Programmation dynamique

Quentin Fortier

July 2, 2022

Sous-problèmes

Pour résoudre un problème, il est courant de le ramener à des sous-problèmes plus simples.

Sous-problèmes

Pour résoudre un problème, il est courant de le ramener à des sous-problèmes plus simples.

Deux grandes méthodes pour le faire :

1. **Diviser pour régner** : résoudre les sous-problèmes (récursivement) puis les combiner pour obtenir une solution du problème initial.

Sous-problèmes

Pour résoudre un problème, il est courant de le ramener à des sous-problèmes plus simples.

Deux grandes méthodes pour le faire :

1. **Diviser pour régner** : résoudre les sous-problèmes (récursivement) puis les combiner pour obtenir une solution du problème initial.
2. **Programmation dynamique / mémoïsation** : similaire, mais en conservant en mémoire tous les sous-problèmes pour éviter de les calculer plusieurs fois.

Sous-problèmes

Exemple pour le calcul des termes de la suite de Fibonacci :

$$F_0 = 0$$

$$F_1 = 1$$

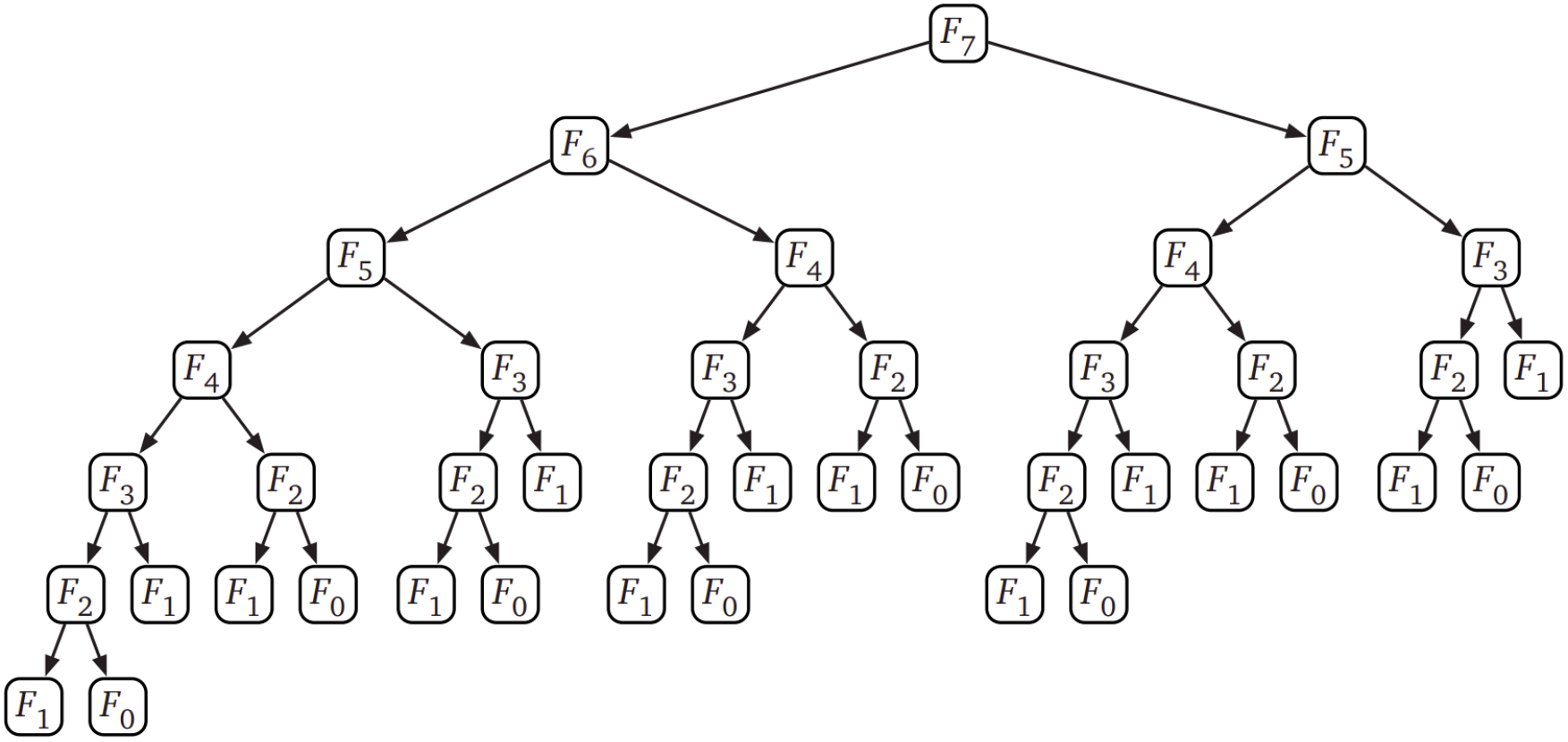
$$F_n = F_{n-1} + F_{n-2}$$

```
int fibo(int n) {
    if (n <= 1)
        return 1;
    return fibo(n - 1) + fibo(n - 2);
}
```

