Chapitre II

Programmation dynamique et algorithmes gloutons

1 Programmation dynamique

1.1 Principe

La programmation dynamique est une méthode algorithmique bottom-up pour résoudre des problèmes d'optimisation. Le concept a été introduit au début des années 1950 par Richard Bellman pour des problèmes de planification.

Cette méthode s'applique à des problèmes d'optimisation : on veut maximiser une fonction d'évaluation f parmi un ensemble de possibilités. Il y a en général plusieurs possibilités qui donnent la valeur optimale. L'objectif est d'en trouver une.

Méthode générale de la programmation dynamique

1) Etudier la structure d'une solution optimale en fonction de découpages possibles en sous-problèmes.
2) A partir de l'étape précédente, écrire une relation récursive sur la valeur optimale de f.
3) Calculer la valeur optimale de f de manière ascendante (bottom-up).
4) Construire une solution optimale à partir des informations calculées.

1.2 Exemple 1 : optimisation de l'associativité pour des multiplications matricielles enchaînées

Problème

Supposons un entier $n \geqslant 1$, une suite de $n+1$ entiers $p = (p_i)_{i \in [0,n]}$ puis une suite de n matrices $(A_1, \ldots, A_n)$ telle que pour tout i, A_i est de dimension $p_{i-1} \times p_i$.
Comme le produit matricielle est une opération sur 2 opérandes seulement, il faut parenthèser le produit $A_1 \ldots A_n$.
On veut déterminer un parenthèsage qui minimise le temps de calcul.

L'algorithme naïf du produit de 2 matrices carrée $p \times p$ est en $O(p^3)$.

Une première version plus performantes a été en $O(p^{log_2 7}) \approx O(p^{2,81})$, l'algorithme de Strassen de type diviser pour régner.

On sait faire mieux aujourd'hui théoriquement et asymptotiquement mais les constantes sont tellement grandes que ces algorithmes ne sont pas performants pour les ordinateurs actuels.

Cf. https://en.wikipedia.org/wiki/Computational_complexity_of_matrix_multiplication).

Dans ces algorithme du produit, l'opération la plus fréquente est la multiplication de coefficients de matrices. La complexité asymptotique est donc proportionnelle au nombre de produits effectués sur les coefficients des matrices.

On choisit donc pour fonction d'évaluation d'un parenthèsage le nombre de produits de coefficients et on cherche un minimum.

1.2.1 Etape 1 : étude de la structure d'une solution optimale

Propriété

Si $(A_1 \dots A_k)(A_{k+1} \dots A_n)$ est un parenthèsage optimal alors $A_1 \dots A_k$ et $A_{k+1} \dots A_n$ sont parenthèsés de manière optimale.

Preuve : Supposons qu'il existe un meilleur parenthèsage pour le produit $A_1 \dots A_k$, en le remplaçant dans $(A_1 \dots A_k)(A_{k+1} \dots A_n)$, on obtiendrait un parenthèsage meilleur pour $A_1 \dots A_n$. (contradiction)
De même pour $A_{k+1} \dots A_n$.

D'après cette propriété, on en déduit un **espace de sous-problèmes à étudier** :

Trouver un parenthèsage optimal pour les produits $A_i \dots A_j$ avec $1 \leqslant i \leqslant j \leqslant n$.

1.2.2 Etape 2 : écriture d'une relation récursive sur les solutions optimales

Notation : Pour tout $1 \leqslant i \leqslant j \leqslant n$, on note $m(i,j)$ le nombre minimum de produits de coefficients matriciels à réaliser pour calculer le produit $A_i \dots A_j$ noté $A_{i,j}$.

Relation récursive

Pour tous $i, j \in [1, n]$ tels que $i \leqslant j$:

- Si $i = j$ alors $m(i,j) = 0$.
- Sinon $m(i,j) = \min\limits_{i \leqslant k < j} \big(m(i,k) + m(k+1,j) + p_{i-1}.p_k.pj\big)$.

Preuve :
Si $i = j$, il n'y a pas d'opération à faire.

