Langages formels (11X003) - Automne 2025

3. Programmation dynamique (I)

Enseignant: Arnaud Casteigts (et F. Raynaud) *Assistant: Brian Pulfer*
Moniteurs: T. von Düring & A. Maendly

La programmation dynamique (dynamic programming ou DP, en anglais) est une méthode algorithmique pour résoudre des problèmes, le plus souvent d'optimisation, mais pas seulement. Comme pour le paradigme "Divide and Conquer", elle consiste à décomposer un problème en sous-problèmes. Ici, cependant, certains sous-problèmes sont susceptibles d'être résolus de nombreuses fois, l'idée principale est donc de mémoriser ces solutions intermédiaires pour ne les calculer qu'une fois, ce qui s'appelle la mémoïsation. Tout l'art de la programmation dynamique est d'organiser les traitements de sorte à exploiter au mieux ces répétitions. D'une certaine manière, il s'agit de faire du recyclage.

9.1 Exemple 1 : Nombres de Fibonacci

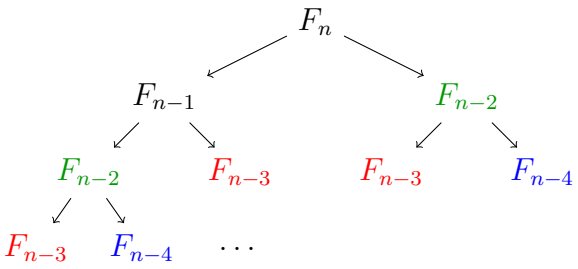
Ce premier exemple n'est pas un problème d'optimisation, mais il illustre bien les concepts de la programmation dynamique. Il s'agit de calculer le n$^{\text{ème}}$ nombre de Fibonacci, ces nombres étant définis inductivement comme suit :

$F_0 = 0$;
$F_1 = 1$;
$F_n = F_{n-1} + F_{n-2}$ pour tout $n \geq 2$.

On utilise souvent ces nombres pour illustrer la récursivité en programmation, avec l'algorithme suivant :

```
Fib(k):
  if k=0:
    return 0
  elif k=1:
    return 1
  else:
    return Fib(k-1) + Fib(k-2)
```

Bien que très élégant et correspondant exactement à la définition, cet algorithme a une complexité en temps catastrophique. Regardons les appels engendrés :



Clairement, les mêmes appels sont effectués de nombreuses fois. Il y a donc beaucoup à gagner si l'on mémorise les résultats intermédiaires ! Analysons d'abord la complexité de cet algorithme. Soit $T(n)$ le temps requis pour calculer le terme d'ordre n. Le calcul de $T(n)$ correspond à la récurrence suivante :

$$T(n) = T(n-1) + T(n-2) + O(1)$$

où $O(1)$ correspond au coût de la somme effectuée à chaque appel (supposé constant). Pour simplifier, observons que $T(n-1)$ est au moins aussi coûteux que $T(n-2)$ (puisque Fib(1) appelera Fib(2)), on a donc $T(n) \geq 2T(n-2)$ et donc $T(n) \geq 2 \times 2 \times \cdots \times 2 = 2^{n/2}$.

La complexité en temps est donc exponentielle. Peut-on faire mieux ?

Voici le même algorithme, avec mémoïsation dans une variable de type dictionnaire (clés,valeurs), initialisée avec la valeur 0 pour la clé 0 et 1 pour la clé 1.

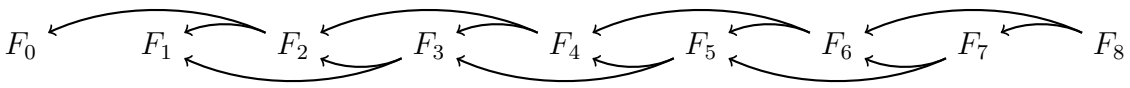
```
dict = {0: 0, 1: 1}

Fib(n):
   if n not in dict:
      dict[n] = Fib(n-1) + Fib(n-2)
   return dict[n]
```

Supposons que la recherche et l'insertion dans le dictionnaire se font en temps constant. Chaque valeur de n ne fait l'objet d'un calcul que la première fois qu'elle est demandée. Les sous-arbres correspondant aux valeurs déjà connues ne seront donc pas explorés, ce qui implique un temps total de $O(n)$ (complexité linéaire). Cet exemple illustre le gain gigantesque que l'on peut faire grâce à la mémoïsation.