Complexité : exponentielle

Sous-problèmes

Problème : le même sous-problème est résolu plusieurs fois, ce qui est inutile et inefficace.



Sous-problèmes

Idée : stocker les valeurs des sous-problèmes pour éviter de les calculer plusieurs fois.

```
int fibo(int n) {
    int* F = malloc(n*sizeof(int));
    F[0] = 0;
    F[1] = 1;
    for(int i=2; i < n; i++)
        F[i] = F[i - 1] + F[i - 2];
    return F[n - 1];
}
```

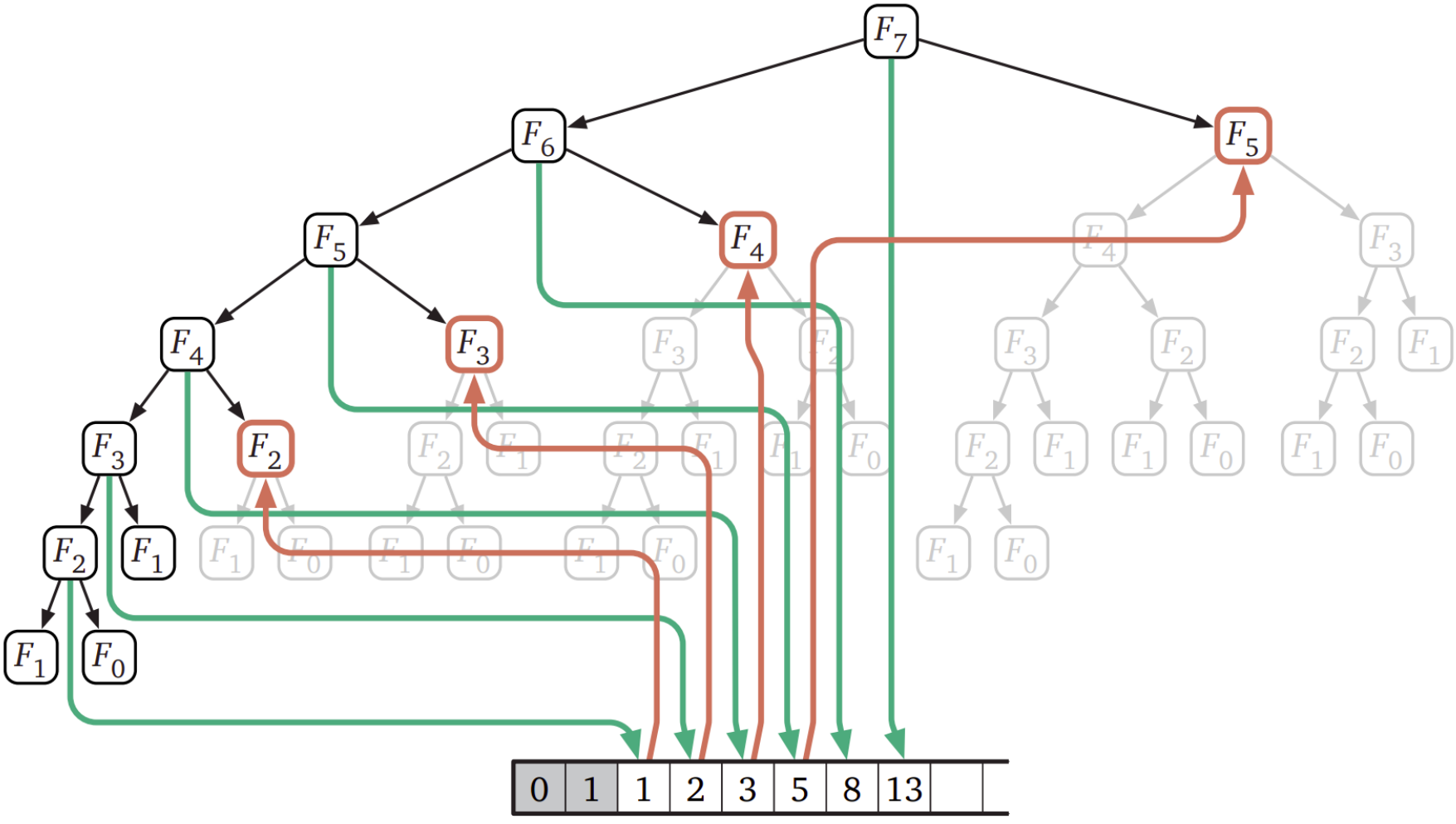
Sous-problèmes

Idée : stocker les valeurs des sous-problèmes pour éviter de les calculer plusieurs fois.

```
int fibo(int n) {
    int* F = malloc(n*sizeof(int));
    F[0] = 0;
    F[1] = 1;
    for(int i=2; i < n; i++)
        F[i] = F[i - 1] + F[i - 2];
    return F[n - 1];
}
```

Complexité : linéaire en `n`

Sous-problèmes



Sous-problèmes

Dans le cas de la suite de Fibonacci, on peut stocker seulement les 2 derniers termes :

```
int fibo(int n) {
    int f0 = 0, f1 = 1;
    for(int i=1; i < n; i++) {
        int tmp = f1;
        f1 = f0 + f1;
        f0 = tmp;
    }
    return f1;
}
```

Programmation dynamique

Pour résoudre un problème de programmation dynamique :

1. Chercher une équation de récurrence. Souvent, cela demande d'introduire un paramètre.

Programmation dynamique

Pour résoudre un problème de programmation dynamique :

