;
- sinon, on la sépare en a instances de taille n/b (en omettant les parties entières), ce qui prend un temps $f_s(n)$;
 - on fait les appels récursifs, ce qui prend un temps $aT(\frac{n}{b})$;
 - on « fusionne », c'est-à-dire que l'on calcule le résultat final à partir des résultats des appels récursifs, ce qui prend un temps $f_f(n)$.

Au total, en notant $f = f_s + f_f$, on a la relation suivante :

$$T(n) = aT(n/b) + f(n)$$

Avant de passer à l'analyse, on peut faire quelques remarques.

- On n'a pas précisé ce que représente $T(n)$, mais il s'agit logiquement de la complexité dans le pire des cas pour une instance de taille n . Comme il n'y a aucune raison que le pire cas pour n donne lieu à des appels récursifs sur des pires cas pour n/b , on aura plutôt $T(n) \leq aT(n/b) + f(n)$. Si l'on souhaite obtenir un \mathcal{O} , ce n'est absolument pas un problème.

- Si n n'est pas un multiple b , ce qui précède n'a pas tellement de sens. On fera donc l'analyse pour $n = b^k$ et l'on généralisera ensuite.
- Pour pouvoir faire cette généralisation, on aura besoin que T soit croissante. Ce n'est pas le cas en général (même si ce n'est « pas loin d'être vrai » le plus souvent) : on peut remplacer $T(n)$ par $T'(n) = \max_{k \leq n} T(k)$ et faire disparaître le problème. On aura encore :

$$T'(n) \leq aT'(n/b) + f(n)$$

et T' est croissante par construction. On pourra alors remarquer que $T'(n) \leq T'(b^{\lceil \log_b n \rceil})$ pour traiter le cas général (où n n'est pas une puissance de b).

La bonne manière de raisonner ici est d'observer la structure de l'arbre d'appels, et de déterminer la quantité de travail effectuée à chaque niveau de cet arbre :

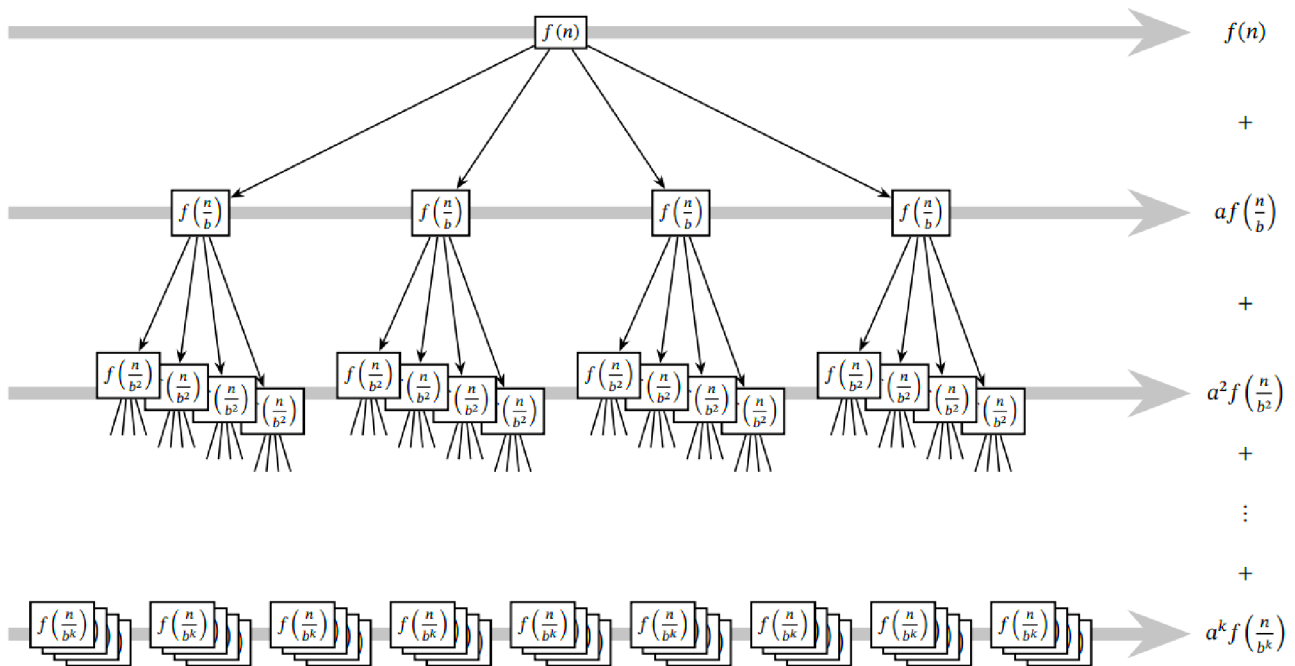


FIGURE 11.1 – Analyse d'une relation $T(n) = aT(n/b) + f(n)$, avec $n = b^k$.

