REPUBLIQUE DU CAMEROUN
REPUBLIC OF CAMEROON
Peace-Work-Fatherland
****
UNIVERSITE DE DSCHANG
UNIVERSITY OF DSCHANG
Scholae Thesaurus Dschangensis Ibi Cordum
BP 96, Dschang (Cameroun)
Tél. /Fax (237) 233 45 13 81
Website : http://www.univ-dschang.org.
E-mail : udsrectorat@univ-dschang.org

FACULTE DES SCIENCES
FACULTY OF SCIENCE

Département de Mathématiques et Informatique
Department of mathematics and Computer Science
BP 96, DSc hang (Cameroun)
Tél. /Fax (237) 233 45 13 81
Website : http://fs.univ-dschang.org.
E-mail : dept.math-info@univ-dschang.org

05/01/2022

COMPLEXITE ET ALGORITHMES AVANCES

COMPLEXITE ET ALGORITHMES AVANCES

EQUIPE DE L'EXPOSE

NOM ET PRENOM	MATRICULE
FOPA KUETE DUCLAIR	CM-UDS-18SCI2116
FOKO TEKOUJOU ISMAEL	CM-UDS-18SCI3008
FONGANG CHENDJOU PIERRE-CLAUDE	CM-UDS-18SCI2114
TIABOU DOUMENE LUCELIA ALANA	CM-UDS-18SCI0805

SOUS LA SUPERVISION DE

Pr. KENGNE VIANNEY

ANNEE ACADEMIQUE 2021/2022

Table des matières

INTRODUCTION

Ce chapitre couvre trois techniques importantes utilisées dans la conception et l'analyse d'algorithmes efficaces : la programmation dynamique (Partie I) et les algorithmes gloutons (Partie II). Les chapitre I a présenté d'autres techniques largement applicables, telles que diviser pour régner et comment résoudre les récurrences. Les techniques de ce chapitre sont un peu plus sophistiquées, mais elles nous aident à attaquer de nombreux problèmes de calcul. La programmation dynamique s'applique généralement aux problèmes d'optimisation dans lesquels nous faisons un ensemble de choix afin d'arriver à une solution optimale. Au fur et à mesure que nous faisons chaque choix, des sous-problèmes de la même forme surgissent souvent. La programmation dynamique est efficace lorsqu'un sous-problème donné peut résulter de plus d'un ensemble partiel de choix ; la technique clé consiste à stocker la solution à chacun de ces sous-problèmes au cas où il réapparaîtrait. La Partie I montre comment cette idée simple peut parfois transformer des algorithmes en temps exponentiel en algorithmes en temps polynomial. Comme les algorithmes de programmation dynamique, les algorithmes gloutons s'appliquent généralement aux problèmes d'optimisation dans lesquels nous faisons un ensemble de choix afin d'arriver à une solution optimale. L'idée de l'algorithme de gré à gré est de faire chaque choix de manière localement optimale. Un exemple simple est le changement de pièces : pour minimiser le nombre de pièces nécessaires pour rendre la monnaie d'un montant donné, nous pouvons sélectionner à plusieurs reprises la pièce la plus grande qui n'est pas plus grande que le montant qui reste. Une approche gloutonne fournit une solution optimale pour beaucoup de ces problèmes beaucoup plus rapidement qu'une approche de programmation dynamique. Cependant, nous ne pouvons pas toujours dire facilement si une approche gourmande sera efficace.

PARTIE I : PROGRAMMATION DYNAMIQUE

La programmation dynamique, comme la méthode diviser pour régner, résout les problèmes en combinant les solutions aux sous-problèmes (« La programmation » dans ce contexte fait référence à une méthode tabulaire, pas à l'écriture de code informatique) pour résoudre le problème d'origine. En revanche, la programmation dynamique s'applique lorsque les sous-problèmes se chevauchent, c'est-à-dire lorsque les sous-problèmes partagent des sous-sous-problèmes. Dans ce contexte, un algorithme diviser pour régner fait plus de travail que nécessaire, résolvant à plusieurs reprises les sous-sous-problèmes courants. Un algorithme de programmation dynamique résout chaque sous-sous-problème une seule fois, puis enregistre sa solution dans un tableau, évitant ainsi le travail de recalculer la solution à chaque fois qu'il résout chaque sous-sous-problème. Nous appliquons généralement la programmation dynamique aux problèmes d'optimisation. De tels problèmes peuvent avoir de nombreuses solutions possibles. Chaque solution a une valeur, et nous souhaitons trouver une solution avec la valeur optimale (minimum ou maximum). Nous appelons une telle solution une solution optimale au problème, par opposition à la solution optimale, car il peut y avoir plusieurs solutions qui atteignent la valeur optimale. Lors du développement d'un algorithme de programmation dynamique, nous suivons une séquence de quatre étapes :

1. Caractériser la structure d'une solution optimale.
2. Définir récursivement la valeur d'une solution optimale.
3. Calculez la valeur d'une solution optimale, généralement de manière ascendante.
4. Construire une solution optimale à partir des informations calculées.

Les étapes 1 à 3 constituent la base d'une solution de programmation dynamique à un problème. Si nous n'avons besoin que de la valeur d'une solution optimale, et non de la solution elle-même, nous pouvons omettre l'étape 4. Lorsque nous effectuons l'étape 4, nous conservons parfois des informations supplémentaires au cours de l'étape 3 afin de pouvoir facilement construire une solution optimale. Les sections qui suivent utilisent la méthode de programmation dynamique pour résoudre certains problèmes d'optimisation. La section I examine le problème de la suite de Fibonacci. Compte tenu de ces exemples de programmation dynamique, la section II discute de deux caractéristiques clés qu'un problème doit avoir pour que la programmation dynamique soit une technique de solution viable. Enfin, la section II traite de quelques problèmes classiques de programmation dynamique.