Si $i < j$, notons $m = \min\limits_{i \leqslant k < j} \big(m(i,k) + m(k+1,j) + p_{i-1}.p_k.pj\big)$

Considérons un parenthèsage optimum pour $A_{i,j}$ et notons q l'indice de la matrice après laquelle on a fait le premier parenthèsage, alors d'après la propriété montrée dans la section précédente :
$m(i,j) = m(i,q) + m(q+1,j) + p_{i-1}.p_q.pj$.

Comme $1 \leqslant q < j$, il suit que $m(i,j) \geqslant m$ puisque m est un minimum.

Inversement, notons q un indice de $[i,j[$ pour lequel le minimum m est atteint.
On a $m = m(i,q) + m(q+1,j) + p_{i-1}.p_q.pj$.

Donc en faisant le premier parenthèsage après A_q, on obtient m comme nombre de multiplications de coefficients.

Il suit que $m(i,j) \leqslant m$ puisque $m(i,j)$ est minimal. (CQFD)

1.2.3 Etape 3 : algorithme bottom-up de calcul d'une solution optimale

En suivant la propriété précédente, on peut écrire un algorithme récursif naïf mais il serait très peu efficace car les sous-problèmes seraient réévalués à de multiples reprises.

Il est nettement meilleur de construire de manière bottom-up les sous-solutions à partir des sous-problèmes minimaux $A_{i,i}$.

Il s'agit de calculer progressivement et stocker les $m(i,j)$ dans une matrice m de taille $n \times n$.

Comme on a nécessairement $i \leqslant j$, il ne faut calculer que les coefficients de la partie triangulaire supérieure de m.

On initialise d'abord la diagonale à 0.
Puis pour un couple (i, j), le calcul de $m(i, j)$ nécessite la connaissance des $m(i, k)$ et $m(k + 1, j)$ pour $k \in [i, j[$, c.a.d. les coefficients qui sont entre la diagonale et $m(i, j)$, sur la même ligne ou la même colonne que lui.

Il faut donc calculer les $m(i, j)$ en faisant des parcours successifs parallèlement à la diagonale et en s'éloignant progressivement de celle-ci.

Un **algorithme par programmation dynamique** sera (les tableaux sont indicés à partir de 1) :

Algorithme 1.

```
fonction ordre_produit_matrice(p)
    n=longueur(p)-1  # nombre de matrices du produit

    créer deux tableaux d'entiers triangulaires supérieurs m et c de dimension n x n
    # en sortie, m(i,j) stockera la valeur optimale pour A_i...A_j
    # c(i,j) stockera le premier parenthèsage à faire pour construire la solution optimale

    # initialisation pour des sous-problème de taille 1
    pour i=1 à n faire
        m[i][i]=0
    pour t=1 à n-1 faire
        # on parcours la t-ième parallèle au-dessus de la diagonale
        # cela correspond aux sous-problèmes avec t+1 matrices
        pour i=1 à n-t faire
            #j est l'indice de la dernière matrice du sous problème
            j=i+t
            # recherche d'un minimum pour les découpages en deux sous-problèmes
            # initialisation avec le premier découpage possible
            m[i,j]= m[i+1][j]+p[i-1]*p[i]*p[j]
            c[i][j]= i
            pour k=i+1 à j-1 faire
                temp=m[i,k]+m[k+1,j]+p[i-1]*p[k]*p[j]
                si temp<m[i,j] alors
                    m[i,j]=temp
                    c[i,j]=k
    retourner m,c
```

Le nombre optimal de multiplication de coefficients pour le produit $A_1 \dots A_n$ est stocké en `m[1,n]`.

Une fois calculer le tableau `c`, pour appliquer ce produit optimal, on a l'algorithme :

Algorithme 2.

```
fonction produit_optimal(A,c,i,j):
    if i==j:
        retourner A[i]
    retourner produit_optimal(A,c,i,c[i,j]).produit_optimal(A,c,c[i,j]+1,j)
```

1.3 Exemple 2 : plus longue sous-sequence commune (PLSC)

Introduisons un peu de vocabulaire :

- On appelle séquence une suite finie de valeurs appartenant à un domaine D : entiers, lettres, bases d'ADN, etc.
- Soit $S = (s_i)$ une séquence, on appelle sous-séquence de S toute suite extraite X de S.
  Pour mémoire, si X est de longueur p, on peut écrire $X = (s_{i_1}, \ldots, s_{i_p})$ avec $i_1 < i_2 < \cdots < i_p$.
- On appelle sous-séquence commune de S et T, une séquence X qui est à la fois une sous-séquence de S et de T.

