Programmation dynamique

Jean-Pierre Becirspahic
Lycée Louis-Le-Grand

Un inconvénient de la programmation récursive

Calcul des coefficients binomiaux

```
let rec binom = fun
  | n 0 -> 1
  | n p when n = p -> 1
  | n p -> binom (n-1) (p-1) + binom (n-1) p ;;
```

Un inconvénient de la programmation récursive

Calcul des coefficients binomiaux

```
let rec binom = fun
  | n 0 -> 1
  | n p when n = p -> 1
  | n p -> binom (n-1) (p-1) + binom (n-1) p ;;
```

Si $C(n,p)$ est le nombre d'additions réalisées par cette fonction,

$$C(n,0) = C(n,p) = 0$$
$$\forall p \in [\![1, n-1]\!], \quad C(n,p) = C(n-1,p-1) + C(n-1,p) + 1$$

On prouve alors par induction que $C(n,p) = \binom{n}{p} - 1$.

Un inconvénient de la programmation récursive

Calcul des coefficients binomiaux

```
let rec binom = fun
  | n 0 -> 1
  | n p when n = p -> 1
  | n p -> binom (n-1) (p-1) + binom (n-1) p ;;
```

Si $C(n,p)$ est le nombre d'additions réalisées par cette fonction,

$$C(n,0) = C(n,p) = 0$$
$$\forall p \in [\![1, n-1]\!], \quad C(n,p) = C(n-1,p-1) + C(n-1,p) + 1$$

On prouve alors par induction que $C(n,p) = \binom{n}{p} - 1$.

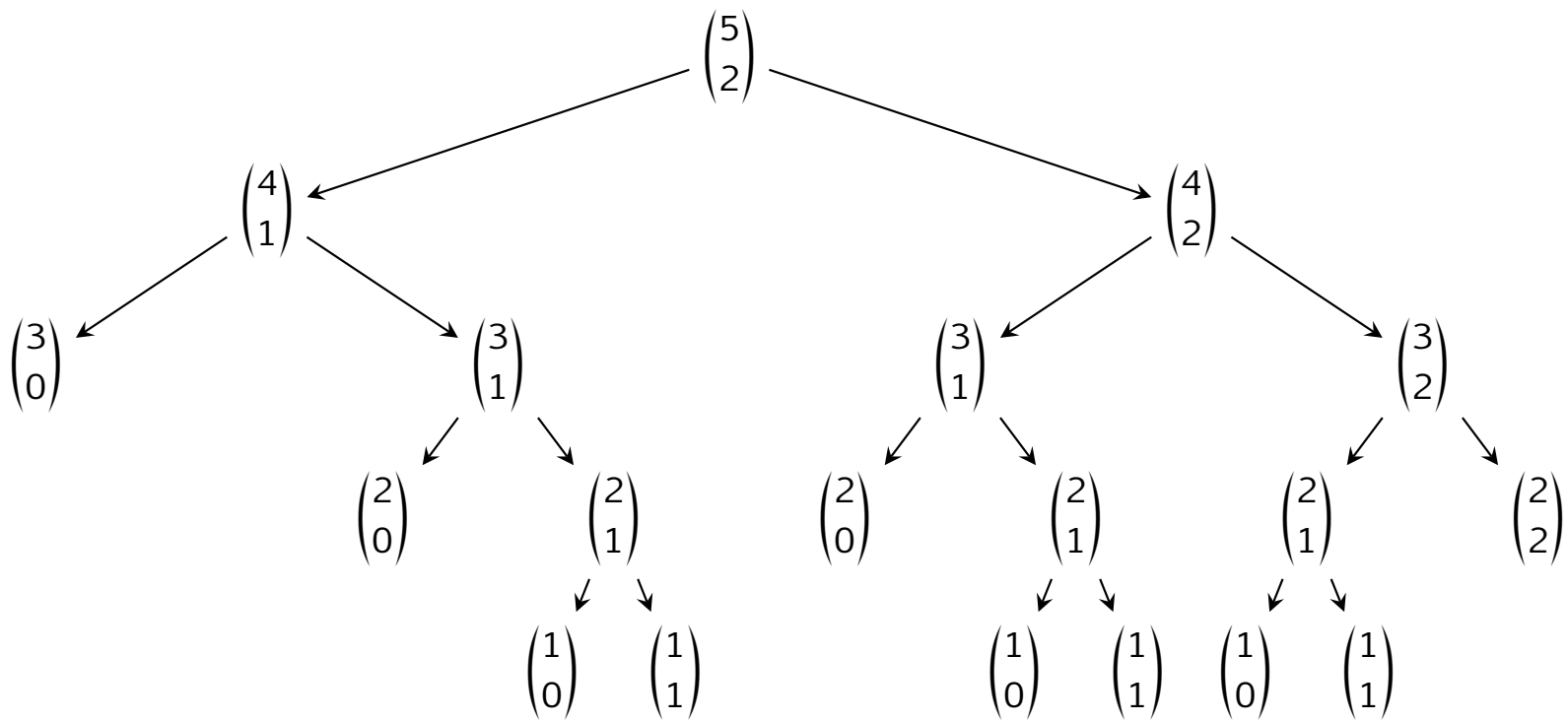
D'après la formule de Stirling, $\binom{2n}{n} \sim \frac{4^n}{\sqrt{\pi n}}$; le calcul de $\binom{2n}{n}$ par cette fonction est de complexité exponentielle.