L'approche que nous avons utilisé ici est de type "top-down" : nous sommes partis du problème le plus général, et en avons résolu les sous-problèmes récursivement (avec mémoïsation). On peut aussi partir des plus petits sous-problèmes et remonter (sans récursion) vers le problème général, c'est l'approche "bottom-up", donnée par l'algorithme suivant :

```
Fib(n):
  dict = {0: 0, 1: 1}
  for i in range(2, n):
     dict[i] ← dict[i-1] + dict[i-2]
  return dict[n]
```

En fait, n'importe quel problème sujet à la programmation dynamique peut être résolu des deux manières (top-down ou bottom-up), avec essentiellement la même complexité. Pour bien comprendre cela, on peut décrire les dépendances entre sous problèmes par un graphe orienté, dont les nœuds sont des (sous-)problèmes et les arcs sont les dépendances. Par exemple, pour le problème de calculer Fib(8), on a le graphe suivant :



L'approche bottom-up consiste alors à commencer par les sous-problèmes qui ne dépendent de rien et de remonter progressivement jusqu'au problème de départ (qui dépend de tous les autres). Ici, l'ordre des entiers naturels fonctionne, mais en général, cela pourrait être plus complexe. Cependant, pour n'importe quel graphe orienté acyclique, on peut facilement trouver un ordre total compatible en utilisant un algorithme de tri topologique (de complexité linéaire). L'approche top-down, quant à elle, revient à effectuer une récursion à partir du problème le plus général (avec de la mémoïsation). In fine, les deux peuvent être vus comme différentes manières de parcourir le graphe. Notez qu'il s'agit là d'une représentation abstraite : en général, un tel graphe n'est pas construit explicitement, comme dans les deux exemples de codes fournis ci-dessus, qui illustrent ces deux approches.

9.2 Remarques

Espace mémoire. Dans certains cas, on cherche aussi à économiser l'espace. C'est souvent possible. Intuitivement, on peut "oublier" une valeur dès que les problèmes qui en dépendent ont été résolus. Pour Fibonacci, par exemple, on pourrait réduire la complexité en espace à $O(1)$, ce qui n'est pas fait dans le code ci-dessus.

Dépendances cycliques. Pour beaucoup de problèmes, une difficulté importante est d'éviter les dépendances cycliques entre sous-problèmes. En effet, les approches décrites ci-dessus ne marchent que si le graphe est acyclique (DAG, pour directed acyclic graph). Le risque autrement est d'engendrer une exécution infinie. Nous en reparlerons.

9.3 Exemple 2 : Justification de texte

Étant donné une liste de mots qui ont vocation à être dans le même paragraphe. L'objectif est de décider où les lignes doivent être coupées, en supposant que le texte doit être aligné sur les deux marges ("justifié"), sauf la dernière ligne. Pour simplifier, on suppera que les mots ne peuvent pas être coupés. Prenons par exemple le texte suivant :

`texte` = "`xxxx xxxxxx xxxxx xx xxxxxxx xxxxxxxx xxxxxxx xx`"

9.3.1 Algorithme glouton ? (mauvaise qualité)

Tant que la ligne courante ne déborde pas, ajouter des mots. Puis passer à la ligne suivante et continuer avec le texte restant. (Cet algorithme a longtemps été utilisé par MS Word, je ne sais pas s'il l'est toujours.)

Si la largeur disponible est de 21 caractères, par exemple. On obtient le résultat suivant :

```
xxxx xxxxxx xxxxx xxx|
xxxxxxx       xxxxxxx|
xxxxxxx xx.
```

Cette solution n'est pas très esthétique. On préférerait peut-être celle-ci :

```
xxxx   xxxxxx   xxxxx|
xxx  xxxxxxx  xxxxxxx|
xxxxxxx xx.
```

Le système LaTeX [1] utilise de la programmation dynamique pour trouver la meilleure solution, en l'occurrence, la seconde (définie par des critères précis).

9.3.2 Définition plus détaillée du problème

Définissons le problème plus précisément. En entrée, nous recevons une liste de n mots (variable `mots`). En sortie, on peut se contenter de d'indiquer les indices des premiers mots de chaque ligne.