1. Chercher une équation de récurrence. Souvent, cela demande d'introduire un paramètre.
2. Stocker en mémoire les résultats des sous-problèmes pour éviter de les calculer plusieurs fois.

Programmation dynamique

Nous allons voir plusieurs applications de la programmation dynamique :

- Sac à dos
- Trouver une sous-suite croissante maximale dans un tableau
- (Plus tard) Plus courts chemins dans un graphe pondéré : Bellman-Ford et Floyd-Warshall

Programmation dynamique : Sac à dos

Problème (sac à dos)

Entrée : un sac à dos de capacité c, des objets $o_1, ..., o_n$ de poids $w_1, ..., w_n$ et valeurs v_1, ..., v_n.
Sortie : la valeur maximum que l'on peut mettre dans le sac.

Programmation dynamique : Sac à dos

Problème (sac à dos)

Entrée : un sac à dos de capacité c, des objets $o_1, ..., o_n$ de poids $w_1, ..., w_n$ et valeurs v_1, ..., v_n.
Sortie : la valeur maximum que l'on peut mettre dans le sac.

Soit $dp[i][j]$ la valeur maximum que l'on peut mettre dans un sac de capacité i, en ne considérant que les objets o_1, ..., o_j.

Programmation dynamique : Sac à dos

Problème (sac à dos)

Entrée : un sac à dos de capacité c, des objets $o_1, ..., o_n$ de poids $w_1, ..., w_n$ et valeurs v_1, ..., v_n.
Sortie : la valeur maximum que l'on peut mettre dans le sac.