Un inconvénient de la programmation récursive

Calcul des coefficients binomiaux

```
let rec binom = fun
  | n 0 -> 1
  | n p when n = p -> 1
  | n p -> binom (n-1) (p-1) + binom (n-1) p ;;
```

Arbre de calcul : les branches sont en interaction.

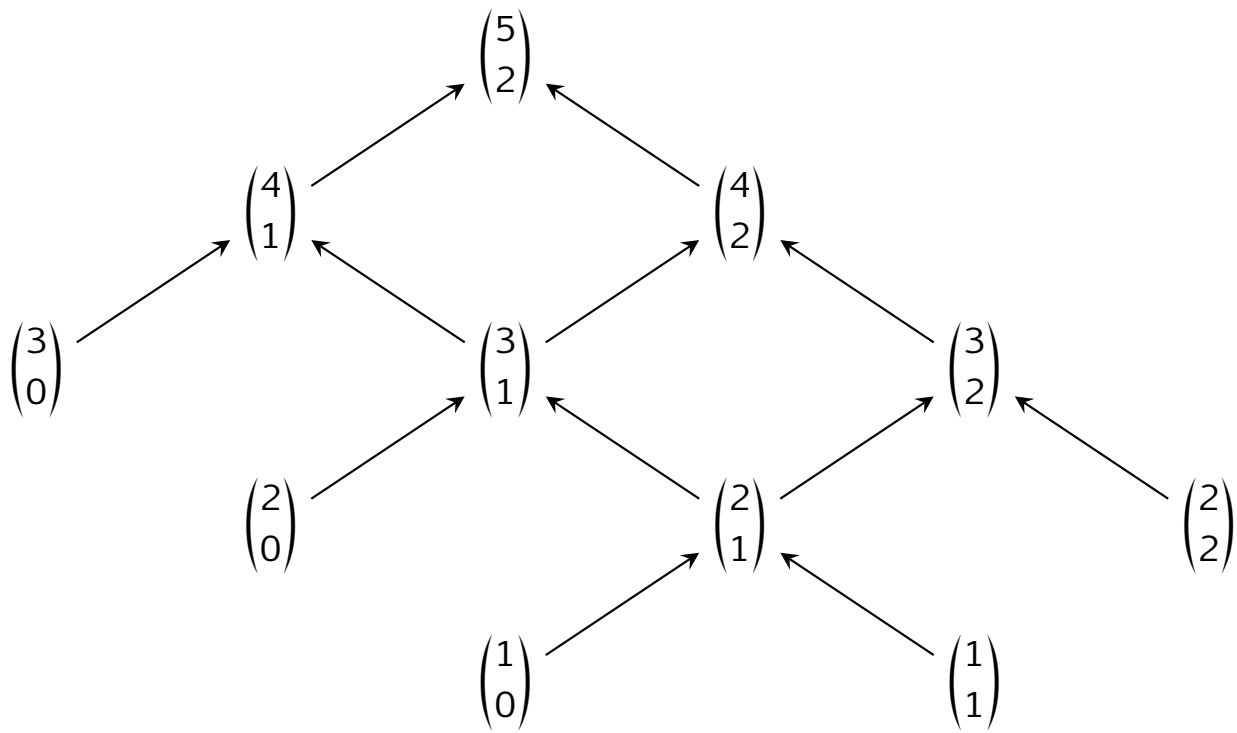


Un inconvénient de la programmation récursive

Calcul des coefficients binomiaux

```
let rec binom = fun
  | n 0 -> 1
  | n p when n = p -> 1
  | n p -> binom (n-1) (p-1) + binom (n-1) p ;;
```