Les feuilles de cet arbre correspondent aux cas de base de l'algorithme; elles sont étiquetées par $f(1)$. La complexité $T(n)$ est alors égale à la somme de toutes les valeurs dans l'arbre de récursion :

$$T(n) = T(b^k) = \sum_{i=0}^k a^i f\left(\frac{n}{b^i}\right) = \sum_{i=0}^k a^i f(b^{k-i})$$

Si l'on souhaite retrouver ce résultat sans faire de dessin, c'est très simple :

- on pose $u_i = \frac{T(b^i)}{a^i}$;

- on a alors, grâce à la relation de récurrence sur T , $u_{i+1} = u_i + \frac{f(b^i)}{a^i}$;
- on somme les $u_{i+1} - u_i$ et l'on obtient : $u_k - u_0 = \sum_{i=0}^{k-1} \frac{f(b^i)}{a^i}$
- on multiplie par a^k , on obtient : $T(b^k) = a^k T(b^0) + \sum_{i=0}^{k-1} a^{k-i} f(b^i)$
- en posant $f(1) = T(1)$ (ce qui est parfaitement logique), on obtient exactement le même résultat qu'auparavant.

Dans le cas particulier, mais classique, où $f(n) = \Theta(n^p)$ pour une certaine constante p , on dispose d'un théorème classique mais hors-programme et pas vraiment utile à retenir :

💡 **Théorème (Master theorem) :** Si T vérifie $T(n) \leq aT(n/b) + An^p$ (avec A une constante) et si T est croissante, on a :

- si $b^p < a$, alors $T(n) = \mathcal{O}(n^{\log_b a})$;
- si $b^p = a$, alors $T(n) = \mathcal{O}(n^{\log_b a} \log n) = \mathcal{O}(n^p \log n)$;
- si $b^p > a$, alors $T(n) = \mathcal{O}(n^p)$.

Revenons à la somme précédente pour comprendre ces résultats. Avec la supposition sur la fonction f , cette somme se réécrit (ça manque un tout petit peu de rigueur, mais c'est suffisant ici) :

$$T(b^k) = \sum_{i=0}^k a^i f(b^{k-i}) = \sum_{i=0}^k Aa^i b^{p(k-i)}$$

Calculons le rapport de deux termes successifs :

$$\frac{Aa^{i+1} b^{p(k-i-1)}}{Aa^i b^{p(k-i)}} = \frac{a}{b^p}$$

Ce rapport est donc constant :

- s'il est strictement plus grand que 1, on somme les termes d'une suite géométrique strictement croissante : c'est le dernier terme qui est prépondérant :

$$T(n) = \mathcal{O}(a^k f(1)) = \mathcal{O}(a^{\log_b n}) = \mathcal{O}(n^{\log_b a});$$

- s'il est strictement plus petit que 1, on somme les termes d'une suite géométrique strictement décroissante : c'est le premier terme qui est prépondérant :

$$T(n) = \mathcal{O}(f(n)) = \mathcal{O}(n^p);$$

- s'il vaut 1, la suite est constante et la somme équivaut au nombre de termes fois un de ces termes :

$$T(n) = \mathcal{O}(\log n f(n)) = \mathcal{O}(n^p \log n)$$

Finalement, ce qui est important ici est de bien comprendre l'intuition derrière ces trois cas :

- dans le premier cas, le terme en $\mathcal{O}(n^p)$ (qui correspond au travail de séparation-fusion) est négligeable devant le coût des appels récurifs. Si l'on considère l'arbre d'appels, la majorité du temps est passé dans les feuilles. Autrement dit, la somme $\sum_{i=0}^k a^i f(b^{k-i})$ est de l'ordre de a^k .
- dans le troisième cas, c'est le coût des appels récurifs qui est négligeable : la majorité du temps est passé à la racine de l'arbre d'appels, la somme est de l'ordre de $f(b^k)$;
- dans le deuxième cas, l'étape de séparation-fusion et les appels récurifs ont un coût similaire : autrement dit, le coût total pour traiter chaque niveau de l'arbre d'appels est le même (approximativement) et le coût total s'obtient en multipliant ce coût par niveau par le nombre de niveaux.