Soit $dp[i][j]$ la valeur maximum que l'on peut mettre dans un sac de capacité i, en ne considérant que les objets o_1, ..., o_j.

$$dp[i][0] = 0$$

$$dp[i][j] = \max(\underbrace{dp[i][j-1]}_{\text{sans prendre } o_j}, \underbrace{dp[i-w_j][j-1] + v_j}_{\text{en prenant } o_j, \text{si } i-w_j \geq 0})$$

Programmation dynamique : Sac à dos

Résolution du sac à dos par programmation dynamique

Pour $i = 0$ à c:
    $dp[i][0] \leftarrow 0$

Pour $i = 0$ à c:
    Pour $j = 0$ à n:
        Si $i - w_j \geq 0$:
            $dp[i][j] \leftarrow \max(dp[i][j-1], dp[i-w_j][i-1] + v_j)$
        Sinon:
            $dp[i][j] \leftarrow dp[i][j-1]$

Complexité :

Programmation dynamique : Sac à dos

Résolution du sac à dos par programmation dynamique

Pour $i = 0$ à c:
    $dp[i][0] \leftarrow 0$

Pour $i = 0$ à c:
    Pour $j = 0$ à n:
        Si $i - w_j \geq 0$:
            $dp[i][j] \leftarrow \max(dp[i][j-1], dp[i-w_j][i-1] + v_j)$
        Sinon:
            $dp[i][j] \leftarrow dp[i][j-1]$

Complexité : $O(nc)$

Programmation dynamique : Sac à dos

```
int knapsack(int c, int n, int *w, int *v) {
    int **dp = malloc((c + 1) * sizeof(int*));
    for (int i = 0; i < c + 1; i++) {
        dp[i] = malloc((n + 1) * sizeof(int));
        dp[i][0] = 0;
    }

    for (int i = 0; i <= c; i++)
        for (int j = 0; j <= n; j++){
            dp[i][j] = dp[i][j - 1];
            if (w[j] <= i)
                dp[i][j] = max(dp[i][j], v[j] + dp[i - w[j]][j - 1]);
      }
    return dp[c][n];
}
```

Programmation dynamique : Sac à dos

Comme on a juste besoin de stocker `dp[...][k - 1]` pour calculer `dp[...][k]` :

Programmation dynamique : Sac à dos

Comme on a juste besoin de stocker `dp[...][k - 1]` pour calculer `dp[...][k]` :

```
int knapsack(int c, int n, int *w, int *v) {
    int *dp = malloc((c + 1)*sizeof(int));
    for (int i = 0; i <= c; i++)
        dp[i] = 0;

    for (int j = 0; j <= n; j++) {
        int *dp_ = malloc((c + 1)*sizeof(int));
        memcpy(dp_, dp, (c + 1)*sizeof(int));
        for (int i = 0; i <= c; i++)
            if (w[j] <= i)
                dp[i] = max(dp[i], v[j] + dp_[i - w[j]]);
    }
    return dp[c];
}
```

Mémoïsation

La programmation dynamique est une stratégie **bottom-up** : on résout les problèmes du plus petit au plus grand. On utilise des boucles for.

Mémoïsation

La programmation dynamique est une stratégie **bottom-up** : on résout les problèmes du plus petit au plus grand. On utilise des boucles for.

La **mémoïsation** est similaire mais avec une stratégie **top-down** : on part du problème initial pour le décomposer. On utilise des appels récursifs.

Mémoïsation

La programmation dynamique est une stratégie **bottom-up** : on résout les problèmes du plus petit au plus grand. On utilise des boucles for.

La **mémoïsation** est similaire mais avec une stratégie **top-down** : on part du problème initial pour le décomposer. On utilise des appels récursifs.

Pour éviter de résoudre plusieurs fois le même problème (comme pour Fibonacci), on mémorise (dans un tableau ou un dictionnaire) les arguments pour lesquelles la fonction récursive a déjà été calculée.