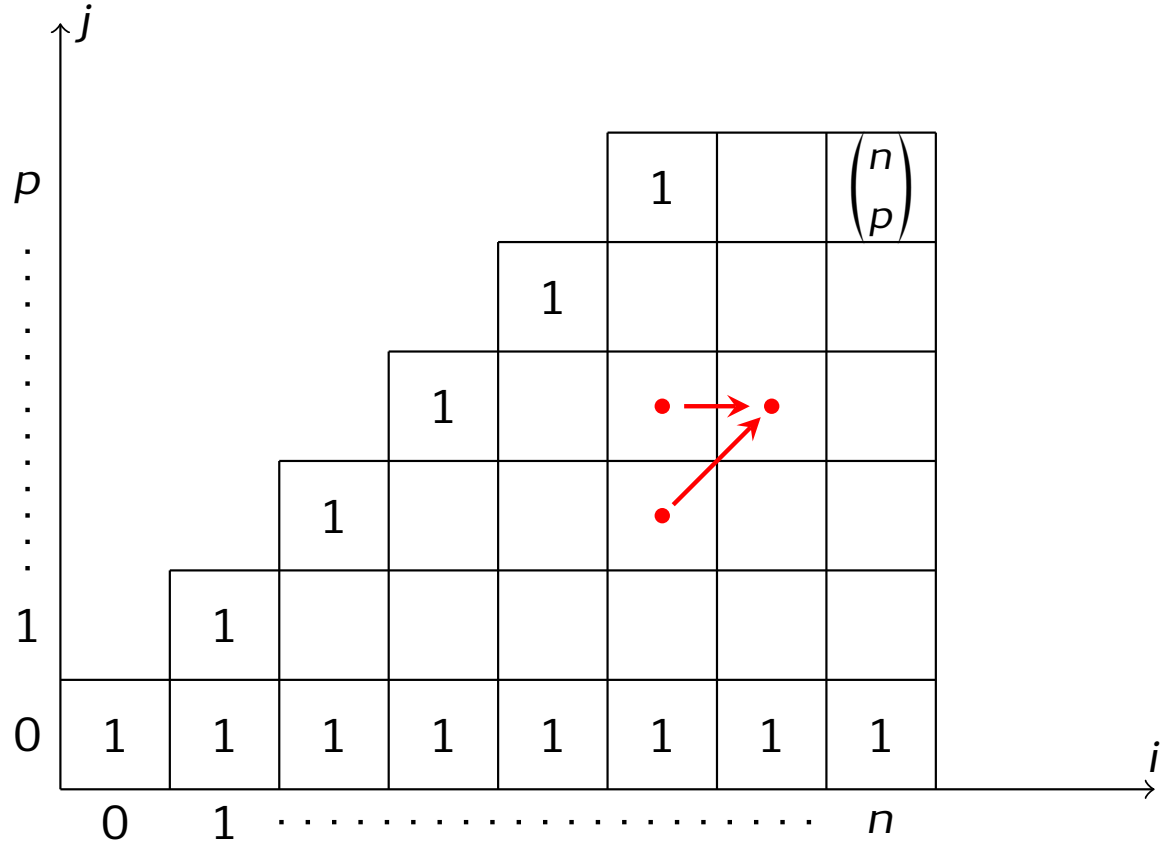
Solution : calculer les différentes valeurs requises du bas vers le haut en les mémorisant.

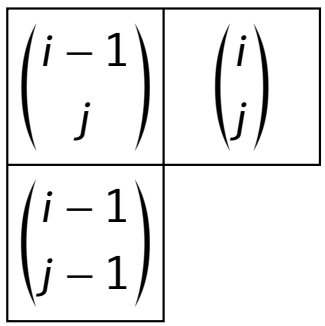


Un inconvénient de la programmation récursive

Calcul des coefficients binomiaux

On utilise un tableau bi-dimensionnel $(n+1) \times (p+1)$ pour stocker les coefficients binomiaux.





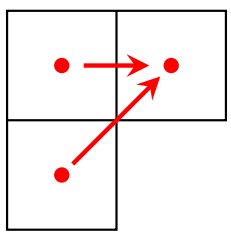
Un inconvénient de la programmation récursive

Calcul des coefficients binomiaux

On utilise un tableau bi-dimensionnel $(n+1) \times (p+1)$ pour stocker les coefficients binomiaux.

```
let binom n p =
  let c = make_matrix (n+1) (p+1) 1 in
  for i = 2 to n do
    for j = 1 to min (i-1) p do
      c.(i).(j) <- c.(i-1).(j) + c.(i-1).(j-1)
    done ;
  done ;
  c.(n).(p) ;;
```

Attention à bien choisir l'ordre dans lequel les cases de ce tableau sont remplies !