Problème

Etant données deux séquences S et T, donner la plus grande longueur d'une sous-séquence commune à S et T puis donner une sous-séquence commune de cette longueur maximale (en abrégé PLSC).

La fonction d'évaluation est donc la longueur d'une sous-séquence commune et on cherche un maximum.

1.3.1 Etape 1 : étude de la structure d'une solution optimale

Propriété

Notation : pour toute séquence S, on note S_i la troncature de la séquence S après l'indice i.
On a donc $S_i = (s_1, \ldots, s_i)$. On peut étendre cette définition pour $i = 0$ par $S_0 = \emptyset$.

Soient $S = (s_1, \ldots, s_n)$ et $R = (r_1, \ldots, r_m)$ deux séquences et $X = (x_1, \ldots, x_k)$ **une PLSC** de S et R.
On a la propriété suivante, vraie également pour $n = 1$ ou $m = 1$:

- Si $s_n = r_m$ alors $x_k = s_n = r_m$ et X_{k-1} est une PLSC de S_{n-1} et R_{m-1}.
- Sinon si $x_k \neq s_n$ alors X est une PLSC de S_{n-1} et R.
- Sinon $x_k \neq r_m$ et X est une PLSC de S et R_{m-1}.

Preuve :

- Cas $s_n = r_m$. Démonstration par l'absurde.
  Supposons que $x_k \neq s_n$. X ne se termine pas par le dernier caractère de S donc X est une sous-séquence de S_{n-1}. De même pour R, donc X est une sous-séquence commune de S_{n-1} et R_{m-1}.
  Mais il suit que $X + (s_n)$ (+ symbolise la concaténation) est une sous-séquence commune à S et R strictement plus longue que S. (*Contradiction*)
  Conclusion : X se termine par le dernier caractère de S et de R, c.a.d. $x_k = s_n = r_m$.
  Montrons maintenant par l'absurde que X_{k-1} est une PLSC de S_{n-1} et R_{m-1}.
  Supposons l'existence d'une sous-séquence Y de S_{n-1} et R_{m-1} strictement plus longue que X_{k-1}. Alors $Y + (s_n)$ est une sous-séquence de S et R strictement plus longue que X (contradiction).
- Cas $s_n \neq r_m$ et $x_k \neq s_n$.
  D'après un raisonnement fait précédemment, comme X ne se termine pas par le dernier caractère de S, X est une sous-séquence commune de S_{n-1} et R.
  Supposons que X ne soit pas une PLSC de S_{n-1} et R. Alors il existe une sous-séquence Y de S_{n-1} et R strictement plus longue que X. Comme une sous-séquence commune de S_{n-1} et R est aussi une sous-séquence commune de S et R, on obtient une contradiction avec X PLSC de S et R.
- Autres cas : on a alors $s_n \neq r_m$ et $x_k = s_n$. Donc $x_k \neq r_m$.
  Il suit, par le même raisonnement que dans le cas précédent, qu'on a X est une PLSC de S et R_{m-1}.

1.3.2 Etape 2 : écriture d'une relation récursive sur la longueur d'une PLSC

D'après la propriété précédente, on identifie l'ensemble des sous-problèmes à étudier. Il s'agit de la recherche de PLSC communes à S_i et R_j pour $i \in [0, n]$ et $j \in [0, m]$.

On note $l(i,j)$ la longueur d'une solution de ce sous-problème.

Relation récursive sur les sous-solutions optimales

- Si $i = 0$ ou $j = 0$ alors $l(i,j) = 0$.
- Sinon si $s[i] = r[j]$ alors $l(i,j) = l(i-1,j-1) + 1$.
- Sinon $l(i,j) = \max(l(i-1,j), l(i,j-1))$.

Preuve :
Le cas $i = 0$ ou $j = 0$ s'explique parce qu'une des séquences au moins est vide.

