Algorithmique des graphes

10 — Programmation dynamique

Anthony Labarre

14 avril 2021

Résumé des épisodes précédents

- On a mentionné trois catégories de techniques algorithmiques et couvert les deux premières :

Résumé des épisodes précédents

- On a mentionné trois catégories de techniques algorithmiques et couvert les deux premières :

  1. l'approche gloutonne ;

Résumé des épisodes précédents

- On a mentionné trois catégories de techniques algorithmiques et couvert les deux premières :
  1. l'approche gloutonne ;
  2. l'approche diviser pour régner ;

Résumé des épisodes précédents

- On a mentionné trois catégories de techniques algorithmiques et couvert les deux premières :

  1. l'approche gloutonne ;
  2. l'approche diviser pour régner ;
  3. la programmation dynamique ;

Résumé des épisodes précédents

- On a mentionné trois catégories de techniques algorithmiques et couvert les deux premières :
  1. l'approche gloutonne ;
  2. l'approche diviser pour régner ;
  3. la programmation dynamique ;
- On a défini la notion de sous-problème ;

Résumé des épisodes précédents

- On a mentionné trois catégories de techniques algorithmiques et couvert les deux premières :
  1. l'approche gloutonne ;
  2. l'approche diviser pour régner ;
  3. la programmation dynamique ;
- On a défini la notion de sous-problème ;
- On va maintenant examiner comment la programmation dynamique en tire parti ;

Programmation dynamique

- Le terme de "programmation dynamique" n'a rien à voir avec l'implémentation (C, C++, ...) ;

Programmation dynamique

- Le terme de “programmation dynamique” n'a rien à voir avec l'implémentation (C, C++, . . .) ;
- Le mot “programmation” désigne ici l'utilisation d'un tableau pour stocker des valeurs (cf. “programmation linéaire”) ;

Programmation dynamique

- Le terme de "programmation dynamique" n'a rien à voir avec l'implémentation (C, C++, ...) ;
- Le mot "programmation" désigne ici l'utilisation d'un tableau pour stocker des valeurs (cf. "programmation linéaire") ;
- Le mot "dynamique" signifie comme on s'y attend que ce tableau et son utilisation vont changer au cours de l'exécution de l'algorithme ;

Principe de la programmation dynamique

- La programmation dynamique est peut-être plus facile à comprendre en prenant comme point de départ les algorithmes récursifs ;

Principe de la programmation dynamique

- La programmation dynamique est peut-être plus facile à comprendre en prenant comme point de départ les algorithmes récursifs ;
- L'idée consiste à accélérer ces algorithmes en stockant les résultats intermédiaires plutôt que d'effectuer des appels récursifs ;

Principe de la programmation dynamique

- La programmation dynamique est peut-être plus facile à comprendre en prenant comme point de départ les algorithmes récursifs ;
- L'idée consiste à accélérer ces algorithmes en stockant les résultats intermédiaires plutôt que d'effectuer des appels récursifs ;
- Si possible, d'autres optimisations suivront ;

Principe de la programmation dynamique

- La programmation dynamique est peut-être plus facile à comprendre en prenant comme point de départ les algorithmes récursifs ;
- L'idée consiste à accélérer ces algorithmes en stockant les résultats intermédiaires plutôt que d'effectuer des appels récursifs ;
- Si possible, d'autres optimisations suivront ;
- Cette approche sera d'autant plus efficace que la solution récursive "naïve" réutilise des résultats déjà connus ;

Exemple : nombres de Fibonacci

Pour rappel, le $n^{\text{ème}}$ nombre de Fibonacci F_n est défini $\forall\ n \in \mathbb{N}$ par :

$$F_n = \begin{cases} n & \text{si } n \leq 1, \\ F_{n-1} + F_{n-2} & \text{sinon.} \end{cases}$$

Exemple : nombres de Fibonacci

Pour rappel, le $n^{\text{ème}}$ nombre de Fibonacci F_n est défini $\forall\ n \in \mathbb{N}$ par :

$$F_n = \begin{cases} n & \text{si } n \leq 1, \\ F_{n-1} + F_{n-2} & \text{sinon.} \end{cases}$$

Version triviale $\textsc{Fibo}(n)$

Entrées : un naturel n.
Sortie : le n-ème nombre de Fibonacci.

si $n \leq 1$ **alors renvoyer** n;
renvoyer $\textsc{Fibo}(n-1) + \textsc{Fibo}(n-2)$;

Exemple : nombres de Fibonacci

Pour rappel, le $n^{\text{ème}}$ nombre de Fibonacci F_n est défini $\forall\, n \in \mathbb{N}$ par :

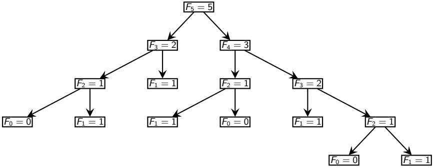
$$F_n = \begin{cases} n & \text{si } n \leq 1, \\ F_{n-1} + F_{n-2} & \text{sinon.} \end{cases}$$

Version triviale $\textsc{Fibo}(n)$

Entrées : un naturel n.
Sortie : le n-ème nombre de Fibonacci.

si $n \leq 1$ **alors renvoyer** n;
renvoyer $\textsc{Fibo}(n-1) + \textsc{Fibo}(n-2)$;

Arbre d'appels pour $\textsc{Fibo}(5)$



Exemple : nombres de Fibonacci

Pour rappel, le $n^{\text{ème}}$ nombre de Fibonacci F_n est défini $\forall\ n \in \mathbb{N}$ par :

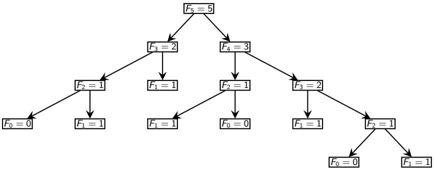
$$F_n = \begin{cases} n & \text{si } n \leq 1, \\ F_{n-1} + F_{n-2} & \text{sinon.} \end{cases}$$

Version triviale $\textsc{Fibo}(n)$

Entrées : un naturel n.
Sortie : le n-ème nombre de Fibonacci.

si $n \leq 1$ **alors renvoyer** n;
renvoyer $\textsc{Fibo}(n-1) + \textsc{Fibo}(n-2)$;

Arbre d'appels pour $\textsc{Fibo}(5)$



Quelle est la complexité de $\textsc{Fibo}(n)$?

Exemple : nombres de Fibonacci

Pour rappel, le $n^{\text{ème}}$ nombre de Fibonacci F_n est défini $\forall\ n \in \mathbb{N}$ par :

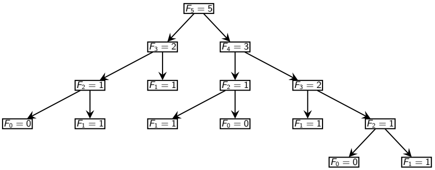
$$F_n = \begin{cases} n & \text{si } n \leq 1, \\ F_{n-1} + F_{n-2} & \text{sinon.} \end{cases}$$

Version triviale $\textsc{Fibo}(n)$

Entrées : un naturel n.
Sortie : le n-ème nombre de Fibonacci.

si $n \leq 1$ **alors renvoyer** n;
renvoyer $\textsc{Fibo}(n-1) + \textsc{Fibo}(n-2)$;

Arbre d'appels pour $\textsc{Fibo}(5)$



Quelle est la complexité de $\textsc{Fibo}(n)$?

- opérations en $O(1) \Rightarrow$ seul le nombre d'appels récursifs compte ;

Exemple : nombres de Fibonacci

Pour rappel, le $n^{\text{ème}}$ nombre de Fibonacci F_n est défini $\forall\ n \in \mathbb{N}$ par :

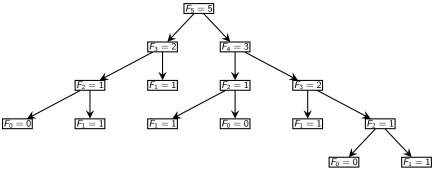
$$F_n = \begin{cases} n & \text{si } n \leq 1, \\ F_{n-1} + F_{n-2} & \text{sinon.} \end{cases}$$

Version triviale $\textsc{Fibo}(n)$

Entrées : un naturel n.
Sortie : le n-ème nombre de Fibonacci.

si $n \leq 1$ **alors renvoyer** n;
renvoyer $\textsc{Fibo}(n-1) + \textsc{Fibo}(n-2)$;

Arbre d'appels pour $\textsc{Fibo}(5)$



Quelle est la complexité de $\textsc{Fibo}(n)$?

- opérations en $O(1) \Rightarrow$ seul le nombre d'appels récursifs compte ;
- $\textsc{Fibo}(n)$ s'appelle deux fois, dont une avec $n-1$;

Exemple : nombres de Fibonacci

Pour rappel, le $n^{\text{ème}}$ nombre de Fibonacci F_n est défini $\forall\, n \in \mathbb{N}$ par :

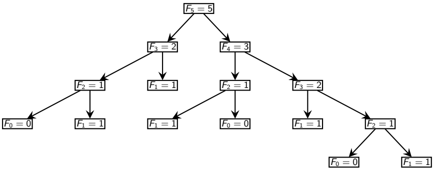
$$F_n = \begin{cases} n & \text{si } n \leq 1, \\ F_{n-1} + F_{n-2} & \text{sinon.} \end{cases}$$

Version triviale $\textsc{Fibo}(n)$

Entrées : un naturel n.
Sortie : le n-ème nombre de Fibonacci.

si $n \leq 1$ **alors renvoyer** n;
renvoyer $\textsc{Fibo}(n-1) + \textsc{Fibo}(n-2)$;

Arbre d'appels pour $\textsc{Fibo}(5)$



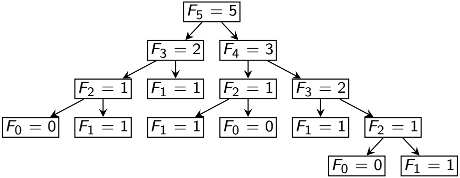
Quelle est la complexité de $\textsc{Fibo}(n)$?