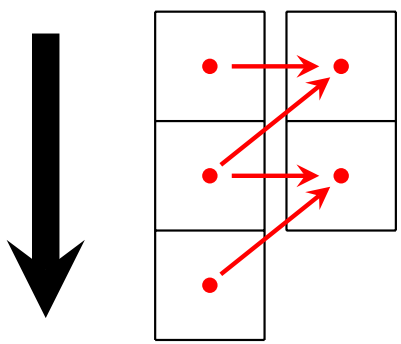
Coût spatial et temporel en $\Theta(np)$.

Un inconvénient de la programmation récursive

Calcul des coefficients binomiaux

On peut se contenter d'un tableau uni-dimensionnel, mais il faut faire encore plus attention aux relations de dépendances :

```
let binom n p =
  let c = make_vect (p+1) 1 in
  for i = 2 to n do
    for j = min (i-1) p downto 1 do
      c.(j) <- c.(j) + c.(j-1)
    done ;
  done ;
  c.(p) ;;
```

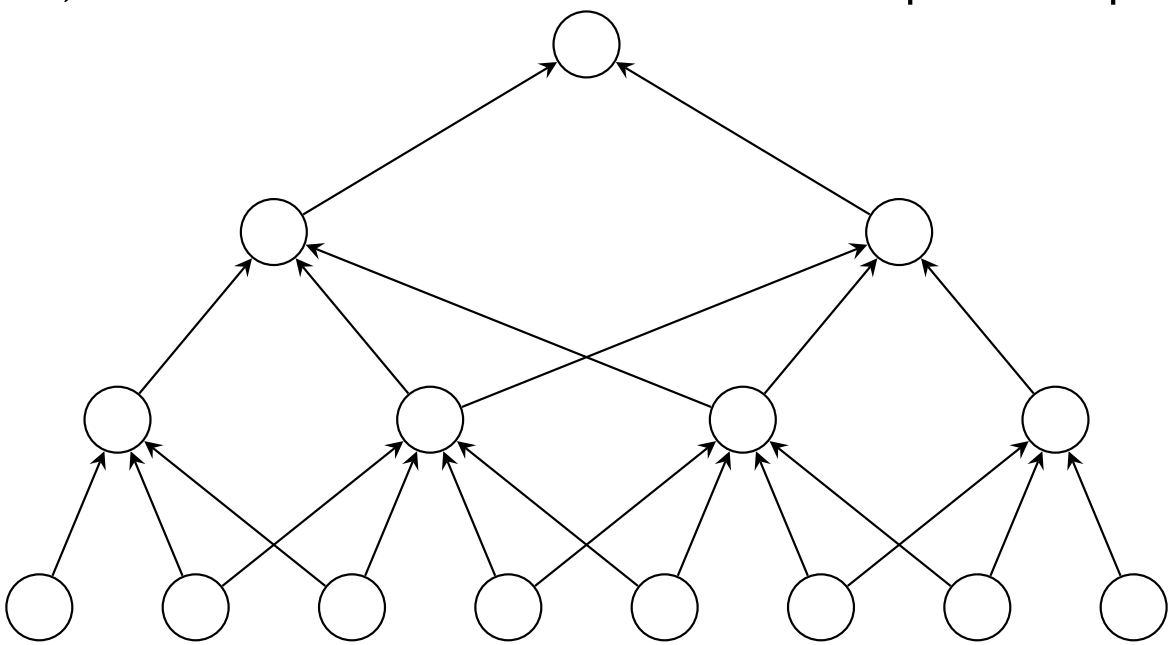


Coût spatial en $\Theta(p)$, coût temporel en $\Theta(np)$.

Programmation dynamique et gloutonne

Résolution d'un problème par programmation dynamique :