👉 **Remarque :** Comme dit plus haut, il est déconseillé de chercher à retenir ce théorème : refaites le raisonnement, en partant de l'arbre d'appels ou directement par le calcul, pour le cas qui vous intéresse.

📖 **Exercice :** Déterminer par cette méthode la complexité temporelle :

1. de la recherche dichotomique;
2. du tri fusion;
3. de l'algorithme de Karatsuba
4. de l'algorithme de Strassen, qui permet de réduire la multiplication de deux matrices $2n \times 2n$ à 7 multiplications de matrices $n \times n$ et un nombre constant d'additions de matrices $n \times n$ ou $2n \times 2n$.



Étudiant ou étudiante de MP2I découvrant le Master theorem – une allégorie

II. Programmation dynamique

2.1. Un exemple classique

Le problème du sac à dos

On dispose d'un sac à dos dont la charge utile est limitée à un poids maximal p_{\max} et de n objets o_0, o_1, \dots, o_{n-1} possédant chacun un poids p_0, \dots, p_{n-1} et une valeur v_0, \dots, v_{n-1} .

Le but est de remplir le sac en emportant la valeur maximale sans dépasser le poids limite.

Formalisation

On donne un entier $n > 0$ et deux tableaux d'entiers p et v , de taille n et un entier p_{\max} .

- un sac est un sous-ensemble de $\llbracket 0, n \rrbracket$
- le poids d'un sac S est $p(S) = \sum_{i \in S} p_i$
- la valeur d'un sac S est $v(S) = \sum_{i \in S} v_i$

La valeur optimale pour un poids p est

$$v_{\text{opt}}(p) = \max_{\substack{S \in \mathcal{P}(\llbracket 0, n \rrbracket) \\ \text{tel que } p(S) \leq p_{\max}}} v(S)$$

Le problème est donc de calculer $v_{\text{opt}}(p_{\max})$ et de fournir un exemple de sac qui réalise cette valeur.

Solution en force brute

On teste tous les sacs possibles : il y a $|\mathcal{P}(\llbracket 0, n \rrbracket)| = 2^n$ sacs différents.

Générer tous ces sacs et calculer leur poids et valeur prend un temps proportionnel à $n2^n$... bof.

Solution récursive

On note $f(k, p)$ la valeur maximale obtenue avec les objets o_0, \dots, o_k ne dépassant pas le poids p .

→ On cherche $f(n-1, p_{\max})$.

Cas de base

$$f(0, p) = \begin{cases} v_0 & \text{si } p_0 \leq p \\ 0 & \text{si } p_0 > p \end{cases}$$

Formule générale

$$f(k+1, p) = \begin{cases} f(k, p) & \text{si } p_{k+1} > p \\ \max \left(\underbrace{v_{k+1} + f(k, p - p_{k+1})}_{\text{si on retient } o_{k+1}}, f(k, p) \right) & \text{si } p_{k+1} \leq p \end{cases}$$

Principe d'optimalité

L'écriture de relation de récurrence repose sur un principe tellement naturel qu'on l'applique sans y penser, mais il est intéressant de le mettre en valeur.

Dans une suite optimale de décisions, toute sous-suite doit aussi être optimale.

Dans notre exemple, on utilise ce principe d'optimalité en affirmant que si o_i appartient aux objets à emporter pour un poids p , alors la solution au problème pour un poids maximal ($p -$

p_i) sera donnée par le même ensemble privé de i .

```
let sac_a_dos p v pmax =  
  let rec sac_aux k poids =  
    match k with  
    | 0 -> if p.(0) > poids then 0 else v.(0)  
    | _ ->  
      if p.(k) > poids then sac_aux (k-1) poids  
      else max (sac_aux (k-1) poids)  
        (v.(k) + sac_aux (k-1) (poids - p.(k)))  
  in sac_aux (Array.length p - 1) pmax
```

Cette fonction est très inefficace.

Pour le voir, on modifie la fonction pour afficher le nombre d'appels à `sac_aux` avec des paramètres `k` et `poids` donnés.

(Voir tableau sur poly_prof).

Le problème est immédiatement visible : on passe notre temps à recalculer la même chose : c'est précisément le cas où les sous-problèmes se chevauchent.