Dans les cas 2 et 3, $i \geqslant 1$ et $j \geqslant 1$. On peut donc appliquer la propriété de l'étape 1. On note $X = (x_1 \dots x_k)$ une PLSC quelconque de S_i et R_j. Par définition de $l(i,j)$, on a $k = l(i,j)$.

Si $s_i = r_j$, alors X se termine par $x_k = s_i = r_j$ et X_{k-1} est une PLSC de S_{i-1} et R_{j-1}. On en conclut que $k - 1 = l(i-1,j-1)$ donc $l(i,j) = l(i-1,j-1) + 1$.

Sinon $x_k \neq s_i$ ou $x_k \neq r_j$, donc X est une PLSC de S_{i-1} et R_j ou une PLSC de S_i et R_{j-1}.

Il suit que $l(i,j) = l(i-1,j)$ ou $l(i,j) = l(i,j-1)$

Par conséquent $l(i,j) \leqslant \max(l(i-1,j), l(i,j-1)$ (1).

Or les sous-séquences communes de S_{i-1} et R_j est un sous-ensemble des sous-séquences communes de S_i et R_j donc pour les longueurs maximales, on a $l(i-1,j) \leqslant l(i,j)$.

De même pour les sous-séquences communes de S_i et R_{j-1} donc $l(i,j-1) \leqslant l(i,j)$.

On en déduit que $l(i,j) \geqslant \max(l(i-1,j), l(i,j-1))$ (2).

(1) et (2) montrent $l(i,j) = \max(l(i-1,j), l(i,j-1))$ (CQFD).

A partir de cette relation, on pourrait écrire un **algorithme récursif naïf** :

Algorithme 3.

```
fonction longueur_plsc_rec(S,R,i,j):
    # cas terminal : sequence vide c.a.d. i ou j vaut 0
    if i == 0 or j == 0:
        return 0
    if S[i]==R[j]:
        return longueur_plsc_rec(S,R,i-1,j-1)+1
    else:
        return max(longueur_plsc_rec(S,R,i-1,j),longueur_plsc_rec(S,R,i,j-1))
```

L'appel initial serait `longueur_plsc_rec(S,R,longueur(S),longueur(R))`.

L'étude de la complexité montre qu'elle est **exponentielle** en $O(2^p)$, en notant $p = \min(longueur(S), longueur(R))$.

Preuve :
Le pire des cas est pour deux séquences dont les valeurs sont dans des ensembles disjoints.
Chaque appel non terminal génère alors 2 appels récursifs.
Si on étudie dans ce cas l'arbre des appels récursifs, à chaque niveau la somme i+j décroit de 1 donc la hauteur de l'arbre est au maximum $longueur(S) + longueur(R) - 1$. Il y a -1 car lorsque $i + j = 1$, alors $i = 0$ ou $j = 0$ donc on a une feuille.
De plus la hauteur minimale d'une feuille est $p = \min(longueur(S), longueur(R))$.
Le nombre d'appels (qui est égal au nombre de noeuds) est donc au moins $2^{p+1} - 1$ c.a.d. le nombre de feuilles d'un arbre binaire complet de hauteur p.
Conclusion : $O(2^p)$ est une borne optimale de la complexité.

Dans une étude plus fine, on pourrait étudier le cas où une des chaînes est de longueur constante (donc le minimum des longueurs est constant).

Cette complexité catastrophique est due à la répétition des appels récursifs pour les mêmes paramètres.

1.3.3 Etape 3 : algorithme bottom-up de calcul d'une solution optimale

A partir de la relation récursive précédente, on peut construire de manière bottom-up les sous-solutions à partir des sous-problèmes minimaux (ici ils sont vides).