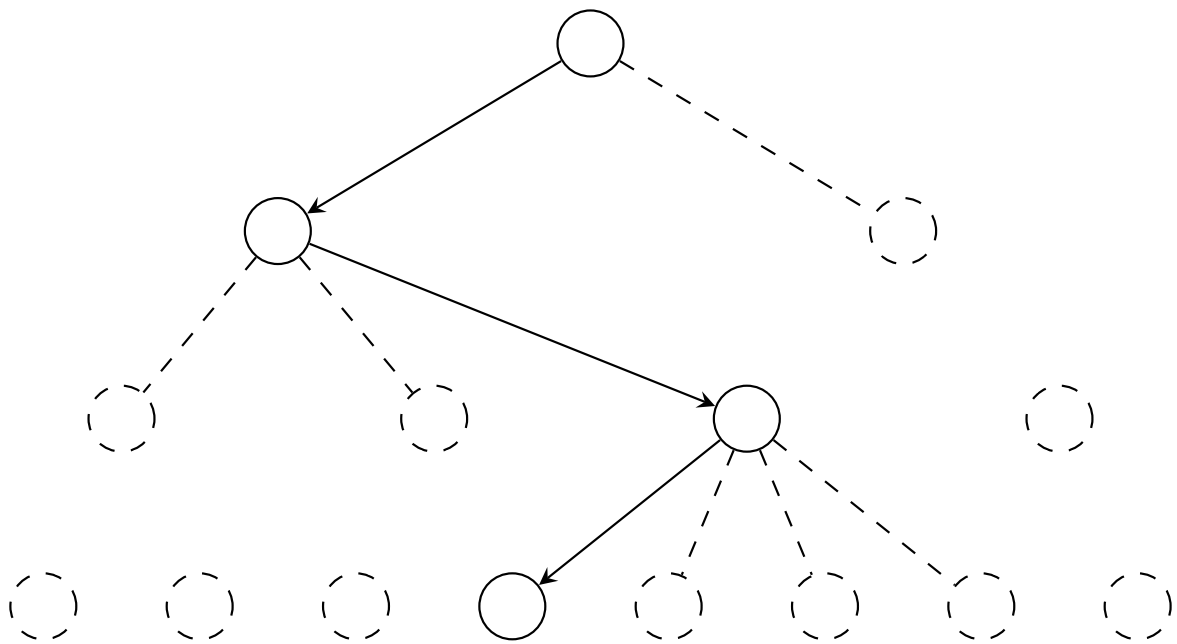
- obtention d'une relation de récurrence liant la solution du problème global à celles de problèmes locaux *non indépendants* ;
- initialisation d'un tableau à l'aide des conditions initiales de la relation obtenue au point précédent ;
- remplissage du tableau en résolvant les problèmes locaux par taille croissante, à l'aide de la relation obtenue au premier point.



Programmation dynamique et gloutonne

Résolution d'un problème par programmation gloutonne :
La stratégie gloutonne consiste à choisir à partir du problème global un problème local et un seul en suivant une heuristique (une stratégie permettant de faire un choix rapide mais pas nécessairement optimal).

Bien entendu, on ne peut en général garantir que la stratégie gloutonne détermine la solution optimale.



Programmation dynamique et gloutonne

Exemple du rendu de monnaie

Comment rendre la monnaie en utilisant un nombre minimal de pièces et de billets ?

Programmation dynamique et gloutonne

Exemple du rendu de monnaie

Comment rendre la monnaie en utilisant un nombre minimal de pièces et de billets ?

Heuristique gloutonne : rendre la pièce ou de billet de valeur maximale possible, et ce tant qu'il reste quelque chose à rendre.
Par exemple, la somme de 48€ est décomposée comme suit :

$$48€ = 20€ + 20€ + 5€ + 2€ + 1€$$

Programmation dynamique et gloutonne

Exemple du rendu de monnaie

Comment rendre la monnaie en utilisant un nombre minimal de pièces et de billets ?

Heuristique gloutonne : rendre la pièce ou de billet de valeur maximale possible, et ce tant qu'il reste quelque chose à rendre.
Par exemple, la somme de 48€ est décomposée comme suit :

$$48€ = 20€ + 20€ + 5€ + 2€ + 1€$$

Il est possible de prouver que dans le cadre du système monétaire européen, l'heuristique gloutonne fournit toujours la solution optimale.

Programmation dynamique et gloutonne

Exemple du rendu de monnaie

Comment rendre la monnaie en utilisant un nombre minimal de pièces et de billets ?

Heuristique gloutonne : rendre la pièce ou de billet de valeur maximale possible, et ce tant qu'il reste quelque chose à rendre.
Par exemple, la somme de 48€ est décomposée comme suit :

$$48€ = 20€ + 20€ + 5€ + 2€ + 1€$$

Il est possible de prouver que dans le cadre du système monétaire européen, l'heuristique gloutonne fournit toujours la solution optimale.

Ce n'était pas le cas pour le système britannique d'avant 1971 qui utilisait les multiples suivant du *penny* : 1, 3, 6, 12, 24, 30.
Avec ce système, l'heuristique gloutonne donne :

$$48p = 30p + 12p + 6p$$

alors que la décomposition optimale est $48p = 24p + 24p$.

Programmation dynamique et gloutonne

Exemple du rendu de monnaie

Comment rendre la monnaie en utilisant un nombre minimal de pièces et de billets ?

Heuristique gloutonne : rendre la pièce ou de billet de valeur maximale possible, et ce tant qu'il reste quelque chose à rendre.
Par exemple, la somme de 48€ est décomposée comme suit :

$$48€ = 20€ + 20€ + 5€ + 2€ + 1€$$

Programmation dynamique : si $f(n, S)$ est le nombre minimal de pièces pour décomposer n à l'aide de S alors

$$f(n,S) = \begin{cases} \min\Big(1 + f(n-c,S), f(n, S \setminus \{c\})\Big) & \text{si } c \leqslant n \\ f(n, S \setminus \{c\}) & \text{sinon} \end{cases}$$

On utilise un tableau bi-dimensionnel pour stocker les valeurs de $f(n, S)$.