La solution est immédiate : se souvenir des calculs déjà faits.

Solution dynamique : version itérative

On construit un tableau contenant les valeurs $f(k, p)$.

On l'initialise à 0, et on le remplit au fur et à mesure (en utilisant la même relation de récurrence).

Quand on a terminé, on récupère le dernier élément du tableau.

```
let sac_a_dos_dyn p v pmax =  
  let n = Array.length p in  
  let t = Array.make_matrix n (pmax + 1) in  
  (*On remplit le tableau ligne par ligne*)  
  for k = p.(0) to pmax do (* }*)  
    t.(0).(k) <- v.(0)      (* } 1ère ligne *)  
  done;                      (* }*)  
  for i = 1 to n-1 do  
    for poids = 0 to pmax do  
      if p.(i) <= poids do  
        t.(i).(poids) <- max (t.(i-1).(poids)) (v.(i) + t.(i-1).(poids - p.(i)))  
      else  
        t.(i).(poids) <- t.(i-1).(poids)  
      done  
    done;  
  t.(n-1).(pmax)
```

Complexité : $\mathcal{O}(n \times p_{\max})$ (spatiale et temporelle)

Solution dynamique : récursion avec mémorisation

Il est tout aussi simple de garder une solution récursive.

On utilise la technique de mémorisation :

- on crée un tableau destiné à recevoir les valeurs déjà calculées
- ce tableau est initialisé avec une valeur « impossible ».
 - ici, `(-1)` car toutes les valeurs calculées sont positives
 - dans le cas général, `None`.
- quand on appelle la fonction récursive sur les valeurs `k` et `poids` :
 1. elle regarde dans le tableau si la case correspondante contient une vraie valeur,
 2. si c'est le cas, on renvoie cette valeur
 3. sinon, on effectue le calcul, on stocke le résultat dans le tableau, avant de le renvoyer.

```
let sac_a_dos_rec p v pmax =  
  let n = Array.length p in  
  let t = Array.make_matrix n (pmax+1) (-1) in  
  let rec aux k poids =  
    if t.(k).(poids) <> -1 then t.(k).(poids)  
    else begin  
      if k = 0 then  
        t.(k).(poids) <- if p.(0) <= poids then v.(0) else 0  
      else if p.(k) <= poids then  
        t.(k).(poids) <- max (aux (k-1) poids) (v.(k) + aux (k-1) (poids - p.(k)))  
      else  
        t.(k).(poids) <- aux (k-1) poids;  
      t.(k).(poids)  
    end  
  in aux (n-1) pmax
```

Reconstruction de la solution optimale.

$$\text{Données : } \begin{array}{c|cccccccccc} p & 3 & 8 & 5 & 1 & 6 & 1 & 2 & 6 & 6 \\ \hline v & 1 & 2 & 6 & 3 & 7 & 8 & 2 & 3 & 4 \end{array}$$

$$p_{\max} = 14$$

Version itérative															
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1
1	0	0	0	1	1	1	1	1	2	2	2	3	3	3	3
2	0	0	0	1	1	6	6	6	7	7	7	7	7	8	8
3	0	3	3	3	4	6	9	9	9	10	10	10	10	10	11
4	0	3	3	3	4	6	9	10	10	10	11	13	16	16	16
5	0	8	11	11	11	12	14	17	18	18	18	19	21	24	24
6	0	8	11	11	13	13	14	17	18	19	20	20	21	24	24
7	0	8	11	11	13	13	14	17	18	19	20	20	21	24	24
8	0	8	11	11	13	13	14	17	18	19	20	20	21	24	24

Version récursive avec mémoïsation															
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1
1	0	0	0	1	1	1	1	1	2	2	2	3	3	3	3
2	0	0	0	x	1	6	6	6	7	x	7	7	7	8	8
3	0	3	3	x	x	6	9	9	9	x	x	10	10	10	11
4	0	3	3	x	x	6	9	10	10	x	x	13	16	16	16
5	0	x	11	x	x	x	14	x	18	x	x	x	21	x	24
6	x	x	11	x	x	x	x	x	18	x	x	x	x	x	24
7	x	x	x	x	x	x	x	x	18	x	x	x	x	x	24
8	x	x	x	x	x	x	x	x	x	x	x	x	x	x	24

- On commence par chercher la valeur optimale en construisant le tableau.
- On regarde ce qu'il faut modifier dans l'algo pour reconstruire la solution : on s'appuie sur la relation de récurrence.
 - Parfois, le tableau suffit.
 - Parfois, il faut stocker un peu plus d'informations.