Pour juger de la qualité d'une solution, on peut utiliser une fonction de coût similaire à celle de LaTeX. À tout moment, une ligne candidate correspond à une sous-liste contigüe de mots, noté `mots[i:j]` (du $i^{\text{ème}}$ au $j^{\text{ème}}$ mot). Le coût qu'on lui attribue est le nombre d'espace qu'elle utilise ($+\infty$, si la ligne déborde), élevé à la puissance 3 pour pénaliser plus

1. Utilisé par exemple pour taper ce document, ainsi que tous les supports du cours d'algorithmique.

fortement les lignes qui en ont beaucoup. Le coût d'une solution complète correspond à la somme des coûts de chaque ligne.

Dans l'exemple ci-dessus, cela donne $3^3 + 7^3 + 1^3 = 371$ pour la solution gloutonne et $6^3+4^3+1^3 = 281$ pour l'autre proposition. L'objectif est de minimiser la solution. La seconde est donc meilleure.

9.3.3 Brute force ? (trop coûteux)

Une solution brute force consisterait à tester toutes les façons possibles de couper les lignes. Autrement dit, pour chaque mot, décider s'il commence une nouvelle ligne ou non, ce qui représente essentiellement 2^n possibilités.

9.3.4 Programmation dynamique (parfait !)

On peut résoudre ce problème par un programme dynamique. La seule chose à faire est de le formuler par une récurrence impliquant des sous-problèmes.

Le problème revient alors à :

1. Choisir les mots de la ligne courante
2. Résoudre le sous-problème correspondant au texte restant
3. Parmi ces choix, retenir celui qui minimise le coût (ligne courante + sous-problème)

Cela suggère un programme récursif simple. Mais combien coûterait-il ? En l'état, très cher, car les mêmes sous-problèmes seront résolus de nombreuses fois. Et si l'on utilise la mémoïsation ?

Analysons la complexité avec mémoïsation. Il n'y a que n sous-problèmes possibles (tous les suffixes possibles de la liste de départ). Combien coûte la résolution de chacun si l'on suppose les autres résolus ? À chaque étape, le travail consiste à explorer les choix possibles pour la ligne courante, récupérer le coût du sous-problème correspondant, puis effectuer un minimum. Le nombre de choix possibles à chaque étape étant clairement inférieur à n, ces trois tâches prennent un temps $O(n)$. Bilan : On a n sous-problèmes qui prennent chacun un temps $O(n)$, soit $O(n^2)$ au total, ce qui est beaucoup mieux ! En pratique, le nombre de choix par ligne est assez petit, donc le temps total est quasiment linéaire.

Reconstruire la solution : Vous aurez remarqué que notre algorithme, en l'état, ne construit pas vraiment la solution, il ne fait que calculer son coût minimum. Pour la construire concrètement, il suffirait que chaque sous-problème mémorise le choix qui lui a coûté le moins cher. Il suffirait alors, une fois le calcul terminé, de suivre ces choix depuis le problème initial pour retrouver le chemin complet dans le graphe des sous-problèmes, ce chemin donnant la

solution en temps linéaire également (ici, la liste d'indice correspondant au premier mot de chaque ligne).

9.3.5 Conclusion

On a réussi à résoudre le problème efficacement et (quasiment) sans réfléchir ! Le seul effort a été de le voir comme une récurrence impliquant des sous-problèmes. En fait, tout programme récursif peut être automatiquement converti en une version qui utilise de la mémoïsation. C'est l'un des grands avantages de la programmation dynamique. Notez qu'on pourrait aussi écrire une version bottom-up, c'est parfois plus efficace en pratique car il n'y a pas à exécuter les appels récursifs (bien que la complexité théorique soit la même). Cependant, il est souvent plus naturel de procéder récursivement de manière top-down.

La semaine prochaine, nous verrons d'autres exemples, notamment des exemples où le problème a des dépendances cycliques qui compliquent (un peu) la mise en place d'un programme dynamique.

```
Subproblems = suffices words [i:]
\#subprobs: n (number of words)
\#choices: <= n-i = O(n)
recurrence: DP(i) = min(DP(j) + badness(i:j)) for j in range(i+1,n)
time/subproblem = O(n)
base case DP(n)=0
```

4) topological order : i = n, n-1, 0 total time : O(n**2)

5) orig. problem : DP(0)

Parent pointers : what was the value of j that gave the best min ? → parent[i]=argmin() it is the best value.

How to reconstruct the solution itself : 0 → parent[0] → parent[parent[0]].

This is absolutly general. For any recursion algorithm, can be automatedly converted into a DP algo.