I. Un exemple introductif : Suite de Fibonacci

Prenons l'exemple de la suite de Fibonacci. La suite suivante est la suite de Fibonacci :

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, …

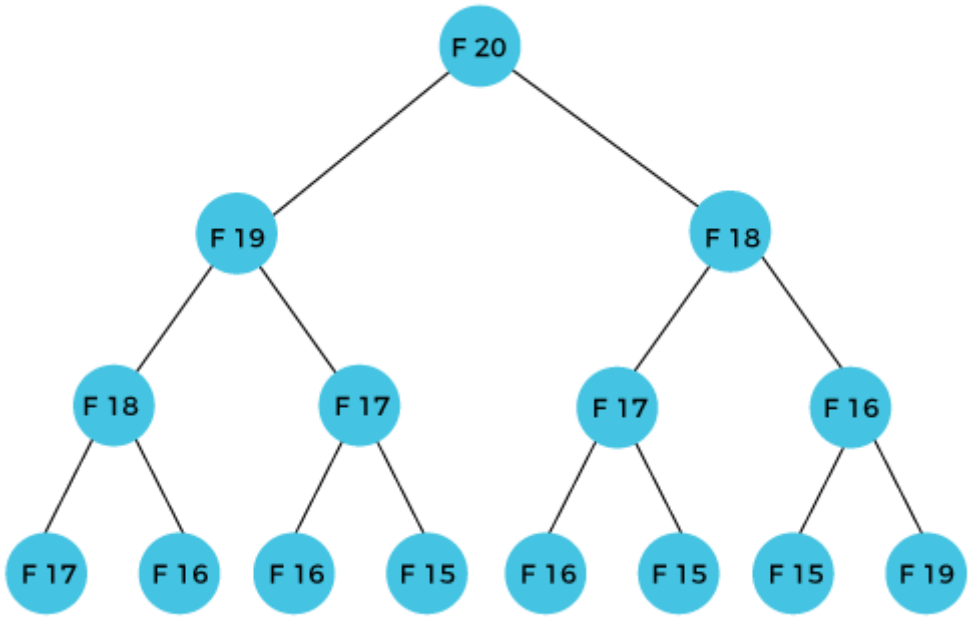
Les nombres de la série ci-dessus ne sont pas calculés au hasard. Mathématiquement, nous pourrions écrire chacun des termes en utilisant la formule ci-dessous :

$F(n) = F(n-1) + F(n-2)$

Avec les valeurs de base $F(0) = 0$ et $F(1) = 1$. Pour calculer les autres nombres, nous suivons la relation ci-dessus. Par exemple, $F(2)$ est la somme $F(0)$ et $F(1)$, qui est égale à 1.

1. Comment peut-on calculer $F(20)$?

Le terme $F(20)$ sera calculé à l'aide de la nième formule de la suite de Fibonacci. La figure ci-dessous montre comment $F(20)$ est calculé.



Comme nous pouvons le constater dans la figure ci-dessus, $F(20)$ est calculé comme la somme de $F(19)$ et $F(18)$. Dans l'approche de programmation dynamique, nous essayons de diviser le problème en sous-problèmes similaires. Si nous récapitulons la définition de la programmation dynamique, cela dit que le sous-problème similaire ne doit pas être calculé plus d'une fois. Pourtant, dans le cas ci-dessus, le sous-problème est calculé deux fois. Dans l'exemple ci-dessus, $F(18)$ est calculé deux fois ; de même, $F(17)$ est également calculé deux fois. Cependant, cette technique est très utile car elle résout les sous-problèmes similaires, mais nous devons être prudents lors du stockage des résultats, car nous ne sommes pas particulièrement soucieux de stocker le résultat que nous avons calculé une fois, cela peut alors entraîner un gaspillage de ressources.

Dans l'exemple ci-dessus, si nous calculons le $F(18)$ dans le sous-arbre de droite, cela conduit à une utilisation considérable des ressources et diminue les performances globales.

La solution au problème ci-dessus consiste à enregistrer les résultats calculés dans un tableau. Tout d'abord, nous calculons $F(16)$ et $F(17)$ et sauvegardons leurs valeurs dans un tableau. Le $F(18)$ est calculé en additionnant les valeurs de $F(17)$ et $F(16)$, qui sont déjà enregistrées dans un tableau. La valeur calculée de $F(18)$ est enregistrée dans un tableau. La valeur de $F(19)$ est calculée en utilisant la somme de $F(18)$ et $F(17)$, et leurs valeurs sont déjà enregistrées dans un tableau. La valeur calculée de $F(19)$ est stockée dans un tableau. La valeur de $F(20)$ peut être calculée en additionnant les valeurs de $F(19)$ et $F(18)$, et les valeurs de $F(19)$ et $F(18)$ sont stockées dans un tableau. La valeur finale calculée de $F(20)$ est stockée dans un tableau.

2. Comment fonctionne l'approche de programmation dynamique ?

Voici les étapes que suit la programmation dynamique :

1. Il décompose le problème complexe en sous-problèmes plus simples.
2. Il trouve la solution optimale à ces sous-problèmes.
3. Il stocke les résultats des sous-problèmes (mémorisation). Le processus de stockage des résultats des sous-problèmes est appelé mémorisation.
4. Il les réutilise pour que le même sous-problème ne soit calculé plus d'une fois.
5. Enfin, il combine les solutions des sous-problèmes pour obtenir la solution du problème complexe.

3. Approches de la programmation dynamique

Il existe deux approches de la programmation dynamique :

- ✍ Une approche descendante (Top-down)
- ✍ Une approche en ascendante (Bottom-up)

Approche descendante

L'approche descendante suit la technique de mémorisation, tandis que l'approche ascendante suit la méthode de tabulation. Ici, la mémorisation est égale à la somme de la récursivité et de la mise en cache. La récursivité signifie appeler la fonction elle-même, tandis que la mise en cache signifie stocker les résultats intermédiaires.