Dans notre exemple, pour le cas général,

$$f(k+1, p) = \max(f(k, p), v_{k+1} + f(k, p - p_{k+1}))$$

- Si la valeur de la case est égale à la valeur juste au dessus, on ne conserve pas l'objet.
- Sinon, on conserve l'objet et on change de colonne.

On imagine avoir modifié le programme précédent pour renvoyer le tableau plutôt que la valeur optimale finale.

```
let remplissage_sac p v pmax =
  let t = sac_a_dos p v pmax in
  let rec aux liste k poids =
    match k with
    | 0 -> if t.(0).(poids) <> 0 then 0 :: liste else liste
    | _ ->
      if t.(k).(poids) = t.(k-1).(poids)
      then aux liste (k-1) poids
      else aux (k::liste) (k-1) (poids - p.(k))
  in aux [] (Array.length p - 1) pmax
```

2.2. Principe général

La programmation dynamique s'applique à des problèmes (d'optimisation) ayant les 2 propriétés suivantes :

- **Sous-structure optimale** : une solution optimale contient des solutions optimales pour des instances plus petites du même problème.
- **Chevauchement de sous-problèmes** : une solution récursive naïve conduit à résoudre plusieurs fois les mêmes sous-problèmes.

2.3. Top-down ou bottom-up ?

On a deux choix de programmation :

- **L'approche ascendante** (*ou bottom up*) consiste à résoudre toutes les instances de taille 1, puis celle de taille 2, et ainsi de suite. Une **solution itérative** est appropriée ici.
- **L'approche descendante** (*ou top down*) consiste à traduire directement la relation de récurrence. Une **solution récursive** est appropriée ici.

En règle générale, l'approche descendante est plus simple à écrire (donc moins d'erreurs...). En terme de performances, ça dépend du problème.

- S'il est possible de déterminer exactement quels calculs vont être nécessaires, l'approche bottom-up peut être intéressante ; dans certains cas, on peut ne garder en mémoire que les résultats récents (ceux correspondants aux instances de taille $n - 1$) et ainsi faire baisser la complexité spatiale.
- En revanche, s'il n'est pas évident de déterminer les calculs nécessaires, l'approche top-down peut éviter des calculs inutiles.

 **Exemple :**

La suite de Fibonacci (F_n)
$$\begin{cases} F_0 = 0 \\ F_1 = 1 \\ F_{n+2} = F_{n+1} + F_n \end{cases}$$

- **Récuratif naïf :**
 - Temporelle : exponentielle,
 - Spatiale : linéaire (pile d'appels)
- **Récuratif avec mémoïsation :**
 - Temporelle : linéaire,
 - Spatiale : linéaire
- **Bottom-up optimisé :**
 - Temporelle : linéaire,
 - Spatiale : constante ($O(1)$) (2 cases mémoires)

2.4. Stockage des résultats

Pour l'instant, on a toujours utilisé des tableaux pour stocker les résultats partiels. C'est possible le plus souvent, mais ça suppose que :

- La récursion se fait sur des paramètres *entiers*
- Ces paramètres sont plus petits pour les résultats partiels que pour le résultat final (ou au moins bornés et prévisibles)

Si ces conditions ne sont pas réunies, on utilise un **dictionnaire**.

 **Exemple :** Suite de Syracuse

Pour toute entier $a \in \mathbb{N}^*$, on définit $(u_n(a))_{n \in \mathbb{N}}$ par :

$$\begin{cases} u_0(a) = a \\ \forall n \in \mathbb{N}, \quad u_{n+1}(a) = \begin{cases} \frac{u_n(a)}{2} & \text{si } u_n(a) \text{ est pair} \\ 3u_n(a) + 1 & \text{sinon} \end{cases} \end{cases}$$

On note $\text{syr}(a)$ le plus petit n tel que $u_n = 1$ (le temps de vol de a)

Si on veut calculer une seule valeur de $\text{syr}(a)$, on procède bêtement :

```
let rec syr a =
  if a = 1 then 0
  else if a mod 2 = 0 then 1 + syr (a/2)
  else 1 + syr (3*a + 1)
```

Dès que l'on calcule plusieurs valeurs, on a de fortes chances de retomber sur une valeur déjà traitée.