Mémoïsation : Fibonacci

```
let fibo n =
  let module M = Map.Make(Int) in
  let rec aux i d =
    if i <= 2 then 1, d
    else match M.find_opt i d with
      | Some v -> v, d
      | None ->
          let v1, d1 = aux (i - 1) d in
          let v2, d2 = aux (i - 2) d1 in
          v1 + v2, M.add i (v1 + v2) d2
  in fst (aux n M.empty)
```

`Map.Make` est un foncteur qui prend en argument un module (ici `Int`, contenant des fonctions sur les entiers) et qui renvoie un module qui permet d'utiliser un dictionnaire implémenté par ABR équilibré.

Mémoïsation : Sac à dos

```
module T = struct
    type t = int*int
    let compare p q = fst p - fst q
end
module M = Map.Make(T)

let knapsack c v w =
    let rec aux i j m = match M.find_opt (i, j) m with
        | Some r -> r, m
        | None ->
            if j = 0 then 0, m
            else let r1, m1 = aux i (j - 1) m in
            if w.(j) > i then r1, m1
            else
                let r2, m2 = aux (i - w.(j)) (j - 1) m1 in
                let r2 = r2 + v.(j) in
                max r1 r2, M.add (i, j) r2 m2
    in fst (aux c (Array.length v - 1) M.empty);;
```

Mémoïsation automatique

Il est possible de « mémoïser » automatiquement une fonction.

- En Python 3.10 :

```python
@cache
def f(n):
    if n <= 1:
        return n
    return f(n-1) + f(n-2)
```

- En OCaml

Sous-suite croissante

Soit T un tableau.

Définition

Une **sous-suite croissante** de T correspond à des éléments $T[i_1] \leq T[i_2] \leq ... \leq T[i_k]$ avec $i_1 \leq i_2 \leq ... \leq i_k$.

Sous-suite croissante

Soit T un tableau.

Définition

Une **sous-suite croissante** de T correspond à des éléments $T[i_1] \leq T[i_2] \leq ... \leq T[i_k]$ avec $i_1 \leq i_2 \leq ... \leq i_k$.

Problème

Trouver la longueur maximum d'une sous-suite croissante de T.

Sous-suite croissante

Soit T un tableau.

Définition

Une **sous-suite croissante** de T correspond à des éléments $T[i_1] \leq T[i_2] \leq ... \leq T[i_k]$ avec $i_1 \leq i_2 \leq ... \leq i_k$.

Problème

Trouver la longueur maximum d'une sous-suite croissante de T.

Exemple :

$$T = [8, 1, 3, 7, 5, 6, 4]$$

Sous-suite croissante

Soit T un tableau.

Définition

Une **sous-suite croissante** de T correspond à des éléments $T[i_1] \leq T[i_2] \leq ... \leq T[i_k]$ avec $i_1 \leq i_2 \leq ... \leq i_k$.

Problème

Trouver la longueur maximum d'une sous-suite croissante de T.

Exemple :

$$T = [8, 1, 3, 7, 5, 6, 4]$$

Longueur maximum : 4.

Sous-suite croissante

Soit T un tableau.
Soit $L[k]$ la longueur d'une plus longue sous-suite croissante (**LIS** en anglais, pour Longest Increasing Subsequence) terminant en $T[k]$ (c'est à dire de la forme $T[i_1] \leq T[i_2] \leq ... \leq T[i_p] = T[k]$).

Sous-suite croissante

Soit T un tableau.
Soit $L[k]$ la longueur d'une plus longue sous-suite croissante (**LIS** en anglais, pour Longest Increasing Subsequence) terminant en $T[k]$ (c'est à dire de la forme $T[i_1] \leq T[i_2] \leq ... \leq T[i_p] = T[k]$).

Exemple :

$$T = [8, 1, 3, 7, 5, 6, 4]$$

LIS terminant en $T[6]$ $(= 4)$:

Sous-suite croissante

Soit T un tableau.
Soit $L[k]$ la longueur d'une plus longue sous-suite croissante (**LIS** en anglais, pour Longest Increasing Subsequence) terminant en $T[k]$ (c'est à dire de la forme $T[i_1] \leq T[i_2] \leq ... \leq T[i_p] = T[k]$).