Avantages

- ☑ Il est très facile à comprendre et à mettre en œuvre.
- ☑ Il résout les sous-problèmes uniquement lorsque cela est nécessaire.
- ☑ Il est facile à déboguer.

Inconvénients

- × Il utilise la technique de récursivité qui occupe plus de mémoire dans la pile d'appels. Parfois, lorsque la récursivité est trop profonde, la condition de débordement de pile se produit.
- × Il occupe plus de mémoire qui dégrade les performances globales.

Approche ascendante

L'approche ascendante est également l'une des techniques qui peuvent être utilisées pour mettre en œuvre la programmation dynamique. Il utilise la technique de tabulation pour mettre en œuvre l'approche de programmation dynamique. Il résout le même genre de problèmes mais il supprime la récursivité. Si nous supprimons la récursivité, il n'y a pas de problème de débordement de pile ni de surcharge des fonctions récursives. Dans cette technique de tabulation, nous résolvons les problèmes et stockons les résultats dans une matrice.

L'approche ascendante est l'approche utilisée pour éviter la récursivité, économisant ainsi l'espace mémoire. L'algorithme ascendant est un algorithme qui commence par le début, tandis que l'algorithme récursif commence par la fin et fonctionne en arrière. Dans l'approche ascendante, nous partons du cas de base pour trouver la réponse à la fin. Comme nous le savons, les cas de base de la série de Fibonacci sont 0 et 1. Puisque l'approche ascendante commence à partir des cas de base, nous allons donc commencer à partir de 0 et 1.

4. Points clés

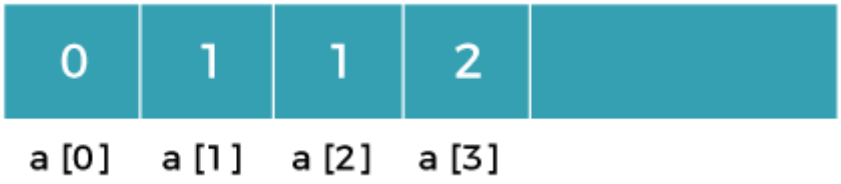
- Nous résolvons tous les sous-problèmes plus petits qui seront nécessaires pour résoudre les sous-problèmes plus importants, puis passons aux problèmes plus importants en utilisant des sous-problèmes plus petits.
- Nous utilisons la boucle *pour* pour parcourir les sous-problèmes.
- L'approche ascendante est également connue sous le nom de méthode de tabulation.

5. Comprenons à travers un exemple

Supposons que nous ayons un tableau qui a des valeurs 0 et 1 aux positions $a[0]$ et $a[1]$, respectivement illustré ci-dessous :

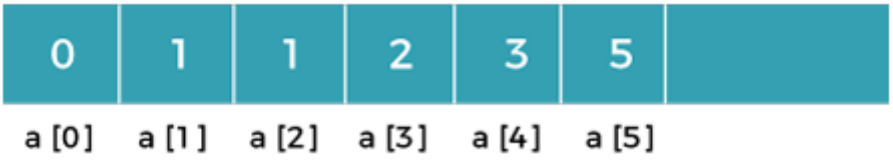
Étant donné que l'approche ascendante commence à partir des valeurs inférieures, les valeurs de $a[0]$ et $a[1]$ sont ajoutées pour trouver la valeur de $a[2]$ indiquée ci-dessous :

La valeur de $a[3]$ sera calculée en ajoutant $a[1]$ et $a[2]$, et elle devient 2 comme ci-dessous :



La valeur de $a[4]$ sera calculée en ajoutant $a[2]$ et $a[3]$, et elle devient 3 comme ci-dessous :

La valeur de $a[5]$ sera calculée en additionnant les valeurs de $a[4]$ et $a[3]$, et elle devient 5 comme ci-dessous :



II. Eléments de programmation dynamique

Bien que nous ayons juste travaillé sur un exemple de méthode de programmation dynamique, vous vous demandez peut-être encore à quel moment la méthode s'applique. Du point de vue de l'ingénierie, quand devons-nous chercher une solution de programmation dynamique à un problème ? Dans cette section, nous examinons les deux ingrédients clés qu'un problème d'optimisation doit avoir pour que la programmation dynamique s'applique : la sous-structure optimale et les sous-problèmes qui se chevauchent. Nous revisitons également et discutons plus en détail de la manière dont la mémorisation pourrait nous aider à tirer parti de la propriété des sous-problèmes qui se chevauchement dans une approche récursive descendante.

1. Sous-structure optimale

La première étape de la résolution d'un problème d'optimisation par programmation dynamique consiste à caractériser la structure d'une solution optimale. Rappelons-vous qu'un problème présente une sous-structure optimale si une solution optimale au problème contient en son sein des solutions optimales aux sous-problèmes. Chaque fois qu'un problème présente une sous-structure optimale, nous avons une bonne idée que la programmation dynamique peut s'appliquer. Dans la programmation dynamique, nous construisons une solution optimale au problème à partir des solutions optimales aux sous-problèmes. Par conséquent, nous devons veiller à ce que la gamme de sous-problèmes que nous considérons inclut ceux utilisés dans une solution optimale. Nous avons découvert une sous-structure optimale dans le problème (suite de Fibonacci) que nous avons examiné dans ce chapitre jusqu'à présent. Nous avons observé que la manière optimale de calculer $Fibonacci(n)$ consiste à calculer de manière optimale $Fibonacci(n-1)$ et $Fibonacci(n-2)$. Vous vous retrouverez à suivre un modèle commun pour découvrir la sous-structure optimale :

1. Vous montrez qu'une solution au problème consiste à faire un choix. Faire ce choix laisse un ou plusieurs sous-problèmes à résoudre.
2. Vous supposez que pour un problème donné, vous avez le choix qui mène à une solution optimale. Vous ne vous préoccupez pas encore de savoir comment déterminer ce choix. Vous supposez simplement qu'il vous a été donné.