- opérations en $O(1) \Rightarrow$ seul le nombre d'appels récursifs compte ;
- $\textsc{Fibo}(n)$ s'appelle deux fois, dont une avec $n-1$;
- on a donc du $O(2^n)$.

Simplification de l'arbre d'appels

Ces appels récursifs effectuent énormément de travail inutile :

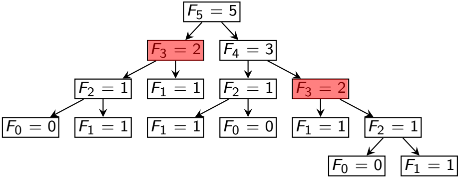
Arbre d'appels



Simplification de l'arbre d'appels

Ces appels récursifs effectuent énormément de travail inutile :

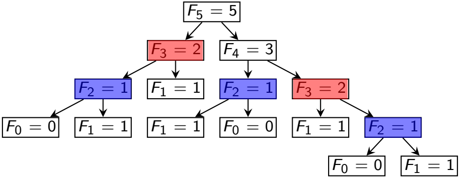
Arbre d'appels



Simplification de l'arbre d'appels

Ces appels récursifs effectuent énormément de travail inutile :

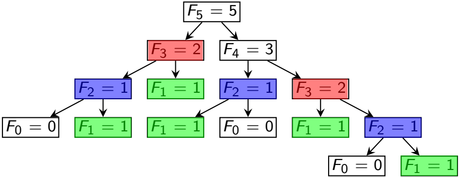
Arbre d'appels



Simplification de l'arbre d'appels

Ces appels récursifs effectuent énormément de travail inutile :

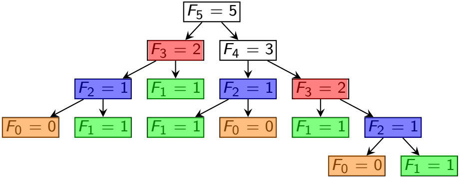
Arbre d'appels



Simplification de l'arbre d'appels

Ces appels récursifs effectuent énormément de travail inutile :

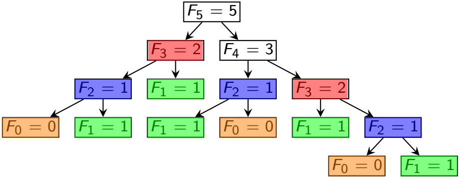
Arbre d'appels



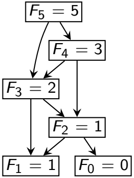
Simplification de l'arbre d'appels

Ces appels récursifs effectuent énormément de travail inutile :

Arbre d'appels



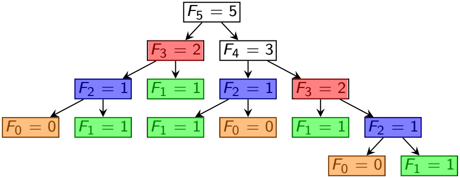
DAG d'appels simplifié



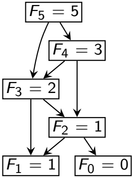
Simplification de l'arbre d'appels

Ces appels récursifs effectuent énormément de travail inutile :

Arbre d'appels



DAG d'appels simplifié



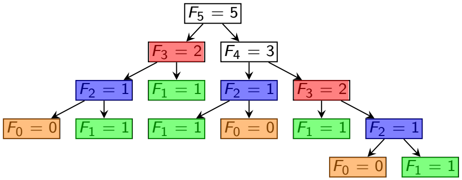
Pour obtenir un algorithme correspondant à la version de droite, on doit stocker les résultats intermédiaires dans un tableau pour pouvoir les exploiter :

i :	0	1	2	3	4	5
F_i :	0	1	-1	-1	-1	-1

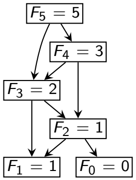
Simplification de l'arbre d'appels

Ces appels récursifs effectuent énormément de travail inutile :

Arbre d'appels



DAG d'appels simplifié



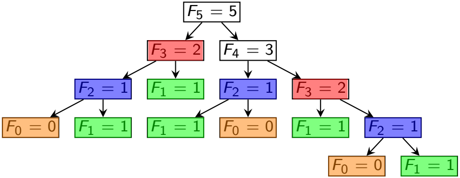
Pour obtenir un algorithme correspondant à la version de droite, on doit stocker les résultats intermédiaires dans un tableau pour pouvoir les exploiter :

$$\begin{array}{rcccccc} i: & 0 & 1 & 2 & 3 & 4 & 5 \\ F_i: & 0 & 1 & -1 & -1 & -1 & -1 \end{array}$$

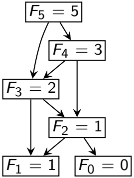
Simplification de l'arbre d'appels

Ces appels récursifs effectuent énormément de travail inutile :

Arbre d'appels



DAG d'appels simplifié



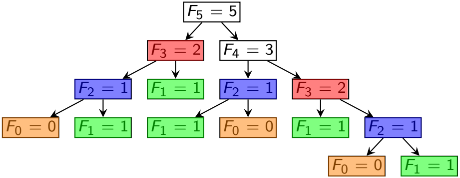
Pour obtenir un algorithme correspondant à la version de droite, on doit stocker les résultats intermédiaires dans un tableau pour pouvoir les exploiter :

i :	0	1	2	3	4	5
F_i :	0	1	-1	-1	-1	-1

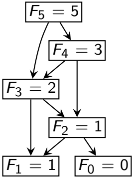
Simplification de l'arbre d'appels

Ces appels récursifs effectuent énormément de travail inutile :

Arbre d'appels



DAG d'appels simplifié



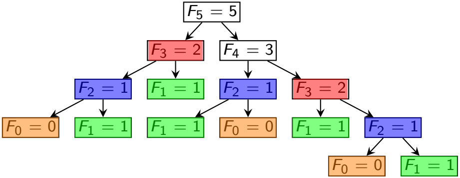
Pour obtenir un algorithme correspondant à la version de droite, on doit stocker les résultats intermédiaires dans un tableau pour pouvoir les exploiter :

$$\begin{array}{rcccccc} i: & 0 & 1 & 2 & 3 & 4 & 5 \\ F_i: & 0 & 1 & -1 & -1 & -1 & -1 \end{array}$$

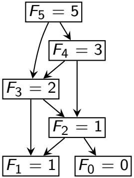
Simplification de l'arbre d'appels

Ces appels récursifs effectuent énormément de travail inutile :

Arbre d'appels



DAG d'appels simplifié



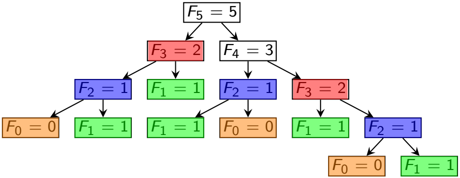
Pour obtenir un algorithme correspondant à la version de droite, on doit stocker les résultats intermédiaires dans un tableau pour pouvoir les exploiter :

$$\begin{array}{rcccccc} i: & 0 & 1 & 2 & 3 & 4 & 5 \\ F_i: & 0 & 1 & -1 & -1 & -1 & -1 \end{array}$$

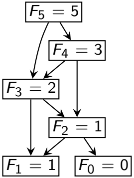
Simplification de l'arbre d'appels

Ces appels récursifs effectuent énormément de travail inutile :

Arbre d'appels



DAG d'appels simplifié



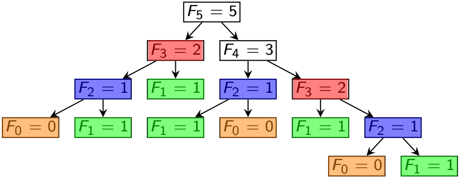
Pour obtenir un algorithme correspondant à la version de droite, on doit stocker les résultats intermédiaires dans un tableau pour pouvoir les exploiter :

i :	0	1	2	3	4	5
F_i :	0	1	1	-1	-1	-1

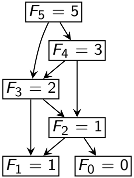
Simplification de l'arbre d'appels

Ces appels récursifs effectuent énormément de travail inutile :

Arbre d'appels



DAG d'appels simplifié



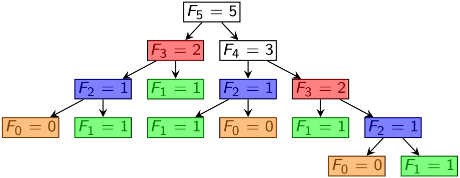
Pour obtenir un algorithme correspondant à la version de droite, on doit stocker les résultats intermédiaires dans un tableau pour pouvoir les exploiter :

i :	0	1	2	3	4	5
F_i :	0	1	1	2	-1	-1

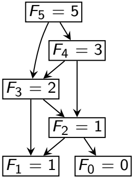
Simplification de l'arbre d'appels

Ces appels récursifs effectuent énormément de travail inutile :

Arbre d'appels



DAG d'appels simplifié



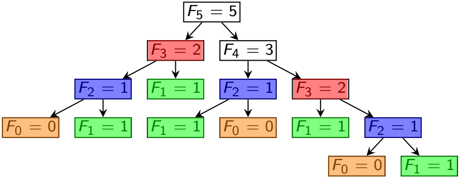
Pour obtenir un algorithme correspondant à la version de droite, on doit stocker les résultats intermédiaires dans un tableau pour pouvoir les exploiter :

$$\begin{array}{rcccccc} i: & 0 & 1 & 2 & 3 & 4 & 5 \\ F_i: & 0 & 1 & 1 & 2 & 3 & -1 \end{array}$$

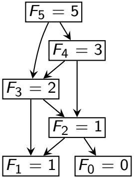
Simplification de l'arbre d'appels

Ces appels récursifs effectuent énormément de travail inutile :

Arbre d'appels



DAG d'appels simplifié



Pour obtenir un algorithme correspondant à la version de droite, on doit stocker les résultats intermédiaires dans un tableau pour pouvoir les exploiter :

i :	0	1	2	3	4	5
F_i :	0	1	1	2	3	5

Exemple : nombres de Fibonacci

On obtient donc la version suivante, qui enregistre les valeurs au fur et à mesure des appels et renvoie celles qui sont connues au lieu de faire des appels récursifs :