Problème du sac à dos

Étant donnés n objets de valeurs $c_1, \ldots, c_n$ et de poids $w_1, \ldots, w_n$, comment remplir un sac maximisant la valeur $\sum_{i \in I} c_i$ en respectant la contrainte $\sum_{i \in I} w_i \leqslant W_{\text{max}}$?

Problème du sac à dos

Étant donnés n objets de valeurs $c_1, \ldots, c_n$ et de poids $w_1, \ldots, w_n$, comment remplir un sac maximisant la valeur $\sum_{i \in I} c_i$ en respectant la contrainte $\sum_{i \in I} w_i \leqslant W_{\max}$?

On note $f(n, W_{\max})$ la valeur maximale qu'il est possible d'atteindre avec n objets :

$$f(n, W_{\max}) = \begin{cases} \max\Big(c_n + f(n-1, W_{\max} - w_n), f(n-1, W_{\max})\Big) & \text{si } w_n \leqslant W_{\max} \\ f(n-1, W_{\max}) & \text{sinon} \end{cases}$$

Problème du sac à dos

Étant donnés n objets de valeurs $c_1, \ldots, c_n$ et de poids $w_1, \ldots, w_n$, comment remplir un sac maximisant la valeur $\sum_{i \in I} c_i$ en respectant la contrainte $\sum_{i \in I} w_i \leqslant W_{\max}$?

On note $f(n, W_{\max})$ la valeur maximale qu'il est possible d'atteindre avec n objets :

$$f(n, W_{\max}) = \begin{cases} \max\Big(c_n + f(n-1, W_{\max} - w_n), f(n-1, W_{\max})\Big) & \text{si } w_n \leqslant W_{\max} \\ f(n-1, W_{\max}) & \text{sinon} \end{cases}$$

Mise en œuvre : on utilise un tableau bi-dimensionnel de taille $(n+1) \times (W_{\max}+1)$ destiné à contenir les valeurs de $f(k, w)$ pour $k \in [\![0, n]\!]$ et $w \in [\![0, W_{\max}]\!]$.

```
let sacados c w wmax =
  let n = vect_length c in
  let f = make_matrix (n+1) (wmax+1) 0 in
  for k = 0 to n-1 do
    for x = 0 to wmax do
      if w.(k) <= x
        then f.(k+1).(x) <- max (c.(k)+f.(k).(x-w.(k))) f.(k).(x)
        else f.(k+1).(x) <- f.(k).(x)
    done
  done ;
  f.(n).(wmax) ;;
```