3. Compte tenu de ce choix, vous déterminez quels sous-problèmes s'ensuivent et comment caractériser au mieux l'espace de sous-problèmes résultant.
4. Vous montrez que les solutions aux sous-problèmes utilisées dans une solution optimale au problème doivent elles-mêmes être optimales en utilisant une technique de « couper-coller ». Vous le faites en supposant que chacune des solutions du sous-problème n'est pas optimale, puis en dérivant une contradiction. En particulier, en "découpant" la solution non optimale de chaque sous-problème et en "collant" la solution optimale, vous montrez que vous pouvez obtenir une meilleure solution au problème d'origine, contredisant ainsi votre supposition que vous aviez déjà une solution optimale. Si une solution optimale donne lieu à plus d'un sous-problème, ils sont généralement si similaires que vous pouvez modifier l'argument couper-coller pour que l'un s'applique aux autres avec peu d'effort.

Pour caractériser l'espace des sous-problèmes, une bonne règle empirique dit d'essayer de garder l'espace aussi simple que possible, puis de l'étendre si nécessaire.

La sous-structure optimale varie selon les domaines de problèmes de deux manières :

1. Combien de sous-problèmes une solution optimale au problème d'origine utilise, et
2. Combien de choix nous avons pour déterminer quel(s) sous-problème(s) utiliser dans une solution optimale.

Dans le problème de la suite de Fibonacci, une solution optimale pour calculer $Fibonacci(n)$ utilise deux sous-problèmes (de taille $n-1$ et $n-2$ $respectivement$).

De manière informelle, le temps d'exécution d'un algorithme de programmation dynamique dépend du produit de deux facteurs : le nombre de sous-problèmes dans son ensemble et le nombre de choix que nous examinons pour chaque sous-problème.

Habituellement, le graphe de sous-problèmes donne une autre façon d'effectuer la même analyse. Chaque sommet correspond à un sous-problème, et les choix pour un sous-problème sont les arêtes incidentes à ce sous-problème.

La programmation dynamique utilise souvent une sous-structure optimale de manière ascendante. C'est-à-dire que nous trouvons d'abord des solutions optimales aux sous-problèmes et, après avoir résolu les sous-problèmes, nous trouvons une solution optimale au problème. Trouver une solution optimale au problème implique de faire un choix parmi les sous-problèmes que nous utiliserons pour résoudre le problème. Le coût de la solution du problème est généralement le coût du sous-problème plus un coût directement attribuable au choix lui-même.

En partie II, nous examinerons les « algorithmes gloutons », qui présentent de nombreuses similitudes avec la programmation dynamique. En particulier, les problèmes auxquels s'appliquent les algorithmes gloutons ont une sous-structure optimale. Une différence majeure entre les algorithmes gloutons et la programmation dynamique est qu'au lieu de trouver d'abord des solutions optimales aux sous-problèmes, puis de faire un choix éclairé, les algorithmes gloutons font d'abord un choix "glouton" - le choix qui semble le mieux à l'époque - puis résolvent le sous-problème résultant, sans se soucier de résoudre tous les petits sous-problèmes liés possibles. Étonnamment, dans certains cas, cette stratégie fonctionne !

2. Subtilités

Vous devez faire attention à ne pas supposer que la sous-structure optimale s'applique alors que ce n'est pas le cas. Considérons les deux problèmes suivants dans lesquels on nous donne un graphe orienté $G = (V, E)$ et des sommets $u, v \in V$.

Chemin le plus court non pondéré : Trouvez un chemin de u à v composé du moins d'arêtes. Un tel chemin doit être simple, car la suppression d'un cycle d'un chemin produit un chemin avec moins d'arêtes.

Chemin simple le plus long non pondéré : Trouvez un chemin simple de u à v composé du plus grand nombre d'arêtes. Nous devons inclure l'exigence de simplicité car sinon nous pouvons parcourir un cycle autant de fois que nous le souhaitons pour créer des chemins avec un nombre arbitrairement grand d'arêtes.

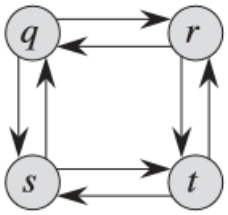


Figure 2 Un graphe orienté montrant que le problème de trouver un chemin simple le plus long dans un graphe orienté non pondéré n'a pas de sous-structure optimale. Le chemin $q \to r \to t$ est un chemin simple le plus long de q à t, mais le sous-chemin $q \to r$ n'est pas un chemin simple le plus long de q à r, pas plus que le sous-chemin $r \to t$ un chemin simple le plus long de r à t.