Algorithme 1 : Fibonacci récursif avec stockage

Algorithme auxiliaire $\text{FIBOMIEUX}(n)$
  valeurs $\leftarrow$ tableau$(n+1, -1)$; valeurs[0] $\leftarrow 0$; valeurs[1] $\leftarrow 1$;
  renvoyer $\text{FIBOREC}(n)$
Algorithme principal $\text{FIBOREC}(n)$
  si *valeurs[n]* $= -1$ **alors**
    valeurs$[n] \leftarrow \text{FIBOREC}(n-1) + \text{FIBOREC}(n-2)$;
  renvoyer valeurs$[n]$

Exemple : nombres de Fibonacci

On obtient donc la version suivante, qui enregistre les valeurs au fur et à mesure des appels et renvoie celles qui sont connues au lieu de faire des appels récursifs :

Algorithme 1 : Fibonacci récursif avec stockage

Algorithme auxiliaire $\text{FIBOMIEUX}(n)$
- valeurs $\leftarrow$ tableau$(n+1, -1)$; valeurs$[0] \leftarrow 0$; valeurs$[1] \leftarrow 1$;
- **renvoyer** $\text{FIBOREC}(n)$

Algorithme principal $\text{FIBOREC}(n)$
- **si** *valeurs[n]* $= -1$ **alors**
  - valeurs$[n] \leftarrow \text{FIBOREC}(n-1) + \text{FIBOREC}(n-2)$;
- **renvoyer** valeurs$[n]$

On peut encore améliorer les choses :

Exemple : nombres de Fibonacci

On obtient donc la version suivante, qui enregistre les valeurs au fur et à mesure des appels et renvoie celles qui sont connues au lieu de faire des appels récursifs :

Algorithme 1 : Fibonacci récursif avec stockage

```
Algorithme auxiliaire FiboMieux(n)
    valeurs ← tableau(n + 1, −1) ; valeurs[0] ← 0 ; valeurs[1] ← 1 ;
    renvoyer FiboRec(n)
Algorithme principal FiboRec(n)
    si valeurs[n] = −1 alors
        valeurs[n] ← FiboRec(n − 1) + FiboRec(n − 2) ;
    renvoyer valeurs[n]
```

On peut encore améliorer les choses :

- ~~récursivité~~ : on se contente de remplir le tableau de gauche à droite ;

Exemple : nombres de Fibonacci

On obtient donc la version suivante, qui enregistre les valeurs au fur et à mesure des appels et renvoie celles qui sont connues au lieu de faire des appels récursifs :

Algorithme 1 : Fibonacci récursif avec stockage

Algorithme auxiliaire $\texttt{FIBOMIEUX}(n)$
    valeurs $\leftarrow$ tableau$(n+1, -1)$; valeurs$[0] \leftarrow 0$; valeurs$[1] \leftarrow 1$;
    renvoyer $\texttt{FIBOREC}(n)$
Algorithme principal $\texttt{FIBOREC}(n)$
    si *valeurs[n]* $= -1$ **alors**
        valeurs$[n] \leftarrow \texttt{FIBOREC}(n-1) + \texttt{FIBOREC}(n-2)$;
    renvoyer valeurs$[n]$

On peut encore améliorer les choses :

- ~~récursivité~~ : on se contente de remplir le tableau de gauche à droite ;
- ~~tableau~~ : seules les deux dernières valeurs nous intéressent à chaque étape ;

Exemple : nombres de Fibonacci

On obtient donc la version suivante, qui enregistre les valeurs au fur et à mesure des appels et renvoie celles qui sont connues au lieu de faire des appels récursifs :

Algorithme 1 : Fibonacci récursif avec stockage

Algorithme auxiliaire $\text{FIBOMIEUX}(n)$
    valeurs $\leftarrow$ tableau$(n+1, -1)$; valeurs[0] $\leftarrow 0$; valeurs[1] $\leftarrow 1$;
    renvoyer $\text{FIBOREC}(n)$
Algorithme principal $\text{FIBOREC}(n)$
    si *valeurs[n]* $= -1$ **alors**
        valeurs[n] $\leftarrow \text{FIBOREC}(n-1) + \text{FIBOREC}(n-2)$;
    renvoyer valeurs[n]

On peut encore améliorer les choses :

- ~~récursivité~~ : on se contente de remplir le tableau de gauche à droite ;
- ~~tableau~~ : seules les deux dernières valeurs nous intéressent à chaque étape ;
- $\Rightarrow$ au final : complexité en $O(n)$, consommation mémoire en $O(1)$;

Programmation dynamique pour les problèmes d'optimisation

- La programmation dynamique procède comme suit :

Programmation dynamique pour les problèmes d'optimisation

- La programmation dynamique procède comme suit :
  1. on découpe le problème de départ en sous-problèmes plus simples ;

Programmation dynamique pour les problèmes d'optimisation

- La programmation dynamique procède comme suit :
  1. on découpe le problème de départ en sous-problèmes plus simples ;
  2. on les résoud de manière optimale ;

Programmation dynamique pour les problèmes d'optimisation

- La programmation dynamique procède comme suit :
  1. on découpe le problème de départ en sous-problèmes plus simples ;
  2. on les résoud de manière optimale ;
  3. on choisit ensuite la solution optimale au problème de départ sur base des solutions des sous-problèmes ;

Programmation dynamique pour les problèmes d'optimisation

- La programmation dynamique procède comme suit :
  1. on découpe le problème de départ en sous-problèmes plus simples ;
  2. on les résoud de manière optimale ;
  3. on choisit ensuite la solution optimale au problème de départ sur base des solutions des sous-problèmes ;
- Comme on l'a vu, il est essentiel de stocker les calculs intermédiaires si l'on veut éviter une complexité prohibitive.

Programmation dynamique $\neq$ algorithmes gloutons

- Les algorithmes gloutons effectuent le meilleur choix local à chaque étape ;

Programmation dynamique $\neq$ algorithmes gloutons

- Les algorithmes gloutons effectuent le meilleur choix local à chaque étape ;
- La programmation dynamique essaie chaque choix possible et résoud de manière optimale le sous-problème résultant ;

Programmation dynamique $\neq$ algorithmes gloutons

- Les algorithmes gloutons effectuent le meilleur choix local à chaque étape ;
- La programmation dynamique essaie chaque choix possible et résoud de manière optimale le sous-problème résultant ;
- On effectuera donc parfois des choix localement sous-optimaux pour obtenir une solution globalement optimale ;

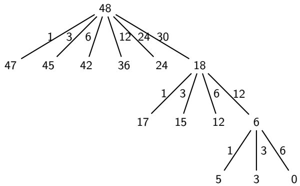
Exemple : approche gloutonne vs. programmation dynamique

Illustrons la différence sur le problème suivant : on veut décomposer un entier M en une somme de valeurs dans un ensemble S en minimisant le nombre de termes. Pour $M = 48$ et $S = \{1, 3, 6, 12, 24, 30, 60, 240\}$:

Exemple : approche gloutonne vs. programmation dynamique

Illustrons la différence sur le problème suivant : on veut décomposer un entier M en une somme de valeurs dans un ensemble S en minimisant le nombre de termes. Pour $M = 48$ et $S = \{1, 3, 6, 12, 24, 30, 60, 240\}$:

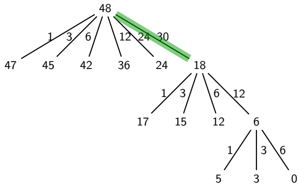
Approche gloutonne



Exemple : approche gloutonne vs. programmation dynamique

Illustrons la différence sur le problème suivant : on veut décomposer un entier M en une somme de valeurs dans un ensemble S en minimisant le nombre de termes. Pour $M = 48$ et $S = \{1, 3, 6, 12, 24, 30, 60, 240\}$:

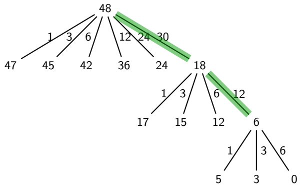
Approche gloutonne



Exemple : approche gloutonne vs. programmation dynamique

Illustrons la différence sur le problème suivant : on veut décomposer un entier M en une somme de valeurs dans un ensemble S en minimisant le nombre de termes. Pour $M = 48$ et $S = \{1, 3, 6, 12, 24, 30, 60, 240\}$:

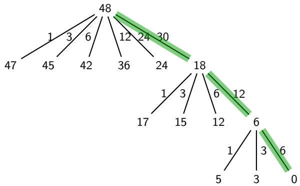
Approche gloutonne



Exemple : approche gloutonne vs. programmation dynamique

Illustrons la différence sur le problème suivant : on veut décomposer un entier M en une somme de valeurs dans un ensemble S en minimisant le nombre de termes. Pour $M = 48$ et $S = \{1, 3, 6, 12, 24, 30, 60, 240\}$:

Approche gloutonne

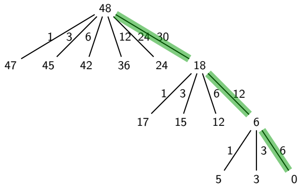


Solution gloutonne de taille 3 :
$48 = 30 + 12 + 6$

Exemple : approche gloutonne vs. programmation dynamique

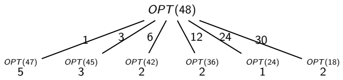
Illustrons la différence sur le problème suivant : on veut décomposer un entier M en une somme de valeurs dans un ensemble S en minimisant le nombre de termes. Pour $M = 48$ et $S = \{1, 3, 6, 12, 24, 30, 60, 240\}$:

Approche gloutonne



Solution gloutonne de taille 3 :
$48 = 30 + 12 + 6$

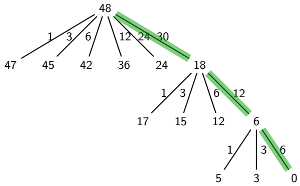
Programmation dynamique



Exemple : approche gloutonne vs. programmation dynamique

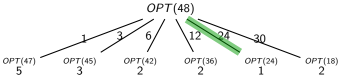
Illustrons la différence sur le problème suivant : on veut décomposer un entier M en une somme de valeurs dans un ensemble S en minimisant le nombre de termes. Pour $M = 48$ et $S = \{1, 3, 6, 12, 24, 30, 60, 240\}$:

Approche gloutonne



Solution gloutonne de taille 3 :
$48 = 30 + 12 + 6$

Programmation dynamique



Solution optimale de taille 2 :
$48 = 24 + 24$

Sous-structure optimale

- La programmation dynamique fonctionne quand le problème possède une **sous-structure optimale** ;

Sous-structure optimale

- La programmation dynamique fonctionne quand le problème possède une **sous-structure optimale** ;
- C'est-à-dire : quand la solution optimale du problème dépend de la solution optimale de certains sous-problèmes ;

Sous-structure optimale

- La programmation dynamique fonctionne quand le problème possède une **sous-structure optimale** ;
- C'est-à-dire : quand la solution optimale du problème dépend de la solution optimale de certains sous-problèmes ;
- Dans le cas de la décomposition en somme de longueur minimale :

$$OPT(n) = \begin{cases} 0 & \text{si } n = 0, \\ 1 + \min_{p \in S : p \leq n} OPT(n-p) & \text{sinon.} \end{cases}$$

Sous-structure optimale

- La programmation dynamique fonctionne quand le problème possède une **sous-structure optimale** ;
- C'est-à-dire : quand la solution optimale du problème dépend de la solution optimale de certains sous-problèmes ;
- Dans le cas de la décomposition en somme de longueur minimale :

$$OPT(n) = \begin{cases} 0 & \text{si } n = 0, \\ 1 + \min_{p \in S : p \leq n} OPT(n-p) & \text{sinon.} \end{cases}$$

- L'équation ci-dessus est qualifiée d'**équation de programmation dynamique** ;

Conception d'algorithmes de programmation dynamique

- Pour concevoir un algorithme de programmation dynamique :

Conception d'algorithmes de programmation dynamique

- Pour concevoir un algorithme de programmation dynamique :
  1. identifier la sous-structure optimale ;

Conception d'algorithmes de programmation dynamique

- Pour concevoir un algorithme de programmation dynamique :
  1. identifier la sous-structure optimale ;
  2. écrire l'équation de programmation dynamique associée ;

Conception d'algorithmes de programmation dynamique

- Pour concevoir un algorithme de programmation dynamique :
  1. identifier la sous-structure optimale ;
  2. écrire l'équation de programmation dynamique associée ;
  3. en déduire un algorithme récursif ;

Conception d'algorithmes de programmation dynamique

- Pour concevoir un algorithme de programmation dynamique :
  1. identifier la sous-structure optimale ;
  2. écrire l'équation de programmation dynamique associée ;
  3. en déduire un algorithme récursif ;
  4. remplacer les appels récursifs par des stockages / consultations de valeurs stockées ;

Conception d'algorithmes de programmation dynamique

- Pour concevoir un algorithme de programmation dynamique :
    1. identifier la sous-structure optimale ;
    2. écrire l'équation de programmation dynamique associée ;
    3. en déduire un algorithme récursif ;
    4. remplacer les appels récursifs par des stockages / consultations de valeurs stockées ;
- Une fois la dernière étape atteinte, on peut parfois encore optimiser l'algorithme résultant ;

Un problème de découpe [2]

DÉCOUPE

Entrées: une longueur de tige n, un tableau de prix p donnant la valeur de chaque morceau de taille 1, 2, ..., n

Objectif: une découpe de la tige en morceaux de longueur $\ell_1, \ell_2, \ldots, \ell_k$ qui maximise $\sum_{i=1}^{k} p[\ell_i]$.

Un problème de découpe [2]

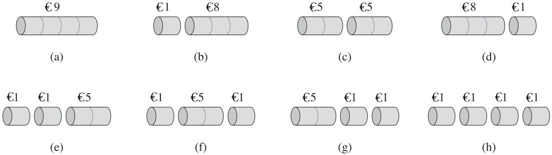
DÉCOUPE

Entrées: une longueur de tige n, un tableau de prix p donnant la valeur de chaque morceau de taille 1, 2, . . . , n

Objectif: une découpe de la tige en morceaux de longueur $\ell_1, \ell_2, \ldots, \ell_k$ qui maximise $\sum_{i=1}^{k} p[\ell_i]$.

Exemple 1

Les 8 manières de découper une tige de longueur 4 en morceaux entiers, avec les prix correspondants [2].



Identification de la sous-structure optimale

- Pour obtenir la structure d'une solution optimale, examinons l'impact du choix d'une découpe ;

Identification de la sous-structure optimale

- Pour obtenir la structure d'une solution optimale, examinons l'impact du choix d'une découpe ;
- Si la tige est de longueur n, on peut la couper en deux morceaux A et B avec :

Identification de la sous-structure optimale

- Pour obtenir la structure d'une solution optimale, examinons l'impact du choix d'une découpe ;
- Si la tige est de longueur n, on peut la couper en deux morceaux A et B avec :
  - longueur$(A) = 1$ et longueur$(B) = n - 1$; ou

Identification de la sous-structure optimale

- Pour obtenir la structure d'une solution optimale, examinons l'impact du choix d'une découpe ;
- Si la tige est de longueur n, on peut la couper en deux morceaux A et B avec :
  - longueur$(A) = 1$ et longueur$(B) = n - 1$; ou
  - longueur$(A) = 2$ et longueur$(B) = n - 2$; ou

Identification de la sous-structure optimale

- Pour obtenir la structure d'une solution optimale, examinons l'impact du choix d'une découpe ;
- Si la tige est de longueur n, on peut la couper en deux morceaux A et B avec :
  - $\text{longueur}(A) = 1$ et $\text{longueur}(B) = n - 1$; ou
  - $\text{longueur}(A) = 2$ et $\text{longueur}(B) = n - 2$; ou

  $\vdots$

Identification de la sous-structure optimale

- Pour obtenir la structure d'une solution optimale, examinons l'impact du choix d'une découpe ;
- Si la tige est de longueur n, on peut la couper en deux morceaux A et B avec :
  - longueur$(A) = 1$ et longueur$(B) = n - 1$; ou
  - longueur$(A) = 2$ et longueur$(B) = n - 2$; ou

    $\vdots$

  - longueur$(A) = n - 1$ et longueur$(B) = 1$; ou

Identification de la sous-structure optimale

- Pour obtenir la structure d'une solution optimale, examinons l'impact du choix d'une découpe ;
- Si la tige est de longueur n, on peut la couper en deux morceaux A et B avec :
  - longueur$(A) = 1$ et longueur$(B) = n - 1$; ou
  - longueur$(A) = 2$ et longueur$(B) = n - 2$; ou

  $\vdots$

  - longueur$(A) = n - 1$ et longueur$(B) = 1$; ou
  - longueur$(A) = n$ et longueur$(B) = 0$ (on ne coupe pas).

Identification de la sous-structure optimale

- Pour obtenir la structure d'une solution optimale, examinons l'impact du choix d'une découpe ;
- Si la tige est de longueur n, on peut la couper en deux morceaux A et B avec :
  - longueur$(A) = 1$ et longueur$(B) = n - 1$; ou
  - longueur$(A) = 2$ et longueur$(B) = n - 2$; ou

  $\vdots$

  - longueur$(A) = n - 1$ et longueur$(B) = 1$; ou
  - longueur$(A) = n$ et longueur$(B) = 0$ (on ne coupe pas).
- On peut donc laisser A entier et chercher une découpe optimale pour B ;

Équation de programmation dynamique

- Rappelons-nous que le but est de maximiser le profit d'une découpe, en connaissant les prix de chaque morceau ;

Équation de programmation dynamique

- Rappelons-nous que le but est de maximiser le profit d'une découpe, en connaissant les prix de chaque morceau ;
- Si une seule découpe était autorisée, on chercherait $\max_{1 \leq i \leq n} prix(i) + prix(n-i)$;

Équation de programmation dynamique

- Rappelons-nous que le but est de maximiser le profit d'une découpe, en connaissant les prix de chaque morceau ;
- Si une seule découpe était autorisée, on chercherait $\max_{1 \leq i \leq n} prix(i) + prix(n-i)$;
- Mais ici, un nombre arbitraire de découpes est permis ;

Équation de programmation dynamique

- Rappelons-nous que le but est de maximiser le profit d'une découpe, en connaissant les prix de chaque morceau ;
- Si une seule découpe était autorisée, on chercherait $\max_{1 \leq i \leq n} prix(i) + prix(n-i)$;
- Mais ici, un nombre arbitraire de découpes est permis ;
- On déduit de la discussion du slide précédent que :

$$profit(n) = \begin{cases} prix(n) & \text{si } n \leq 1, \\ \max_{1 \leq i \leq n} prix(i) + profit(n-i) & \text{sinon.} \end{cases}$$

Première version récursive

$$profit(n) = \begin{cases} prix(n) & \text{si } n \leq 1, \\ \max_{1 \leq i \leq n} prix(i) + profit(n-i) & \text{sinon.} \end{cases}$$

L'équation nous donne directement l'algorithme récursif suivant :

Algorithme 2 : $\textsc{DécoupeOptimaleNaïve}(n, \text{prix})$

Entrées : une longueur naturelle de tige, un tableau de prix donnant pour chaque longueur possible le prix correspondant.

Sortie : le profit maximal réalisable.