On obtient l'algorithme suivant (les tableaux sont indicés à partir de 0, les séquences à partir de l'indice 1) :

Algorithme 4.

```
fonction longueur_plsc(S,R)
    créer deux tableaux d'entiers l et c de dimension (n+1) x (m+1)
    # en sortie, l(i,j) stockera la PLSC de Si et Rj
    # c(i,j) stockera 1, 2 ou 3 selon le cas dans la relation récursive 1.3.2

    #initialisation pour des sous-problème de taille 0
    pour i de 0 à n faire
        l[i,0]=0
    pour j de 1 à m faire
        l[0,j]=0
    pour i de 1 à n faire
        pour j de 1 à m faire
        #calcul de la longueur d'une PLSC de S_i et R_j
            si S[i]==R[j] alors
                l[i,j]=l[i-1,j-1]+1
                c[i,j]=1
            sinon si l[i-1,j]>=l[i,j-1] alors
                l[i,j]=l[i-1,j]
                c[i,j]=2
            sinon
                l[i,j]=l[i,j-1]
                c[i,j]=3

    retourner l,c
```

Cet algorithme a une complexité temporelle en $O(n.m)$.
Preuve immédiate en calculant le nombre d'itération de la boucle sur j.

Pour construire une PLSC correspondant à la longueur optimale, une fois exécutée la fonction précédente, on peut suivre un **algorithme récursif** :

Algorithme 5.

```
fonction plsc_rec(S,R,c,i,j)
    si i==0 ou j==0 alors
        retourner séquence vide.

    si c[i,j]==1 alors
        retourner plsc(S,R,c,i-1,j-1) + (S[i])  # + symbolise la concaténation

    si c[i,j]==2 alors
        retourner plsc(S,R,c,i-1,j)
    #sinon
    retourner plsc(S,R,c,i,j-1)
```

Cet algorithme a une complexité temporelle en $O(n+m)$.

Preuve :
Chaque exécution de la fonction effectue au plus un appel récursif, et en dehors de cet appel, les opérations sont à temps constant.
A chaque appel récursif, $i+j$ diminue de 1 ou 2. Dans le pire des cas (séquences S et R à valeurs dans des ensembles disjoints), $i+j$ ne diminue que de 1 à chaque fois. Comme $i+j$ vaut $n+m$ au premier appel, dans le pire des cas, il faut $n+m-1$ appels pour atteindre $i=0$ ou $j=0$.

ou un **algorithme itératif** :

Algorithme 6.

```
fonction plsc_iter(S,R,c)
    #initialisation
    i=n
    j=m
    lcs=créer une séquence vide
    #parcours de la matrice des choix
    tant que i>0 et j>0 faire
        si c[i][j]==1 alors
            lcs= (S[i-1]) + lcs    # + symbolise la concaténation
            i=i-1
            j=j-1
        sinon si c[i][j]==2 alors
            i=i-1
        sinon
            j=j-1
return lcs
```

Cet algorithme a une complexité temporelle en $O(n+m)$.

Preuve :
A chaque itération, $i+j$ diminue de 1 ou 2. Dans le pire des cas (séquences S et R à valeurs dans des ensembles disjoints), $i+j$ ne diminue que de 1 à chaque fois. Comme $i+j$ est initialisé à $n+m$, dans le pire des cas, il faut $n+m-1$ itérations pour atteindre $i=0$ ou $j=0$.

NB : il est possible de diminuer la complexité spatiale de l'algorithme de plusieurs manières : éviter l'utilisation du tableau c, réduire la taille de la matrice l à deux lignes...

2 Algorithmes gloutons

2.1 Principe

Comme la programmation dynamique, les algorithmes gloutons s'appliquent à des problèmes d'optimisation où il s'agit de faire une succession de choix pour trouver une solution optimale.

Idée générale d'un alorithme glouton

L'idée générale est d'effectuer chaque choix sans connaître le résultat des choix suivants, on fait donc un choix optimum local.

Expliquons un peu plus. Dans les deux problèmes de programmation dynamique que nous avons étudiés, nous avions une suite de choix à faire pour déterminer une solution optimale.

Le premier choix à faire nécessitait de connaître les sous-solutions optimales pour comparer entre elles les solutions découlant de ce premier choix.

Supposons que l'analyse d'une solution optimale montre qu'une telle comparaison n'est pas nécessaire. On peut alors faire le premier choix sans connaître les sous-solutions optimales, c'est-à-dire les choix optimum suivants.