Le problème du plus court chemin non pondéré présente une sous-structure optimale, comme suit. Supposons que $u \neq v$, de sorte que le problème est non trivial. Ensuite, tout chemin p de u à v doit contenir un sommet intermédiaire, disons w. (Notez que w peut être u ou v.) Ainsi, nous pouvons décomposer le chemin $u \xrightarrow{p} v$ en sous-chemins $u \xrightarrow{p_1} w \xrightarrow{p_2} v$. Clairement, le nombre d'arêtes dans p est égal au nombre d'arêtes dans p_1 plus le nombre d'arêtes dans p_2. Nous affirmons que si p est un chemin optimal (c'est-à-dire le plus court) de u à v, alors p_1 doit être un chemin le plus court de u à w. Pourquoi ? Nous utilisons un argument "couper-coller" : s'il y avait un autre chemin, disons p_1', de u à w avec moins d'arêtes que p_1, alors nous pourrions couper p_1 et coller dans p_1' pour produire un chemin $u \xrightarrow{p_1'} w \xrightarrow{p_2} v$ avec moins d'arêtes que p, contredisant ainsi l'optimalité de p. Symétriquement, p_2 doit être le chemin le plus court de w à v. Ainsi, nous pouvons trouver un chemin le plus court de u à v en considérant tous les sommets intermédiaires w, en trouvant un chemin le plus court de u à w et un chemin le plus court de w à v, et en choisissant un sommet intermédiaire w qui donne le chemin global le plus court. Dans la section 25.2, nous utilisons une variante de cette observation de sous-structure optimale pour trouver le chemin le plus court entre chaque paire de sommets sur un graphe orienté pondéré. Vous pourriez être tenté de supposer que le problème de trouver un chemin simple le plus long non pondéré présente également une sous-structure optimale. Après tout, si nous décomposons un chemin simple le plus long en sous-chemins $u \xrightarrow{p_1} w \xrightarrow{p_2} v$, alors p_1 ne doit-il pas être un chemin simple le plus long de u à w, et p_2 ne doit-il pas être un chemin simple le plus long de w à v ? La réponse est non ! La figure 15.6 fournit un exemple. Considérez le chemin $q \longrightarrow r \longrightarrow t$, qui est le chemin simple le

plus long de q à t. Est-ce que $q \longrightarrow r$ un chemin simple le plus long de q à r ? Non, le chemin $q \longrightarrow r \longrightarrow s \longrightarrow t \longrightarrow r$ est un chemin simple qui est plus long. Est-ce que $r \longrightarrow t$ un chemin simple le plus long de r à t ? Non encore, le chemin $r \longrightarrow q \longrightarrow s \longrightarrow t$ est un chemin simple qui est plus long.

Cet exemple montre que pour les chemins simples les plus longs, non seulement le problème manque de sous-structure optimale, mais on ne peut pas nécessairement assembler une solution "légale" au problème à partir de solutions aux sous-problèmes. Si nous combinons les chemins simples les plus longs $q \longrightarrow s \longrightarrow t \longrightarrow r$ et $r \longrightarrow q \longrightarrow s \longrightarrow t$, on obtient le chemin $q \longrightarrow s \longrightarrow t \longrightarrow r \longrightarrow q \longrightarrow s \longrightarrow t$, ce qui n'est pas simple. En effet, le problème de trouver un chemin simple le plus long non pondéré ne semble pas avoir de sous-structure optimale. Aucun algorithme de programmation dynamique efficace pour ce problème n'a jamais été trouvé. En fait, ce problème est NP-complet, ce qui, signifie qu'il est peu probable que nous trouvions un moyen de le résoudre en temps polynomial. Pourquoi la sous-structure du plus long chemin simple est-elle si différente de celle du plus court chemin ? Bien qu'une solution à un problème pour les chemins les plus longs et les plus courts utilise deux sous-problèmes, les sous-problèmes pour trouver le chemin simple le plus long ne sont pas indépendants, alors qu'ils le sont pour les chemins les plus courts. Qu'entendons-nous par sous-problèmes indépendants ? Nous voulons dire que la solution d'un sous-problème n'affecte pas la solution d'un autre sous-problème du même problème. Pour l'exemple de la figure 2, nous avons le problème de trouver le chemin simple le plus long de q à t avec deux sous-problèmes : trouver les chemins simples les plus longs de q à r et de r à t. Pour le premier de ces sous-problèmes, nous choisissons le chemin $q \longrightarrow s \longrightarrow t \longrightarrow r$, et nous avons donc également utilisé les sommets s et t. On ne peut plus utiliser ces sommets dans le deuxième sous-problème, puisque la combinaison des deux solutions aux sous-problèmes donnerait un chemin qui n'est pas simple. Si nous ne pouvons pas utiliser le sommet t dans le deuxième problème, alors nous ne pouvons pas le résoudre du tout, car t doit être sur le chemin que nous trouvons, et ce n'est pas le sommet auquel nous « collons » ensemble les solutions du sous-problème (ce sommet étant r). Parce que nous utilisons les sommets s et t dans une solution de sous-problème, nous ne pouvons pas les utiliser dans l'autre solution de sous-problème. Cependant, nous devons utiliser au moins l'un d'eux pour résoudre l'autre sous-problème, et nous devons utiliser les deux pour le résoudre de manière optimale. Ainsi, on dit que ces sous-problèmes ne sont pas indépendants. Vu d'une autre manière, l'utilisation de ressources pour résoudre un sous-problème (ces ressources étant des sommets) les rend indisponibles pour l'autre sous-problème.

Pourquoi, alors, les sous-problèmes sont-ils indépendants pour trouver le chemin le plus court ? La réponse est que par nature, les sous-problèmes ne partagent pas de ressources. Nous prétendons que si un sommet w se trouve sur le chemin le plus court p de u à v, alors nous pouvons assembler n'importe quel chemin le plus court $u \overset{p_1}{\rightarrow} w$ et n'importe quel chemin le plus court $w \overset{p_2}{\rightarrow} v$ pour produire un chemin le plus court de u à v. On est assuré qu'en dehors de w, aucun sommet ne peut apparaître dans les deux chemins p_1 et p_2. Pourquoi ? Supposons qu'un sommet $x \neq w$ apparaisse à la fois dans p_1 et p_2, de sorte que nous puissions décomposer p_1 en $u \overset{p_{ux}}{\longrightarrow} x \rightarrow w$ et p_2 en $w \longrightarrow x \overset{p_{xv}}{\longrightarrow} v$. Par la sous-structure optimale de ce problème, le chemin p a autant d'arêtes que p_1 et p_2 ensemble ; disons que p a e arêtes. Construisons maintenant un chemin $p' = u \overset{p_{ux}}{\longrightarrow} x \overset{p_{xv}}{\longrightarrow} v$ de u à v.