`// aucune découpe possible: renvoyer directement le prix`

si $n \leq 1$ **alors renvoyer** prix[n];

`// tester toutes les découpes possibles et garder la meilleure`

meilleur_prix ← prix[n];

pour chaque *i allant de* 1 *à* $n-1$ **faire**

  meilleur_prix ← max(meilleur_prix, prix[i] + $\textsc{DécoupeOptimaleNaïve}(n - i, \text{prix})$);

renvoyer meilleur_prix;

Inconvénients de l'approche naïve

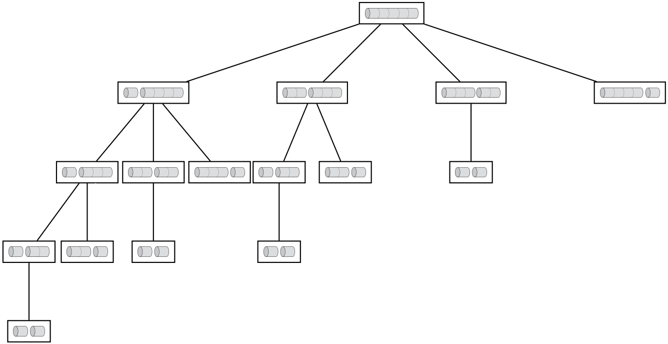
Les calculs de cette première version sont en $O(1)$ ... mais chaque appel récursif en effectue $n-1$;

Inconvénients de l'approche naïve

Les calculs de cette première version sont en $O(1)$ ... mais chaque appel récursif en effectue $n-1$;

Exemple 2

Voici l'arbre d'appels pour une tige de longueur 5 :

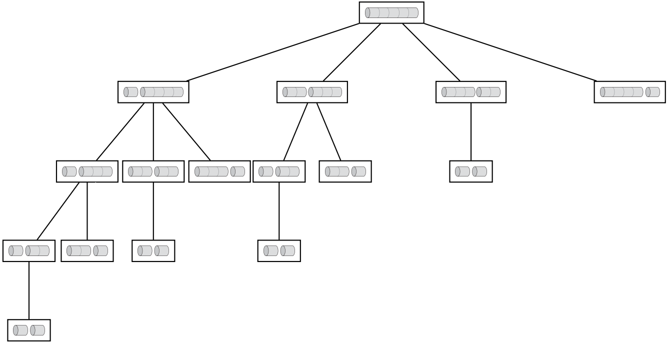


Inconvénients de l'approche naïve

Les calculs de cette première version sont en $O(1)$ ... mais chaque appel récursif en effectue $n-1$;

Exemple 2

Voici l'arbre d'appels pour une tige de longueur 5 :



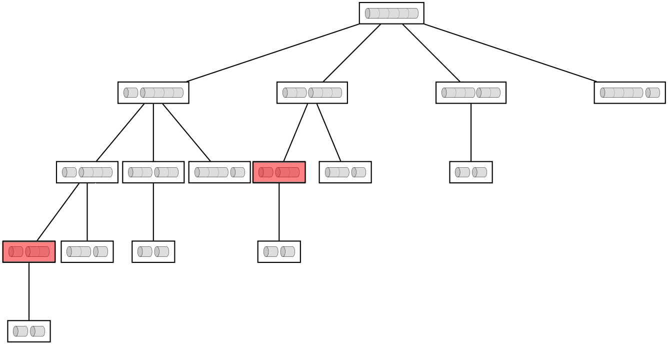
On a donc du $O(n^n)$, qu'on va heureusement pouvoir améliorer.

Inconvénients de l'approche naïve

Les calculs de cette première version sont en $O(1)$ ... mais chaque appel récursif en effectue $n-1$;

Exemple 2

Voici l'arbre d'appels pour une tige de longueur 5 :



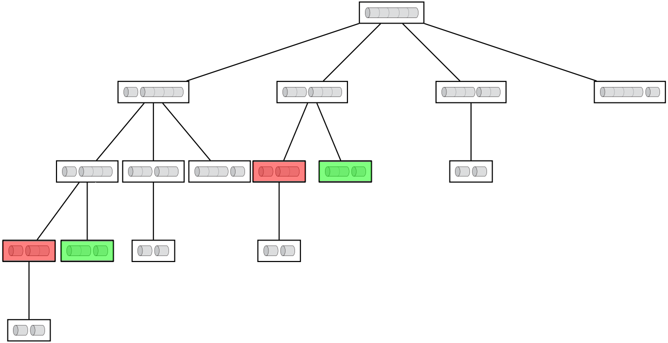
On a donc du $O(n^n)$, qu'on va heureusement pouvoir améliorer.

Inconvénients de l'approche naïve

Les calculs de cette première version sont en $O(1)$ ... mais chaque appel récursif en effectue $n-1$;

Exemple 2

Voici l'arbre d'appels pour une tige de longueur 5 :



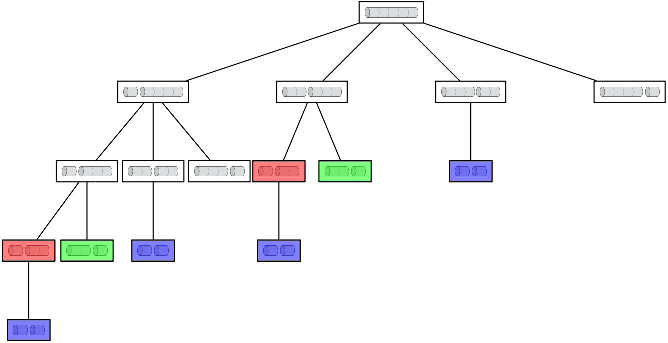
On a donc du $O(n^n)$, qu'on va heureusement pouvoir améliorer.

Inconvénients de l'approche naïve

Les calculs de cette première version sont en $O(1)$ ... mais chaque appel récursif en effectue $n-1$;

Exemple 2

Voici l'arbre d'appels pour une tige de longueur 5 :



On a donc du $O(n^n)$, qu'on va heureusement pouvoir améliorer.

Stockage des solutions aux sous-problèmes

On stocke dans un tableau "profits" la solution optimale pour chaque sous-problème utile dans la suite, rempli par longueur croissante :

0. profits[0] $= 0$;

Stockage des solutions aux sous-problèmes

On stocke dans un tableau "profits" la solution optimale pour chaque sous-problème utile dans la suite, rempli par longueur croissante :

0. profits[0] $= 0$;
1. profits[1] $=$ prix[1] (aucune découpe permise) ;

Stockage des solutions aux sous-problèmes

On stocke dans un tableau "profits" la solution optimale pour chaque sous-problème utile dans la suite, rempli par longueur croissante :

0. profits[0] = 0 ;
1. profits[1] = prix[1] (aucune découpe permise) ;
2. deux choix : aucune découpe, ou deux morceaux de longueur 1
   $\Rightarrow$ profits[2] = max(prix[2], prix[1]+profits[1]) ;

Stockage des solutions aux sous-problèmes

On stocke dans un tableau "profits" la solution optimale pour chaque sous-problème utile dans la suite, rempli par longueur croissante :

0. profits[0] = 0 ;
1. profits[1] = prix[1] (aucune découpe permise) ;
2. deux choix : aucune découpe, ou deux morceaux de longueur 1
   $\Rightarrow$ profits[2] = max(prix[2], prix[1]+profits[1]) ;
3. trois choix : aucune découpe, "1 + découper 2", ou "2 + découper 1" :
   $\Rightarrow$ profits[3] = max(prix[3], prix[1]+profits[2], prix[2]+profits[1]) ;

Stockage des solutions aux sous-problèmes

On stocke dans un tableau “profits” la solution optimale pour chaque sous-problème utile dans la suite, rempli par longueur croissante :

0. profits[0] $=$ 0 ;
1. profits[1] $=$ prix[1] (aucune découpe permise) ;
2. deux choix : aucune découpe, ou deux morceaux de longueur 1
   $\Rightarrow$ profits[2] $=$ max(prix[2], prix[1]+profits[1]) ;
3. trois choix : aucune découpe, “1 + découper 2”, ou “2 + découper 1” :
   $\Rightarrow$ profits[3] $=$ max(prix[3], prix[1]+profits[2], prix[2]+profits[1]) ;

$\vdots$

Programmation dynamique pour le problème de découpe

On obtient donc l'algorithme suivant.

Algorithme 3 : $\textsc{DécoupeOptimaleProgDyn}(n, \text{prix})$

Entrées : une longueur naturelle de tige n, un tableau de prix donnant pour chaque longueur possible le prix correspondant.

Sortie : le profit maximal réalisable.

profits $\leftarrow$ prix;
pour chaque *k allant de* 2 *à* n **faire**
  pour chaque *i allant de* 1 *à* $k-1$ **faire**
    profits$[k] \leftarrow \max($profits$[k]$, prix$[i]$ + profits$[k-i])$;
renvoyer profits$[n]$;

Pour une longueur valant n, on consomme cette fois-ci un espace mémoire de l'ordre de $O(n)$, ce qui nous permet d'éviter les appels récursifs, les calculs inutiles, et de ramener la complexité à du $O(n^2)$.

Distance d'édition

La **distance d'édition**, ou **distance de Levenshtein**, entre deux chaînes de caractères est le plus petit nombre d'insertions, de suppressions et de modifications de caractères nécessaires à la transformation d'une de ces chaînes en l'autre.

Distance d'édition

La **distance d'édition**, ou **distance de Levenshtein**, entre deux chaînes de caractères est le plus petit nombre d'insertions, de suppressions et de modifications de caractères nécessaires à la transformation d'une de ces chaînes en l'autre.

Exemple 3

demain matin

Distance d'édition

La **distance d'édition**, ou **distance de Levenshtein**, entre deux chaînes de caractères est le plus petit nombre d'insertions, de suppressions et de modifications de caractères nécessaires à la transformation d'une de ces chaînes en l'autre.

Exemple 3

$$\text{demain} \xrightarrow{\quad 6 \quad} \emptyset \xrightarrow{\quad 5 \quad} \text{matin}$$

Distance d'édition

La **distance d'édition**, ou **distance de Levenshtein**, entre deux chaînes de caractères est le plus petit nombre d'insertions, de suppressions et de modifications de caractères nécessaires à la transformation d'une de ces chaînes en l'autre.

Exemple 3

$$\begin{tikzcd} \text{demain} \arrow[r, "6"] \arrow[dr, "2"'] & \emptyset \arrow[r, "5"] & \text{matin} \\ & \text{main} \arrow[ur, "1"'] & \end{tikzcd}$$

Approche récursive

Tentons de construire un algorithme récursif pour calculer la distance de S à T.