Adaptation de la programmation dynamique à la méthode gloutonne

1) Etudier la structure d'une solution optimale en fonction de découpages possibles en sous-problèmes.
2) A partir de l'étape précédente, écrire une relation récursive sur la valeur optimale de f.
3) Démontrer qu'à chaque étape de la récursivité, l'un des choix optimaux est un choix glouton.
4) Démontrer qu'après un choix glouton, il n'y a plus qu'un seul sous problème à résoudre.

2.2 Application : problème de choix d'activités

Supposons un ensemble de n activités $A = \{a_1, \ldots, a_n\}$ qui ont besoin d'une même ressource par exemple une salle de conférence. Cette ressource ne peut être utilisée que par une activité à la fois.

Chaque activité a_i se caractérise par une heure de début d_i et une heure de fin $f_i > d_i$ qui ne sont pas modifiables.

Le problème du choix d'activités consiste à déterminer un sous-ensemble de S de taille maximale d'activités compatibles pour utiliser la ressource commune. En bref, on veut planifier un nombre maximum d'activités.

2.2.1 Structure d'une solution optimale S

Propriété

Soit S une solution optimale et soit $a_k \in S$.
On note $S'_k = \{a_i \in S | f_i \leqslant d_k\}$ et $S''_k = \{a_i \in S | d_i \geqslant f_k\}$.
Alors $S = S'_k \cup \{a_k\} \cup S''_k$ est une partition de S vérifiant :

- S'_k est une solution optimale pour le sous-ensemble des activités se terminant avant le début de a_k.
- S''_k est une solution optimale pour le sous-ensemble des activités commençant après la fin de a_k.

Preuve :
Il est évident que les autres activités de S se terminent avant a_k ou commencent après a_k. On peut donc partitionner S en $S'_k \cup \{a_k\} \cup S''_k$.

De plus S'_k est une solution optimale du sous-problème réduit aux activités se terminant avant a_k.
Supposons qu'il y a un ensemble T, de cardinal strictement supérieur à celui de S'_k, d'activités compatibles

se terminant avant a_k, alors $T \cup \{a_k\} \cup S''$ serait une solution globale de cardinal strictement plus grand que S (contradiction).

On fait le même raisonnement pour S''_k.

2.2.2 Relation récursive

On définit les sous-ensembles $A_{i,j}$ des activités de A se déroulant entre a_i et a_j.

En effectuant de manière récursive un découpage d'une solution optimale S, on obtient des sous-solutions optimales $S_{i,j}$ pour les sous-problèmes $A_{i,j}$.

Cette notation pose un problème pour désigner les ensembles comme S'_k et S''_k du paragraphe précédent.

On étend donc les notations $A_{i,j}$ et $S_{i,j}$ en introduisant deux activités fictives :

- a_0 de fin $f_0 < \min\{d_i | i \in [1,n]\}$,
- a_{n+1} de début $d_{n+1} > \max\{f_i | i \in [1,n]\}$.

De cette manière, dans le paragraphe précédent, on a $S = S_{0,n+1}$, $S'_k = S_{0,k}$ et $S''_k = S_{k,n+1}$.

Enfin, on note $m_{i,j}$ le cardinal d'une solution optimale du sous-problème $A_{i,j}$.

Avec ses notations, on a la **propriété** :

- si $A_{i,j} = \emptyset$ alors $m_{i,j} = 0$.
- sinon $m_{i,j} = 1 + \max\{m_{i,k} + m_{k,j} | a_k \in A_{i,j}\}$.

Preuve :