Parce que nous avons excisé les chemins de x à w et de w à x, dont chacun contient au moins une arête, le chemin p' contient au plus $e - 2$ arêtes, ce qui contredit l'hypothèse selon laquelle p est le plus court chemin. Ainsi, nous sommes assurés que les sous-problèmes du problème du plus court chemin sont indépendants.

3. Chevauchement de sous-problèmes

Le deuxième ingrédient qu'un problème d'optimisation doit avoir pour que la programmation dynamique s'applique est que l'espace des sous-problèmes doit être « petit » dans le sens où un algorithme récursif pour le problème résout les mêmes sous-problèmes encore et encore, plutôt que de toujours générer de nouveaux sous-problèmes. Typiquement, le nombre total de sous-problèmes distincts est un polynôme dans la taille d'entrée. Lorsqu'un algorithme récursif revisite le même problème à plusieurs reprises, nous disons que le problème d'optimisation a des sous-problèmes qui se chevauchent. Les algorithmes de programmation dynamique tirent généralement parti des sous-problèmes qui se chevauchent en résolvant chaque sous-problème une fois, puis en stockant la solution dans une table où elle peut être consultée en cas de besoin, en utilisant un temps constant par recherche. Notre solution de programmation dynamique réduit un algorithme récursif à temps exponentiel au temps quadratique.

4. Reconstruire une solution optimale

En pratique, nous stockons souvent le choix que nous avons fait dans chaque sous-problème dans une table afin que nous n'ayons pas à reconstruire cette information à partir des coûts que nous avons stockés.

Mémorisation

Comme nous l'avons vu pour le problème de coupe de tiges, il existe une approche alternative à la programmation dynamique qui offre souvent l'efficacité de l'approche de programmation dynamique ascendante tout en maintenant une stratégie descendante. L'idée est de mémoriser l'algorithme récursif naturel, mais inefficace. Comme dans l'approche ascendante, nous maintenons une table avec des solutions de sous-problèmes, mais la structure de contrôle pour remplir la table ressemble plus à l'algorithme récursif. Un algorithme récursif mémorisé maintient une entrée dans un tableau pour la solution de chaque sous-problème. Chaque entrée de table contient initialement une valeur spéciale pour indiquer que l'entrée n'a pas encore été remplie. Lorsque le sous-problème est rencontré pour la première fois au fur et à mesure que l'algorithme récursif se déroule, sa solution est calculée puis stockée dans la table. Chaque fois que nous rencontrons ce sous-problème, nous recherchons simplement la valeur stockée dans la table et la renvoyons.

En pratique générale, si tous les sous-problèmes doivent être résolus au moins une fois, un algorithme de programmation dynamique ascendant surpasse généralement l'algorithme mémorisé descendant correspondant d'un facteur constant, car l'algorithme ascendant n'a pas de surcharge pour la récursivité et moins de surcharge pour maintenir la table. De plus, pour certains problèmes, nous pouvons exploiter le modèle régulier d'accès aux tables dans l'algorithme de programmation dynamique pour réduire encore plus les besoins en temps ou en espace. Alternativement, si certains sous-problèmes dans

l'espace des sous-problèmes n'ont pas du tout besoin d'être résolus, la solution mémorisée a l'avantage de résoudre uniquement les sous-problèmes qui sont vraiment nécessaires.

III. Quelques problèmes classiques

1. MULTIPLICATION D'UNE CHAINE DE MATRICES

Notre premier exemple de programmation dynamique est un algorithme qui résout le problème de la multiplication d'une chaine de matrices. On nous donne une séquence (chaîne) $\langle A_1, A_2, \dots, A_n \rangle$ de n matrices à multiplier, et on souhaite calculer le produit

$$A_1 \times A_2 \times \dots \times A_n \tag{1.1}$$

On peut évaluer l'expression (1.1) à l'aide de l'algorithme standard pour multiplier des paires de matrices en tant que sous-routine une fois que nous l'avons mis entre parenthèses pour résoudre toutes les ambiguïtés dans la façon dont les matrices sont multipliées ensemble. La multiplication matricielle est associative, et donc toutes les parenthèses donnent le même produit. Un produit de matrices est entièrement parenthésés s'il s'agit d'une seule matrice ou du produit de deux produits matriciels entièrement parenthésés, entourés de parenthèses. Par exemple, si la chaîne de matrices est (A_1, A_2, A_3, A_4), alors nous pouvons entièrement parenthésés le produit (A_1, A_2, A_3, A_4) de cinq manières distinctes :

$(A_1\ (A_2\ (A_3\ A_4)))$

$(A_1\ ((A_2\ A_3)\ A_4))$

$(((A_1\ A_2)A_3)\ A_4)$

$((A_1\ (A_2\ A_3))\ A_4)$

$((A_1\ A_2\ )(A_3\ A_4))$

La façon dont nous parenthésons une chaîne de matrices peut avoir un impact considérable sur le coût d'évaluation du produit. Considérons d'abord le coût de la multiplication de deux matrices. L'algorithme standard est donné par le pseudocode suivant, qui généralise la procédure SQUARE-MATRIX-MULTIPLY. Les attributs *rows* et *columns* sont les nombres de lignes et de colonnes dans une matrice.

```
MATRIX-MULTIPLY(A, B)
1  if A.columns ≠ B.rows
2       error "incompatible dimensions"
3  else let C be a new A.rows × B.columns matrix
4       for i = 1 to A.rows
5            for j = 1 to B.columns
6                 c_ij = 0
7                 for k = 1 to A.columns
8                      c_ij = c_ij + a_ik · b_kj
9       return C
```