Exemple :

$$T = [8, 1, 3, 7, 5, 6, 4]$$

LIS terminant en $T[6]$ $(= 4)$:

$$T = [8, 1, 3, 7, 5, 6, 4]$$

$L[6] = 3$

Sous-suite croissante

Soit $T[i_1] \leq ... \leq T[i_{p-1}] \leq T[i_p] = T[k]$ une LIS terminant en $T[k]$.

Sous-suite croissante

Soit $T[i_1] \leq ... \leq T[i_{p-1}] \leq T[i_p] = T[k]$ une LIS terminant en $T[k]$.

Alors $T[i_1] \leq ... \leq T[i_{p-1}]$ est une LIS terminant en $T[i_{p-1}]$

Sous-suite croissante

Soit $T[i_1] \leq ... \leq T[i_{p-1}] \leq T[i_p] = T[k]$ une LIS terminant en $T[k]$.

Alors $T[i_1] \leq ... \leq T[i_{p-1}]$ est une LIS terminant en $T[i_{p-1}]$ (s'il y avait une LIS plus grande on pourrait l'utiliser dans la LIS initiale pour contredire sa maximalité).

Sous-suite croissante

Soit $T[i_1] \leq ... \leq T[i_{p-1}] \leq T[i_p] = T[k]$ une LIS terminant en $T[k]$.

Alors $T[i_1] \leq ... \leq T[i_{p-1}]$ est une LIS terminant en $T[i_{p-1}]$ (s'il y avait une LIS plus grande on pourrait l'utiliser dans la LIS initiale pour contredire sa maximalité).
Donc :

$$L[k] = 1 + L[i_{p-1}]$$

Sous-suite croissante

Soit $T[i_1] \leq ... \leq T[i_{p-1}] \leq T[i_p] = T[k]$ une LIS terminant en $T[k]$.

Alors $T[i_1] \leq ... \leq T[i_{p-1}]$ est une LIS terminant en $T[i_{p-1}]$ (s'il y avait une LIS plus grande on pourrait l'utiliser dans la LIS initiale pour contredire sa maximalité).
Donc :

$$L[k] = 1 + L[i_{p-1}]$$

Comme on ne connaît pas i_{p-1}, on peut essayer toutes les possibilités et conserver le maximum :

$$\boxed{L[k] = 1 + \max_{\substack{i \leq k \\ T[i] \leq T[k]}} L[i]}$$

Sous-suite croissante

Soit $T[i_1] \leq ... \leq T[i_{p-1}] \leq T[i_p] = T[k]$ une LIS terminant en $T[k]$.

Alors $T[i_1] \leq ... \leq T[i_{p-1}]$ est une LIS terminant en $T[i_{p-1}]$ (s'il y avait une LIS plus grande on pourrait l'utiliser dans la LIS initiale pour contredire sa maximalité).
Donc :

$$L[k] = 1 + L[i_{p-1}]$$

Comme on ne connaît pas i_{p-1}, on peut essayer toutes les possibilités et conserver le maximum :

$$\boxed{L[k] = 1 + \max_{\substack{i \leq k \\ T[i] \leq T[k]}} L[i]}$$

Exercice : le programmer.

Annexe : Théorème d'Erdős-Szekeres

Lemme

Supposons que $L[k]$ contienne p fois la même valeur. Montrer que T possède une sous-suite décroissante de longueur p.

Annexe : Théorème d'Erdős-Szekeres

Lemme

Supposons que $L[k]$ contienne p fois la même valeur. Montrer que T possède une sous-suite décroissante de longueur p.

Théorème d'Erdős-Szekeres

Si n est la taille de T, montrer que T contient soit une sous-suite croissante de longueur $\lfloor\sqrt{n}\rfloor$, soit une sous-suite décroissante de longueur $\lfloor\sqrt{n}\rfloor$