- Cas de base :

Approche récursive

Tentons de construire un algorithme récursif pour calculer la distance de S à T.

- Cas de base :
  1. S est vide : on ne peut obtenir T qu'en insérant tous les caractères de $T \Rightarrow$ coût : $|T|$ opérations

Approche récursive

Tentons de construire un algorithme récursif pour calculer la distance de S à T.

- Cas de base :

  1. S est vide : on ne peut obtenir T qu'en insérant tous les caractères de $T \Rightarrow$ coût : $|T|$ opérations
  2. T est vide : on ne peut obtenir T qu'en supprimant tous les caractères de $S \Rightarrow$ coût : $|S|$ opérations.

Approche récursive

Tentons de construire un algorithme récursif pour calculer la distance de S à T.

- Cas de base :
  1. S est vide : on ne peut obtenir T qu'en insérant tous les caractères de $T \Rightarrow$ coût : $|T|$ opérations
  2. T est vide : on ne peut obtenir T qu'en supprimant tous les caractères de $S \Rightarrow$ coût : $|S|$ opérations.
- Cas général :

Approche récursive

Tentons de construire un algorithme récursif pour calculer la distance de S à T.

- Cas de base :
  1. S est vide : on ne peut obtenir T qu'en insérant tous les caractères de $T \Rightarrow$ coût : $|T|$ opérations
  2. T est vide : on ne peut obtenir T qu'en supprimant tous les caractères de $S \Rightarrow$ coût : $|S|$ opérations.
- Cas général :
  - les opérations peuvent se produire à n'importe quel endroit, mais nous pouvons choisir l'ordre dans lequel nous les examinons ;

Approche récursive

Tentons de construire un algorithme récursif pour calculer la distance de S à T.

- Cas de base :
  1. S est vide : on ne peut obtenir T qu'en insérant tous les caractères de $T \Rightarrow$ coût : $|T|$ opérations
  2. T est vide : on ne peut obtenir T qu'en supprimant tous les caractères de $S \Rightarrow$ coût : $|S|$ opérations.
- Cas général :
  - les opérations peuvent se produire à n'importe quel endroit, mais nous pouvons choisir l'ordre dans lequel nous les examinons ;
  - nous allons donc les examiner par la fin et baser nos appels récursifs sur des préfixes de nos chaînes ;

Exploitation des sous-problèmes

Notons S^* la chaîne obtenue en supprimant le dernier caractère de S (pareil pour T). On a trois choix possibles :

d e m a i ~~n~~	d e m a i n	d e m a i n
m a t i n	m a t i ~~n~~	m a t i n
(1)	(2)	(3)

Exploitation des sous-problèmes

Notons S^* la chaîne obtenue en supprimant le dernier caractère de S (pareil pour T). On a trois choix possibles :

(1)	(2)	(3)
d e m a i ~~n~~	d e m a i n	d e m a i n
m a t i n	m a t i ~~n~~	m a t i n

1. supprimer S_k coûte $1 + d(S^*, T)$;

Exploitation des sous-problèmes

Notons S^* la chaîne obtenue en supprimant le dernier caractère de S (pareil pour T). On a trois choix possibles :

d e m a i ~~n~~	d e m a i n	d e m a i n
m a t i n	m a t i ~~n~~	m a t i n
(1)	(2)	(3)

1. supprimer S_k coûte $1 + d(S^*, T)$;
2. supprimer T_n coûte $1 + d(S, T^*)$;

Exploitation des sous-problèmes

Notons S^* la chaîne obtenue en supprimant le dernier caractère de S (pareil pour T). On a trois choix possibles :

d e m a i ~~n~~	d e m a i n	d e m a i n
m a t i n	m a t i ~~n~~	m a t i n
(1)	(2)	(3)

1. supprimer S_k coûte $1 + d(S^*, T)$;
2. supprimer T_n coûte $1 + d(S, T^*)$;
3. associer S_k et T_n coûte leur différence plus $d(S^*, T^*)$;

Première version récursive

Ceci nous donne donc l'équation suivante pour calculer la distance qui nous intéresse :

$$d(S, T) = \begin{cases} |S| & \text{si } |T| = 0, \\ |T| & \text{si } |S| = 0, \\ \min(1 + d(S^*, T), 1 + d(S, T^*), 1_{S_{k-1} \neq T_{n-1}} + d(S^*, T^*)) & \text{sinon.} \end{cases}$$

Première version récursive

Ceci nous donne donc l'équation suivante pour calculer la distance qui nous intéresse :

$$d(S, T) = \begin{cases} |S| & \text{si } |T| = 0, \\ |T| & \text{si } |S| = 0, \\ \min(1 + d(S^*, T), 1 + d(S, T^*), 1_{S_{k-1} \neq T_{n-1}} + d(S^*, T^*)) & \text{sinon.} \end{cases}$$

Algorithme 4 : $\textsc{DistanceEditionNaïve}(S,\ k,\ T,\ n)$

Entrées : deux chaînes de caractères S et T et leurs longueurs (respectivement k et n)

Sortie : la distance d'édition entre S et T

si $k = 0$ **alors renvoyer** n;
si $n = 0$ **alors renvoyer** k;
option1 $\leftarrow 1 + \textsc{DistanceEditionNaïve}(S,\ k-1,\ T,\ n)$;
option2 $\leftarrow 1 + \textsc{DistanceEditionNaïve}(S,\ k,\ T,\ n-1)$;
option3 $\leftarrow (S[k-1] \neq T[n-1]) + \textsc{DistanceEditionNaïve}(S,\ k-1,\ T,\ n-1)$;
renvoyer min(option1, option2, option3);

Amélioration

- Sans surprise, la complexité de l'approche naïve est catastrophique : $O(3^{\min(k,n)})$

Amélioration

- Sans surprise, la complexité de l'approche naïve est catastrophique : $O(3^{\min(k,n)})$
- Pour l'accélérer, on va stocker les résultats des solutions optimales des sous-problèmes examinés ;

Amélioration

- Sans surprise, la complexité de l'approche naïve est catastrophique : $O(3^{\min(k,n)})$
- Pour l'accélérer, on va stocker les résultats des solutions optimales des sous-problèmes examinés ;
- On utilisera une matrice stockant la distance entre tous les préfixes de S et de T ;

Construction de la matrice de distances

- On insère un caractère vide au début de S et de T ;

		m	a	t	i	n
	0	1	2	3	4	5
d	1	1	2	3	4	5
e	2	2	2	3	4	5
m	3	2	3	3	4	5
a	4	3	2	3	4	5
i	5	4	3	3	3	4
n	6	5	4	4	4	3

Construction de la matrice de distances

- On insère un caractère vide au début de S et de T ;
- La première ligne correspond au cas de base "S est vide" ;

		m	a	t	i	n
	0	1	2	3	4	5
d	1	1	2	3	4	5
e	2	2	2	3	4	5
m	3	2	3	3	4	5
a	4	3	2	3	4	5
i	5	4	3	3	3	4
n	6	5	4	4	4	3

Construction de la matrice de distances

- On insère un caractère vide au début de S et de T ;
- La première ligne correspond au cas de base "S est vide" ;
- La première colonne correspond au cas de base "T est vide" ;

		m	a	t	i	n
	0	1	2	3	4	5
d	1	1	2	3	4	5
e	2	2	2	3	4	5
m	3	2	3	3	4	5
a	4	3	2	3	4	5
i	5	4	3	3	3	4
n	6	5	4	4	4	3

Construction de la matrice de distances

- On insère un caractère vide au début de S et de T ;
- La première ligne correspond au cas de base "S est vide" ;
- La première colonne correspond au cas de base "T est vide" ;
- Pour chaque autre case $M[i][j]$, on garde le minimum entre :

		m	a	t	i	n
	0	1	2	3	4	5
d	1	1	2	3	4	5
e	2	2	2	3	4	5
m	3	2	3	3	4	5
a	4	3	2	3	4	5
i	5	4	3	3	3	4
n	6	5	4	4	4	3

Construction de la matrice de distances

- On insère un caractère vide au début de S et de T ;
- La première ligne correspond au cas de base "S est vide" ;
- La première colonne correspond au cas de base "T est vide" ;
- Pour chaque autre case $M[i][j]$, on garde le minimum entre :
  - $M[i-1][j]+1$ (insérer un caractère à la fin de S^*) ;

		m	a	t	i	n
	0	1	2	3	4	5
d	1	1	2	3	4	5
e	2	2	2	3	4	5
m	3	2	3	3	4	5
a	4	3	2	3	4	5
i	5	4	3	3	3	4
n	6	5	4	4	4	3

Construction de la matrice de distances

- On insère un caractère vide au début de S et de T ;
- La première ligne correspond au cas de base "S est vide" ;
- La première colonne correspond au cas de base "T est vide" ;
- Pour chaque autre case $M[i][j]$, on garde le minimum entre :
  - $M[i-1][j]+1$ (insérer un caractère à la fin de S^*) ;
  - $M[i][j-1]+1$ (insérer un caractère à la fin de T^*) ;

		m	a	t	i	n
	0	1	2	3	4	5
d	1	1	2	3	4	5
e	2	2	2	3	4	5
m	3	2	3	3	4	5
a	4	3	2	3	4	5
i	5	4	3	3	3	4
n	6	5	4	4	4	3

Construction de la matrice de distances

- On insère un caractère vide au début de S et de T ;
- La première ligne correspond au cas de base "S est vide" ;
- La première colonne correspond au cas de base "T est vide" ;
- Pour chaque autre case $M[i][j]$, on garde le minimum entre :
  - $M[i-1][j]+1$ (insérer un caractère à la fin de S^*) ;
  - $M[i][j-1]+1$ (insérer un caractère à la fin de T^*) ;
  - $M[i-1][j-1]+(S[i] \neq T[j])$ (faire correspondre les derniers caractères) ;

		m	a	t	i	n
	0	1	2	3	4	5
d	1	1	2	3	4	5
e	2	2	2	3	4	5
m	3	2	3	3	4	5
a	4	3	2	3	4	5
i	5	4	3	3	3	4
n	6	5	4	4	4	3