On ne peut multiplier deux matrices A et B que si elles sont compatibles : le nombre de colonnes de A doit être égal au nombre de lignes de B. Si A est une matrice $p \times q$ et B est une matrice $q \times r$, la matrice résultante C est une matrice $p \times r$. Le temps de calcul de C est dominé par le nombre de multiplications scalaires à la ligne 8, qui est $p \times q \times r$. Dans ce qui suit, nous exprimerons les coûts en termes de nombre de multiplications scalaires. Pour illustrer les différents coûts engendrés par les différents parenthésages du produit matriciel, considérons le problème d'une chaîne (A_1, A_2, A_3) de trois matrices. Supposons que les dimensions des matrices soient respectivement de 10×100, 100×5 et 5×50. Si nous multiplions selon le parenthésage $((A_1A_2)A_3)$, nous effectuons $10 \times 100 \times 5 = 5000$ multiplications scalaires pour calculer le produit matriciel A_1A_2 de dimension 10×5, plus encore $10 \times 5 \times 50 = 2500$ multiplications scalaires pour multiplier cette matrice par A_3, pour un total de 7500 multiplications scalaires. Si à la place nous multiplions selon le parenthésage $(A_1(A_2A_3))$, nous effectuons $100 \times 5 \times 50 = 25000$ multiplications scalaires pour calculer le produit matriciel A_2A_3 de dimension 100×50, plus $10 \times 100 \times 50 = 50000$ multiplications scalaires supplémentaires pour multiplier A_1 par cette matrice, pour un total de 75000 multiplications scalaires. Ainsi, le calcul du produit selon le premier parenthésage est 10 fois plus rapide. Nous énonçons le problème de **multiplication d'une chaine de matrices** comme suit : étant donné une chaîne $(A_1, A_2, \dots, A_n)$ de n matrices, où pour $i = 1, 2, \dots, n$, la matrice A_i a pour dimension $p_{i-1} \times p_i$, parenthésez entièrement le produit $A_1A_2 \dots A_n$ d'une manière qui minimise le nombre de multiplications scalaires. Notez que dans le problème de multiplication matrice-chaîne, nous ne multiplions pas réellement les matrices. Notre objectif est uniquement de déterminer un ordre de multiplication des matrices qui a le coût le plus bas. Typiquement, le temps investi dans la détermination de cet ordre optimal est plus que payé par le temps gagné plus tard lors de l'exécution effective des multiplications matricielles (comme effectuer seulement 7500 multiplications scalaires au lieu de 75000).

COMPTER LE NOMBRE DE PARENTHÉSAGES

Avant de résoudre le problème de multiplication d'une chaine de matrices par programmation dynamique, convainquons-nous qu'une vérification exhaustive de toutes les parenthèses possibles ne donne pas un algorithme efficace. Notons le nombre de parenthèses alternatives d'une séquence de n matrices par $\mathcal{P}(n)$. Lorsque $n = 1$, nous n'avons qu'une seule matrice et donc une seule façon de parenthéser entièrement le produit matriciel. Lorsque $n = 2$, un produit matriciel entièrement parenthésé est le produit de deux sous-produits matriciels entièrement parenthésés, et la division entre les deux sous-produits peut se produire entre les matrices numéro k et $(k + 1)$ pour tout $k = 1, 2, \dots, n - 1$. Ainsi, on obtient la récurrence

$$\mathcal{P}(n) = \begin{cases} 1 \; si \; n = 1 \\ \sum_{k=1}^{n-1} \mathcal{P}(k)\mathcal{P}(n-k) \; si \; n \geq 2 \end{cases}$$

Le nombre de solutions est donc exponentiel en n, et la méthode de force brute de recherche exhaustive constitue une mauvaise stratégie pour déterminer comment parenthéser de manière optimale une chaîne matricielle.

2. LA PLUS LONGUE SOUS-SÉQUENCE COMMUNE

Les applications biologiques doivent souvent comparer l'ADN de deux (ou plusieurs) organismes différents, Un brin d'ADN se compose d'une chaîne de molécules appelées bases, où les bases possibles sont **l'adénine, la guanine, la cytosine et la thymine**.

En représentant chacune de ces bases par sa lettre initiale, nous pouvons exprimer un brin d'ADN sous forme de chaîne sur l'ensemble fini $\{A, C, G, T\}$. Par exemple, l'ADN d'un organisme peut être $S_1 = ACCGGTCGAGTGCGCGGAAGCCGGCGCCGAA$, et l'ADN d'un autre organisme peut être $S_2 = GTCGTTCGGAATGCCGTTGCTCTGTAAA$. L'une des raisons de comparer deux brins d'ADN est de déterminer à quel point les deux brins sont « similaires », comme une mesure de l'étroite relation entre les deux organismes. Nous pouvons, et nous le faisons, définir la similarité de différentes manières. Par exemple, on peut dire que deux brins d'ADN sont similaires si l'un est une sous-chaîne de l'autre. Dans notre exemple, ni S_1 ni S_2 ne sont une sous-chaîne de l'autre. Alternativement, nous pourrions dire que deux brins sont similaires si le nombre de changements nécessaires pour transformer l'un en l'autre est petit. Une autre façon de mesurer la similitude des brins S_1 et S_2 consiste à trouver un troisième brin S_3 dans lequel les bases de S_3 apparaissent dans chacun des brins S_1 et S_2 ; ces bases doivent apparaître dans le même ordre, mais pas nécessairement consécutivement. Plus le brin S_3 que nous pouvons trouver est long, plus S_1 et S_2 sont similaires. Dans notre exemple, le brin le plus long S_3 est $GTCGTCGGAAGCCGGCCGAA$. Nous formalisons cette dernière notion de similarité comme le problème de la plus longue sous-chaine commune. Une sous-séquence d'une séquence donnée est juste la séquence donnée avec zéro ou plusieurs éléments omis. Formellement, étant donné une séquence $X = \langle x_1, x_2, \dots, x_m \rangle$, une autre séquence $Z = \langle z_1, z_2, \dots, z_m \rangle$ est une sous-séquence de X s'il existe une séquence strictement croissante $\langle i_1, i_2, \dots, i_k \rangle$ des indices de X tels que pour tout $j = 1, 2, \dots, k$, on a $x_{ij} = z_j$. Par exemple, $Z = \langle B, C, D, B \rangle$ est une sous-séquence de $X = \langle A, B, C, B, D, A, B \rangle$ avec la séquence d'index correspondante $\langle 2, 3, 5, 7 \rangle$. Étant donné deux séquences X et Y, on dit qu'une séquence Z est une sous-séquence commune de X et Y si Z est une sous-séquence à la fois de X et Y. Par exemple, si $X = \langle A, B, C, B, D, A, B \rangle$ et $Y = \langle B, D, C, A, B, A \rangle$, la séquence $\langle B, C, A \rangle$ est une sous-séquence commune de X et Y. La séquence $\langle B, C, A \rangle$ n'est pas la plus longue sous-séquence commune (LSC) de X et Y, cependant, puisqu'il a une longueur 3 et que la séquence $\langle B, C, B, A \rangle$, qui est également commune à X et Y, a une longueur 4. La séquence $\langle B, C, B, A \rangle$ est une (LSC) de X et Y, tout comme la séquence $\langle B, D, A, B \rangle$, puisque X et Y n'ont pas de sous-séquence commune de longueur 5 ou plus. Dans le problème de **la plus longue sous-séquence commune**, on nous donne deux séquences $X = \langle x_1, x_2, \dots, x_m \rangle$ et $Y = \langle y_1, y_2, \dots, y_n \rangle$ et souhaitons trouver une sous-séquence commune de longueur maximale de X et Y.