- Le cas $A_{i,j} = \emptyset$ est évident.
- si $A_{i,j} \neq \emptyset$, alors notons $m = 1 + \max\{m_{i,k} + m_{k,j} | a_k \in A_{i,j}\}$.
  La définition de m a un sens car l'ensemble n'est pas vide.
  Soit $S_{i,j}$ une solution optimale pour $A_{i,j}$, $S_{i,j}$ n'est pas vide car $A_{i,j}$ ne l'est pas. On choisit $a_k \in S_{i,j}$.
  D'après la propriété de la section précédente, $m_{i,j} = \text{card}(S_{i,j}) = 1 + m_{i,k} + m_{k,j}$.
  Comme $a_k \in A_{i,j}$, il suit que $m_{i,j} \leqslant \max\{1 + m_{i,k} + m_{k,j} | a_k \in A_{i,j}\} = 1 + m$.
  Montrons que $m_{i,j} \geqslant 1 + m$. Notons a_k une activité pour laquelle le maximum m est atteint, c.a.d. $m = 1 + m_{i,k} + m_{k,j}$.
  Il existe donc deux solutions $S_{i,k}$ pour $A_{i,k}$ et $S_{k,j}$ pour $A_{k,j}$ de cardinal respectif $m_{i,k}$ et $m_{k,j}$.
  $S = \{a_k\} \cup S_{i,k} \cup S_{k,j}$ est une solution pour $A_{i,j}$ et son cardinal est $1 + m_{i,k} + m_{k,j} = 1 + m$ car les trois ensembles sont deux à deux disjoints.
  Il suit que le cardinal d'une solution optimale pour $A_{i,j}$ vérifie $m_{i,j} \geqslant 1 + m$.

On peut en déduire un **algorithme récursif naïf**

Algorithme 7.

```
fonction choix_activite_naif(d,f,debut,fin)
    #effectue un choix optimal pour l'intervalle de temps [debut,fin]
    n = longueur(A)
    max = 0     # nombre maximum d'activités compatibles
    pour k de 1 à n faire
        si d[k]>=debut et f[k]<=fin alors
            m = 1 + choix_activite_naif(d,f,debut,d[k]) + choix_activite_naif(d,f,f[k],fin)
            si m > max alors
                max = m
    retourner max
```

2.2.3 Existence d'un choix glouton optimum

Propriété

En notant a_p l'activité qui se termine en premier dans $A_{i,j}$, on a : $m_{i,j} = 1 + m_{p,j}$.

Preuve :
Constatons d'abord que $A_{i,p} = \emptyset$ car pour tout $a_k \in A_{i,j}$ distinctes de a_p, $f_k \geqslant f_p > d_p$ donc a_k n'est pas programmable avant a_p. Il suit que $m_{i,p} = 0$.

Soit $S_{i,j}$ une solution optimale pour $A_{i,j}$. Si on note a_q l'activité qui se termine en premier dans $S_{i,j}$.
On a : $S_{i,j} = \{a_q\} \cup S_{q,j}$ où $S_{q,j}$ est le sous-ensemble de $S_{i,j}$ des activités commençant après a_q.
On sait d'après la propriété 2.2.1, que $S_{q,j}$ est une solution optimale de $A_{q,j}$.
Donc $m_{i,j} = 1 + m_{q,j} \leqslant 1 + m_{p,j}$ car $A_{q,j} \subset A_{p,j}$ puisque $d_p \leqslant d_q$.

Inversement, si on choisit une solution optimale pour $A_{p,j}$, à laquelle on ajoute a_p, on obtient une solution pour $A_{i,j}$. Donc $m_{p,j} + 1 \leqslant m_{i,j}$. (CQFD)

Conséquence immédiate : **choix glouton**

Choisir dans $A_{i,j}$ l'activité qui se termine en premier est toujours un choix optimal.

2.2.4 Algorithme glouton

Il suit que l'on peut résoudre le problème du choix d'activité **en classant A par ordre croissant de fin des activités** puis en appliquant l'**algorithme** suivant :

Algorithme 8.

```
fonction choix_planning_glouton(a,d,f)
    n=longueur(a)
    s=(a[1])                # choix glouton pour la première activité
    last=1                  # indice de la dernière activité planifiée
    pour i de 2 à n faire
        si d[i]>=f[last] alors
            s=s+(a[i])     # choix glouton
            last=i           # mise à jour de l'indice last
    retourner s
```

Méthode générale de développement d'un algorithme glouton

1) Transformation du problème d'optimisation en un problème dans lequel on fait un choix à la suite duquel il n'y a qu'un problème à résoudre.
2) Démontrer qu'à chaque étape de la récursivité, l'un des choix optimaux est un choix glouton.
3) Démontrer qu'après un choix glouton, on peut combiner une solution optimale du sous-problème restant et le choix glouton pour déterminer la solution globale optimale.