Calcul à l'aide de la matrice de distances

Algorithme 5 : DistanceEditionProgDyn(S, T)

Entrées : deux chaînes de caractères S et T et leurs longueurs (respectivement k et n)
Sortie : la distance d'édition entre S et T

`/* initialiser la matrice de coûts avec les cas de base */`
$k \leftarrow |S|$;
$n \leftarrow |T|$;
coûts $\leftarrow$ matrice de zéros de dimension $(k+1) \times (n+1)$;
pour chaque i *allant de* 0 *à* k **faire** coûts[i][0] $\leftarrow i$;
pour chaque j *allant de* 0 *à* n **faire** coûts[0][j] $\leftarrow j$;
`/* remplir la matrice de coûts à l'aide de la récurrence */`
pour chaque i *allant de* 1 *à* k **faire**
    pour chaque j *allant de* 1 *à* n **faire**
        option1 $\leftarrow 1 +$ coûts[$i-1$][j];
        option2 $\leftarrow 1 +$ coûts[i][$j-1$];
        option3 $\leftarrow (S[i-1] \neq T[j-1]) +$ coûts[$i-1$][$j-1$];
        coûts[i][j] $\leftarrow$ min(option1, option2, option3);
renvoyer coûts[k][n];

Complexité de l'algorithme de programmation dynamique

- Complexité : $O(kn)$, espace : $O(kn)$
- On peut améliorer les choses pour l'espace, car on n'a pas besoin de *toute* la matrice du début à la fin de l'algorithme ;
- Par contre, il n'est pas possible de faire mieux (à moins que ...) pour la complexité [1] ;

Obtention des *solutions* optimales

- Les algorithmes de programmation dynamique vus jusqu'ici ne donnent que la *valeur* d'une solution optimale ;

Obtention des *solutions* optimales

- Les algorithmes de programmation dynamique vus jusqu'ici ne donnent que la *valeur* d'une solution optimale ;
- Pour obtenir une *solution explicite*, on doit pouvoir reconstruire les choix faits à chaque étape ;

Obtention des *solutions* optimales

- Les algorithmes de programmation dynamique vus jusqu'ici ne donnent que la *valeur* d'une solution optimale ;
- Pour obtenir une *solution explicite*, on doit pouvoir reconstruire les choix faits à chaque étape ;
- On peut :

Obtention des *solutions* optimales

- Les algorithmes de programmation dynamique vus jusqu'ici ne donnent que la *valeur* d'une solution optimale ;
- Pour obtenir une *solution explicite*, on doit pouvoir reconstruire les choix faits à chaque étape ;
- On peut :
  1. parcourir la solution de la fin vers le début en reconstruisant les choix ; ou

Obtention des *solutions* optimales

- Les algorithmes de programmation dynamique vus jusqu'ici ne donnent que la *valeur* d'une solution optimale ;
- Pour obtenir une *solution explicite*, on doit pouvoir reconstruire les choix faits à chaque étape ;
- On peut :
  1. parcourir la solution de la fin vers le début en reconstruisant les choix ; ou
  2. stocker explicitement, lors de la construction de la structure contenant les solutions optimales aux sous-problèmes, les décisions prises.

Découpe optimale

- Reprenons le problème de découpe d'une tige ;

Découpe optimale

- Reprenons le problème de découpe d'une tige ;
- Chaque opération découpe un morceau en deux : A, qu'on ne découpe plus, et B, qu'on découpe de manière optimale ;

Découpe optimale

- Reprenons le problème de découpe d'une tige ;
- Chaque opération découpe un morceau en deux : A, qu'on ne découpe plus, et B, qu'on découpe de manière optimale ;
- Pour reconstruire la découpe optimale, on stocke la taille d'un des deux morceaux de chaque découpe optimale, puis :

Découpe optimale

- Reprenons le problème de découpe d'une tige ;
- Chaque opération découpe un morceau en deux : A, qu'on ne découpe plus, et B, qu'on découpe de manière optimale ;
- Pour reconstruire la découpe optimale, on stocke la taille d'un des deux morceaux de chaque découpe optimale, puis :
  - on affiche la longueur k de la dernière pièce non coupée de la solution pour n ;

Découpe optimale

- Reprenons le problème de découpe d'une tige ;
- Chaque opération découpe un morceau en deux : A, qu'on ne découpe plus, et B, qu'on découpe de manière optimale ;
- Pour reconstruire la découpe optimale, on stocke la taille d'un des deux morceaux de chaque découpe optimale, puis :
  - on affiche la longueur k de la dernière pièce non coupée de la solution pour n ;
  - on répète l'opération pour le morceau de longueur $n - k$;

Découpe optimale

- Reprenons le problème de découpe d'une tige ;
- Chaque opération découpe un morceau en deux : A, qu'on ne découpe plus, et B, qu'on découpe de manière optimale ;
- Pour reconstruire la découpe optimale, on stocke la taille d'un des deux morceaux de chaque découpe optimale, puis :
  - on affiche la longueur k de la dernière pièce non coupée de la solution pour n ;
  - on répète l'opération pour le morceau de longueur $n - k$;
  - et ainsi de suite jusqu'à ce qu'on ait tout reconstruit ;

Utilisation des longueurs des derniers morceaux

Si l'on modifie l'algorithme vu précédemment pour qu'il stocke la taille de la dernière pièce de la découpe, on obtient en plus des données de départ la table suivante :

longueur	0	1	2	3	4	5	6	7	8	9
prix	0	1	5	8	9	10	17	17	20	24
profit	0	1	5	8	10	13	17	18	22	25

Utilisation des longueurs des derniers morceaux

Si l'on modifie l'algorithme vu précédemment pour qu'il stocke la taille de la dernière pièce de la découpe, on obtient en plus des données de départ la table suivante :

longueur	0	1	2	3	4	5	6	7	8	9
prix	0	1	5	8	9	10	17	17	20	24
profit	0	1	5	8	10	13	17	18	22	25
dernière taille	0	1	2	3	2	2	6	1	2	3

Utilisation des longueurs des derniers morceaux

Si l'on modifie l'algorithme vu précédemment pour qu'il stocke la taille de la dernière pièce de la découpe, on obtient en plus des données de départ la table suivante :

longueur	0	1	2	3	4	5	6	7	8	9
prix	0	1	5	8	9	10	17	17	20	24
profit	0	1	5	8	10	13	17	18	22	25
dernière taille	0	1	2	3	2	2	6	1	2	3

Utilisation des longueurs des derniers morceaux

Si l'on modifie l'algorithme vu précédemment pour qu'il stocke la taille de la dernière pièce de la découpe, on obtient en plus des données de départ la table suivante :

longueur	0	1	2	3	4	5	6	7	8	9
prix	0	1	5	8	9	10	17	17	20	24
profit	0	1	5	8	10	13	17	18	22	25
dernière taille	0	1	2	3	2	2	6	1	2	3

Algorithme modifié

L'algorithme vu précédemment doit donc légèrement changer pour enregistrer ces informations :

Algorithme 6 : DÉCOUPEOPTIMALESOLUTION(n, prix)

Entrées : une longueur naturelle de tige, un tableau de prix donnant pour chaque longueur possible le prix correspondant.

Sortie : les profits maximaux réalisables pour chaque longueur de tige, ainsi que la longueur du dernier morceau pour chaque découpe optimale.

```
profits ← prix;
taille_dernière_pièce ← tableau(n + 1, 0);
pour chaque k allant de 2 à n faire
    bénéfice ← prix[k];
    taille_dernière_pièce[k] ← k;
    pour chaque i allant de 1 à k − 1 faire
        si prix[i] + profits[k − i] > bénéfice alors
            bénéfice ← prix[i] + profits[k − i];
            taille_dernière_pièce[k] ← i;
    profits[k] ← bénéfice;
renvoyer (profits, taille_dernière_pièce);
```

Distance d'édition

- La même technique pourrait s'appliquer à la distance d'édition ;
- Mais cela impliquerait de stocker une matrice de $O(kn)$ cases supplémentaire ;
- On va donc partir de la matrice de coûts qu'on doit de toute façon calculer et reconstruire les choix qui nous ont mené à la solution optimale à partir de la fin :

  $\nwarrow$: supprime le dernier caractère de S et le dernier caractère de T ;

  $\leftarrow$: supprime le dernier caractère de T ;

  $\uparrow$: supprime le dernier caractère de S.

Reconstruction d'un chemin optimal

Reprenons la matrice de coûts déjà vue, et partons de la dernière case :

		m	a	t	i	n
	0	1	2	3	4	5
d	1	1	2	3	4	5
e	2	2	2	3	4	5
m	3	2	3	3	4	5
a	4	3	2	3	4	5
i	5	4	3	3	3	4
n	6	5	4	4	4	3

S = d e m a i n

T = m a t i n

Reconstruction d'un chemin optimal

Reprenons la matrice de coûts déjà vue, et partons de la dernière case :

		m	a	t	i	n
	0	1	2	3	4	5
d	1	1	2	3	4	5
e	2	2	2	3	4	5
m	3	2	3	3	4	5
a	4	3	2	3	4	5
i	5	4	3	3	3	4
n	6	5	4	4	4	3

S = d e m a i n

T = m a t i n

Reconstruction d'un chemin optimal

Reprenons la matrice de coûts déjà vue, et partons de la dernière case :

		m	a	t	i	n
	0	1	2	3	4	5
d	1	1	2	3	4	5
e	2	2	2	3	4	5
m	3	2	3	3	4	5
a	4	3	2	3	4	5
i	5	4	3	3	3	4
n	6	5	4	4	4	3

S = d e m a i n

T = m a t i n

Reconstruction d'un chemin optimal

Reprenons la matrice de coûts déjà vue, et partons de la dernière case :

		m	a	t	i	n
	0	1	2	3	4	5
d	1	1	2	3	4	5
e	2	2	2	3	4	5
m	3	2	3	3	4	5
a	4	3	2	3	4	5
i	5	4	3	3	3	4
n	6	5	4	4	4	3