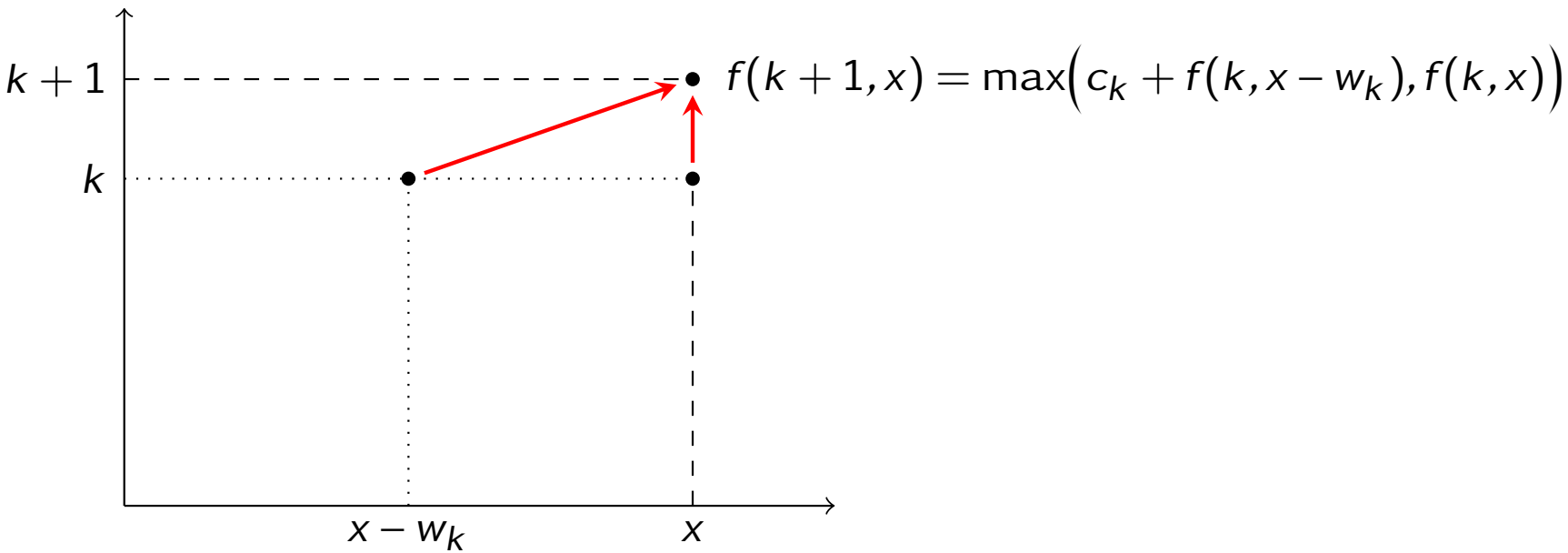
Problème du sac à dos

Étant donnés n objets de valeurs $c_1, \ldots, c_n$ et de poids $w_1, \ldots, w_n$, comment remplir un sac maximisant la valeur $\sum_{i \in I} c_i$ en respectant la contrainte $\sum_{i \in I} w_i \leqslant W_{\max}$?

On note $f(n, W_{\max})$ la valeur maximale qu'il est possible d'atteindre avec n objets :

$$f(n, W_{\max}) = \begin{cases} \max\Big(c_n + f(n-1, W_{\max} - w_n), f(n-1, W_{\max})\Big) & \text{si } w_n \leqslant W_{\max} \\ f(n-1, W_{\max}) & \text{sinon} \end{cases}$$

Ordre de dépendance des calculs :



Problème du sac à dos

Étant donnés n objets de valeurs $c_1, \ldots, c_n$ et de poids $w_1, \ldots, w_n$, comment remplir un sac maximisant la valeur $\sum_{i \in I} c_i$ en respectant la contrainte $\sum_{i \in I} w_i \leqslant W_{\max}$?

On note $f(n, W_{\max})$ la valeur maximale qu'il est possible d'atteindre avec n objets :

$$f(n, W_{\max}) = \begin{cases} \max\Big(c_n + f(n-1, W_{\max} - w_n), f(n-1, W_{\max})\Big) & \text{si } w_n \leqslant W_{\max} \\ f(n-1, W_{\max}) & \text{sinon} \end{cases}$$

Avec un tableau uni-dimensionnel :

```
let sacados c w wmax =
  let n = vect_length c in
  let f = make_vect (wmax+1) 0 in
  for k = 0 to n-1 do
    for x = wmax downto 0 do
      if w.(k) <= x && c.(k)+f.(x-w.(k)) > f.(x)
        then f.(x) <- c.(k)+f.(x-w.(k))
    done
  done ;
  f.(wmax) ;;
```

Coût temporel : $\Theta(nW_{\max})$ coût spatial : $\Theta(W_{\max})$.

Problème du sac à dos

Étant donnés n objets de valeurs $c_1, \ldots, c_n$ et de poids $w_1, \ldots, w_n$, comment remplir un sac maximisant la valeur $\sum_{i \in I} c_i$ en respectant la contrainte $\sum_{i \in I} w_i \leqslant W_{\max}$?

On note $f(n, W_{\max})$ la valeur maximale qu'il est possible d'atteindre avec n objets :

$$f(n, W_{\max}) = \begin{cases} \max\Big(c_n + f(n-1, W_{\max} - w_n), f(n-1, W_{\max})\Big) & \text{si } w_n \leqslant W_{\max} \\ f(n-1, W_{\max}) & \text{sinon} \end{cases}$$

Avec la liste des objets à emporter :

```
let sacados c w wmax =
  let n = vect_length c in
  let f = make_vect (wmax+1) (0, []) in
  for k = 0 to n-1 do
    for x = wmax downto 0 do
      if w.(k) <= x && c.(k)+fst f.(x-w.(k)) > fst f.(x)
        then f.(x) <- c.(k)+fst f.(x-w.(k)), k::snd f.(x-w.(k))
    done
  done ;
  f.(wmax) ;;
```

Distance d'édition

Elle est égale au nombre minimal de caractères qu'il faut supprimer, insérer ou remplacer pour passer d'une chaîne de caractères à une autre.

Exemple : la distance de **`polynomial`** à **`polygonal`** est égale à 3 :

- suppression de la lettre **`'i'`** : **`polynomial`** → **`polynomal`** ;
- remplacement du **`'n'`** par un **`'g'`** : **`polynomal`** → **`polygomal`** ;
- remplacement du **`'m'`** par un **`'n'`** : **`polygomal`** → **`polygonal`** ;

Distance d'édition

Elle est égale au nombre minimal de caractères qu'il faut supprimer, insérer ou remplacer pour passer d'une chaîne de caractères à une autre.