On veut calculer $\max_{k \in [1, n]} \text{syr}(k)$: on utilise un dictionnaire car on ne sait pas borner k pour les appels à $\text{syr}(k)$.

```
let max_syr n =
  let cache = Hashtbl.create n in
  Hashtbl.add cache 1 0;
  let rec syr a =
    if Hashtbl.mem cache a then Hashtbl.find cache a (*mem = member*)
    else ( (* a ≠ 1*)
      let res =
        if a mod 2 = 0 then 1 + syr (a/2)
        else 1 + syr (3*a + 1)
      in Hashtbl.add cache a res;
      res
    )
  in
  let rec max_aux k =
    if k = 1 then 0
    else max (syr k) (max_aux (k-1))
  in max_aux n
```

III. Algorithmes gloutons

Les algorithmes gloutons (*en anglais : greedy algorithms*) sont une classe d'algorithmes pour des problèmes d'optimisation qui ont les propriétés suivantes :

- Ils construisent leur solution de manière graduelle (autrement dit, ils décomposent le choix global à réaliser en une série de choix successifs **sur lequel ils ne reviennent pas ultérieurement**);
- À chaque étape, ils font un choix **localement optimal**.

Ces algorithmes ont des avantages :

- Simple à concevoir et à implémenter
- Exécution rapide

En général, on n'obtient pas forcément une solution globalement optimale.

Trois situations possibles :

1. Il existe un algorithme glouton qui fournit une **solution optimale** : cet algorithme est alors souvent le meilleur pour le problème étudié.
2. Il existe un algorithme glouton qui fournit une **solution proche de la solution optimale** : si le problème est très compliqué, on s'en contentera.
3. Les algorithmes gloutons donnent des **résultats trop éloignés du résultat optimal** : on fait autrement.

 **Exemple :** Le rendu de monnaie    

On définit un système monétaire comme étant un ensemble $S = \{a_1, a_2, \dots, a_p\}$ avec $1 = a_1 < a_2 < \dots < a_p$, les a_i sont les valeurs des pièces disponibles (i.e. 10, 20, 50, 100, ...).

On cherche une méthode permettant à un commerçant de rendre la monnaie à un client en utilisant *le moins de pièces possibles*.

Étant donné un entier n , on cherche à déterminer un p -uplet (x_1, \dots, x_p) tel que

$$\sum_{i=1}^p x_i a_i = n$$

$$\sum_{i=1}^p x_i \quad \text{soit minimal}$$

Algo glouton : utilise à chaque étape la plus grande pièce possible.

Précondition :

- `system` de longueur p
- `system` est strictement croissant

- `system[0] = 1`
- `target ≥ 0`

```
int* greedy_change(int system[], int p, int target) {
    int* change = malloc(sizeof(int) * p);
    for (int k = p-1; k >= 0; k--) {
        change[k] = target / system[k];
        target = target % system[k];
    }
    return change;
}
```

Exemple :

 Pour le système monétaire européen :

1€, 2€, 5€, 10€, 20€, 50€, 100€, 200€

L'algorithme glouton est optimal.

- Une pièce de 1, 5, 10, 50 ou 100€ n'est jamais utilisée 2 fois dans une solution optimale : pour chacune, il existe une pièce de valeur double.
- Une pièce de 2 ou 20€ n'est jamais utilisée 3 fois dans une solution optimale.

$$3 \times 2 = 1 \times 5 + 1 \times 1$$

- Une pièce de 1€ n'accompagne jamais 2 pièces de 2€
- Un billet de 10€ n'accompagne jamais 2 billets de 20€

La somme maximale que l'on peut atteindre dans une solution optimale sans utiliser la pièce de 200€ :

$$0 \times 1 + 2 \times 2 + 1 \times 5 + 0 \times 10 + 2 \times 20 + 1 \times 50 + 1 \times 100 = 199$$

Si la somme à rendre est strictement supérieure à 199, il faut donc utiliser une pièce de 200€.

On recommence le même raisonnement, sans la pièce de 100 : le nombre est 99.

On recommence le même raisonnement, sans la pièce de 50 : le nombre est 49.

Exemple :

 Système anglais pré-1971 :

1, 3, 6, 12, 24, 30

Pour rendre 48 :

- $30 + 12 + 6$: Algo glouton
- 2×24 : Solution optimale

Crashs d'avions (qui n'ont pas fait de morts)

Une note importante de Raphaël

- vol US1549
- vol BA38
- vol airtransat
- vol aircanada (j'ai plus le nom du vol) (note : c'est aircanada 143)