S = d e m a i n

T = m a t i n

Reconstruction d'un chemin optimal

Reprenons la matrice de coûts déjà vue, et partons de la dernière case :

		m	a	t	i	n
	0	1	2	3	4	5
d	1	1	2	3	4	5
e	2	2	2	3	4	5
m	3	2	3	3	4	5
a	4	3	2	3	4	5
i	5	4	3	3	3	4
n	6	5	4	4	4	3

S = d e m a i n

T = m a t i n

Reconstruction d'un chemin optimal

Reprenons la matrice de coûts déjà vue, et partons de la dernière case :

		m	a	t	i	n
	0	1	2	3	4	5
d	1	1	2	3	4	5
e	2	2	2	3	4	5
m	3	2	3	3	4	5
a	4	3	2	3	4	5
i	5	4	3	3	3	4
n	6	5	4	4	4	3

S = d e m a i n

T = m a t i n

Reconstruction d'un chemin optimal

Reprenons la matrice de coûts déjà vue, et partons de la dernière case :

		m	a	t	i	n
	0	1	2	3	4	5
d	1	1	2	3	4	5
e	2	2	2	3	4	5
m	3	2	3	3	4	5
a	4	3	2	3	4	5
i	5	4	3	3	3	4
n	6	5	4	4	4	3

S = d e m a i n

T = m a t i n

Reconstruction d'un chemin optimal

Reprenons la matrice de coûts déjà vue, et partons de la dernière case :

		m	a	t	i	n
	0	1	2	3	4	5
d	1	1	2	3	4	5
e	2	2	2	3	4	5
m	3	2	3	3	4	5
a	4	3	2	3	4	5
i	5	4	3	3	3	4
n	6	5	4	4	4	3

S = d e m a i n

T = m a t i n

Reconstruction d'un chemin optimal

Reprenons la matrice de coûts déjà vue, et partons de la dernière case :

		m	a	t	i	n
	0	1	2	3	4	5
d	1	1	2	3	4	5
e	2	2	2	3	4	5
m	3	2	3	3	4	5
a	4	3	2	3	4	5
i	5	4	3	3	3	4
n	6	5	4	4	4	3

S = d e m a i n

opérations = s s a a i a a

T = m a t i n

Chemin suivi à l'envers : $\uparrow\uparrow\nwarrow\nwarrow\leftarrow\nwarrow\nwarrow\equiv$ "ssaaiaa" (**s**uppression (dans S), **a**ssociation, et **i**nsertion (dans S)).

Reconstruction d'un chemin optimal

Reprenons la matrice de coûts déjà vue, et partons de la dernière case :

		m	a	t	i	n
	0	1	2	3	4	5
d	1	1	2	3	4	5
e	2	2	2	3	4	5
m	3	2	3	3	4	5
a	4	3	2	3	4	5
i	5	4	3	3	3	4
n	6	5	4	4	4	3

S = ~~d~~ e m a i n

opérations = s s a a i a a

T = m a t i n

Chemin suivi à l'envers : $\uparrow\uparrow\nwarrow\nwarrow\leftarrow\nwarrow\nwarrow \equiv$ "ssaaiaa" (**s**uppression (dans S), **a**ssociation, et **i**nsertion (dans S)).

Reconstruction d'un chemin optimal

Reprenons la matrice de coûts déjà vue, et partons de la dernière case :

		m	a	t	i	n
	0	1	2	3	4	5
d	1	1	2	3	4	5
e	2	2	2	3	4	5
m	3	2	3	3	4	5
a	4	3	2	3	4	5
i	5	4	3	3	3	4
n	6	5	4	4	4	3

S = ~~d~~ ~~e~~ m a i n

opérations = s s a a i a a

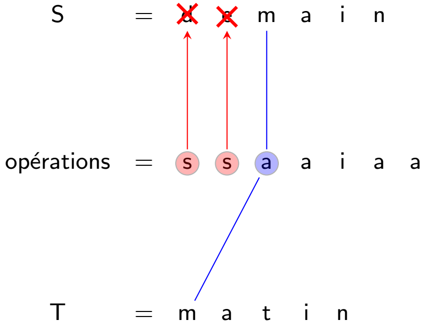
T = m a t i n

Chemin suivi à l'envers : ↑↑↖↖←↖↖≡ "ssaaiaa" (**s**uppression (dans S), **a**ssociation, et **i**nsertion (dans S)).

Reconstruction d'un chemin optimal

Reprenons la matrice de coûts déjà vue, et partons de la dernière case :

		m	a	t	i	n
	0	1	2	3	4	5
d	1	1	2	3	4	5
e	2	2	2	3	4	5
m	3	2	3	3	4	5
a	4	3	2	3	4	5
i	5	4	3	3	3	4
n	6	5	4	4	4	3

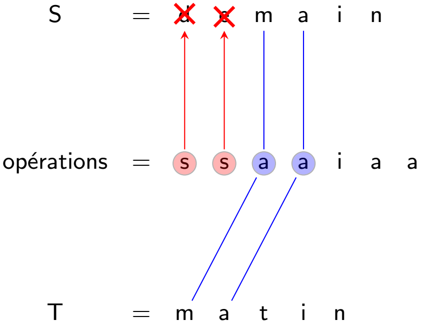


Chemin suivi à l'envers : $\uparrow\uparrow\nwarrow\nwarrow\leftarrow\nwarrow\nwarrow\equiv$ "ssaaiaa" (**s**uppression (dans S), **a**ssociation, et **i**nsertion (dans S)).

Reconstruction d'un chemin optimal

Reprenons la matrice de coûts déjà vue, et partons de la dernière case :

		m	a	t	i	n
	0	1	2	3	4	5
d	1	1	2	3	4	5
e	2	2	2	3	4	5
m	3	2	3	3	4	5
a	4	3	2	3	4	5
i	5	4	3	3	3	4
n	6	5	4	4	4	3

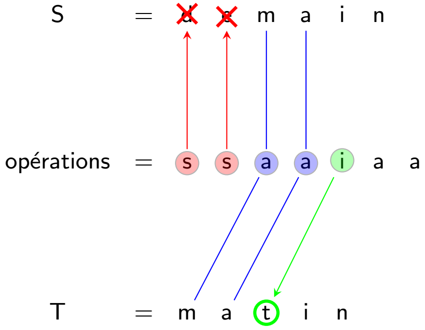


Chemin suivi à l'envers : $\uparrow\uparrow\nwarrow\nwarrow\leftarrow\nwarrow\nwarrow\equiv$ "ssaaiaa" (**s**uppression (dans S), **a**ssociation, et **i**nsertion (dans S)).

Reconstruction d'un chemin optimal

Reprenons la matrice de coûts déjà vue, et partons de la dernière case :

		m	a	t	i	n
	0	1	2	3	4	5
d	1	1	2	3	4	5
e	2	2	2	3	4	5
m	3	2	3	3	4	5
a	4	3	2	3	4	5
i	5	4	3	3	3	4
n	6	5	4	4	4	3

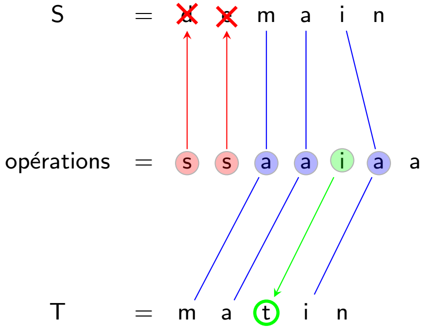


Chemin suivi à l'envers : ↑↑↖↖←↖↖≡ "ssaaiaa" (**s**uppression (dans S), **a**ssociation, et **i**nsertion (dans S)).

Reconstruction d'un chemin optimal

Reprenons la matrice de coûts déjà vue, et partons de la dernière case :

		m	a	t	i	n
	0	1	2	3	4	5
d	1	1	2	3	4	5
e	2	2	2	3	4	5
m	3	2	3	3	4	5
a	4	3	2	3	4	5
i	5	4	3	3	3	4
n	6	5	4	4	4	3

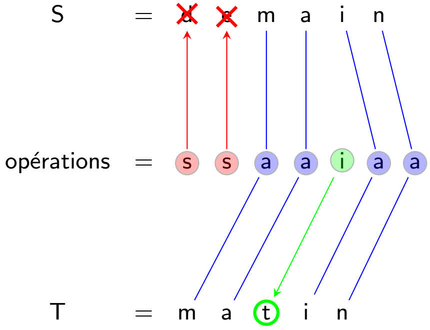


Chemin suivi à l'envers : ↑↑↖↖←↖↖≡ "ssaaiaa" (**s**uppression (dans S), **a**ssociation, et **i**nsertion (dans S)).

Reconstruction d'un chemin optimal

Reprenons la matrice de coûts déjà vue, et partons de la dernière case :

		m	a	t	i	n
	0	1	2	3	4	5
d	1	1	2	3	4	5
e	2	2	2	3	4	5
m	3	2	3	3	4	5
a	4	3	2	3	4	5
i	5	4	3	3	3	4
n	6	5	4	4	4	3



Chemin suivi à l'envers : $\uparrow\uparrow\nwarrow\nwarrow\leftarrow\nwarrow\nwarrow \equiv$ "ssaaiaa" (**s**uppression (dans S), **a**ssociation, et **i**nsertion (dans S)).

Bilan de la programmation dynamique

- La programmation dynamique est un outil puissant quand on peut l'appliquer ;
- Attention, elle ne marche pas toujours : il faut que le problème possède une sous-structure optimale ;
- Même quand elle fonctionne, elle ne nous donne pas nécessairement un algorithme polynomial (sous-problèmes (presque) tous indépendants et / ou espace de recherche trop grand) ;

Bibliographie

[1] Arturs Backurs and Piotr Indyk.
Edit distance cannot be computed in strongly subquadratic time (unless SETH is false).
SIAM Journal on Computing, 47(3) :1087–1097, 2018.

[2] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein.
Introduction to Algorithms.
MIT Press, 3ème edition, 2009.