On note $\mathrm{d}(i,j)$ la distance d'édition entre $a_1 a_2 \cdots a_i$ et $b_1 b_2 \cdots b_j$. Alors :

- si a_i a été supprimé, $\mathrm{d}(i,j) = \mathrm{d}(i-1,j) + 1$;
- si b_j a été inséré, $\mathrm{d}(i,j) = \mathrm{d}(i,j-1) + 1$;
- si a_i a été remplacé par b_j, $\mathrm{d}(i,j) = \mathrm{d}(i-1,j-1) + 1$;
- si $a_i = b_j$, $\mathrm{d}(i,j) = \mathrm{d}(i-1,j-1)$.

On en déduit :

$$\mathrm{d}(i,j) = \begin{cases} \min\Big(\mathrm{d}(i-1,j), \mathrm{d}(i,j-1), \mathrm{d}(1-1,j-1)\Big) + 1 & \text{si } a_i \neq b_j \\ \min\Big(\mathrm{d}(i-1,j) + 1, \mathrm{d}(i,j-1) + 1, \mathrm{d}(i-1,j-1)\Big) & \text{si } a_i = b_j \end{cases}$$

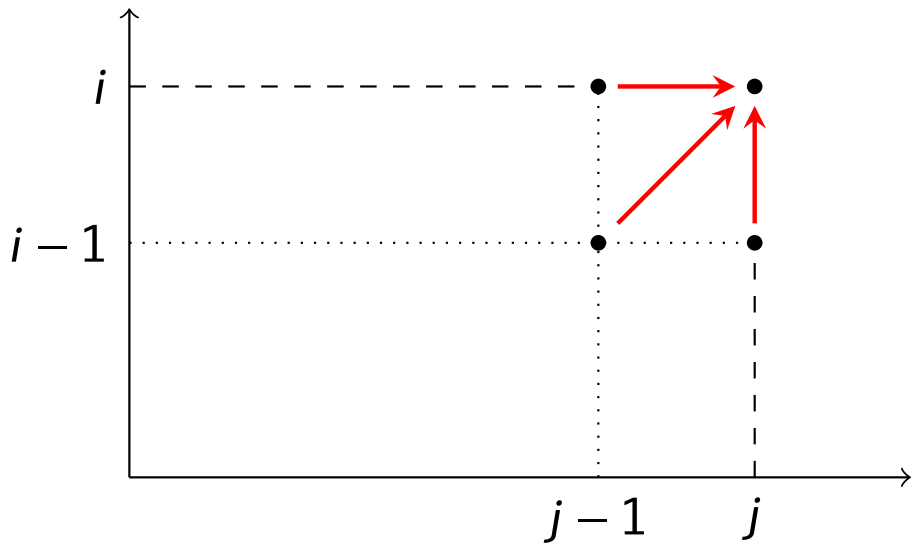
avec $\mathrm{d}(i,0) = i$ et $\mathrm{d}(0,j) = j$.

Distance d'édition

Elle est égale au nombre minimal de caractères qu'il faut supprimer, insérer ou remplacer pour passer d'une chaîne de caractères à une autre.

$$\mathrm{d}(i,j) = \begin{cases} \min\Big(\mathrm{d}(i-1,j), \mathrm{d}(i,j-1), \mathrm{d}(1-1,j-1)\Big) + 1 & \text{si } a_i \neq b_j \\ \min\Big(\mathrm{d}(i-1,j) + 1, \mathrm{d}(i,j-1) + 1, \mathrm{d}(i-1,j-1)\Big) & \text{si } a_i = b_j \end{cases}$$

Schema de dépendance :



Distance d'édition

Elle est égale au nombre minimal de caractères qu'il faut supprimer, insérer ou remplacer pour passer d'une chaîne de caractères à une autre.

```
let dist a b =
  let m = string_length a and n = string_length b in
  let d = make_matrix (m + 1) (n + 1) 0 in
  for i = 1 to m do
    d.(i).(0) <- i
  done ;
  for j = 1 to n do
    d.(0).(j) <- j
  done ;
  for i = 1 to m do
    for j = 1 to n do
      d.(i).(j) <- min d.(i).(j-1) d.(i-1).(j) ;
      if a.[i-1] = b.[j-1]
        then d.(i).(j) <- min d.(i).(j) d.(i-1).(j-1) + 1
        else d.(i).(j) <- min (d.(i).(j) + 1) d.(i-1).(j-1)
    done
  done ;
  d.(m).(n) ;;
```

Coût temporel et spatial en $\Theta(mn)$.