PARTIE II : ALGORITHMES GLOUTONS

Les algorithmes pour les problèmes d'optimisation passent généralement par une séquence d'étapes, avec un ensemble de choix à chaque étape. Pour de nombreux problèmes d'optimisation, l'utilisation de la programmation dynamique pour déterminer les meilleurs choix est exagérée ; des algorithmes plus simples et plus efficaces feront l'affaire. Un algorithme glouton fait toujours le choix qui semble le mieux pour le moment. C'est-à-dire qu'il fait un choix localement optimal dans l'espoir que ce choix conduira à une solution globalement optimale. Cette partie explore les problèmes d'optimisation pour lesquels les algorithmes gloutonnes fournissent des solutions optimales. Avant de lire cette partie, vous devriez vous renseigner sur la programmation dynamique au chapitre en partie I. Les algorithmes gloutons ne donnent pas toujours des solutions optimales, mais ils le font pour de nombreux problèmes. Nous examinerons d'abord, dans la section I, un problème simple mais non trivial, le problème de sélection d'activités, pour lequel un algorithme glouton calcule efficacement une solution optimale. Nous arriverons à l'algorithme glouton en considérant d'abord une approche de programmation dynamique et en montrant ensuite que l'on peut toujours faire des choix gloutons pour arriver à une solution optimale. La section II passe en revue les éléments de base de la stratégie gloutonne, donnant une approche directe pour prouver que les algorithmes gloutons sont corrects. Enfin la section III présente une application importante des stratégies gloutonnes : le problème de séquencement d'un ensemble de taches avec délais. La méthode gloutonne est assez puissante et fonctionne bien pour un large éventail de problèmes.

I. Un exemple introductif : Problème de sélection d'activité

Notre premier exemple est le problème de l'ordonnancement de plusieurs activités concurrentes qui nécessitent l'utilisation exclusive d'une ressource commune, dans le but de sélectionner un ensemble de taille maximale d'activités compatibles entre elles. Supposons que nous ayons un ensemble $S = \{a_1, a_2, \dots, a_n\}$ de n activités proposées qui souhaitent utiliser une ressource, telle qu'un amphithéâtre, qui ne peut servir qu'une seule activité à la fois. Chaque activité a_i a une heure de début s_i et une heure de fin f_i, où $0 \leq s_i < f_i < \infty$. Si elle est sélectionnée, l'activité a_i a lieu pendant l'intervalle de temps de demi-ouverture $[s_i, f_i[$. Les activités a_i et a_j sont compatibles si les intervalles $[s_i, f_i[$ et $[s_j, f_j[$ ne se chevauchent pas. C'est-à-dire que a_i et a_j sont compatibles si $s_i \geq f_j \; ou \; s_j \geq f_i$. Dans le problème de sélection d'activités, nous souhaitons sélectionner un sous-ensemble de taille maximale d'activités mutuellement compatibles. Nous supposons que les activités sont triées par ordre monotone croissant de temps de fin :

$$f_1 \leq f_2 \leq f_3 \leq \cdots \leq f_{n-1} \leq f_n \qquad (2.1)$$

(Nous verrons plus loin l'avantage que procure cette hypothèse.) Par exemple, considérons l'ensemble S d'activités suivant :

Pour cet exemple, le sous-ensemble $\{a_3, a_9, a_{11}\}$ se compose d'activités mutuellement compatibles. Ce n'est pas un sous-ensemble maximum, cependant, puisque le sous-ensemble $\{a_1, a_4, a_8, a_{11}\}$ est plus grand. En fait, $\{a_1, a_4, a_8, a_{11}\}$ est le plus grand sous-ensemble d'activités mutuellement compatibles ; un autre plus grand sous-ensemble est

$\{a_2, a_4, a_9, a_{11}\}$. Nous allons résoudre ce problème en plusieurs étapes. Nous commençons par réfléchir à une solution de programmation dynamique, dans laquelle nous considérons plusieurs choix pour déterminer quels sous-problèmes utilisent une solution optimale. Nous observerons alors que nous n'avons besoin de considérer qu'un seul choix — le choix glouton — et que lorsque nous faisons le choix glouton, il ne reste qu'un seul sous-problème. Sur la base de ces observations, nous développerons un algorithme glouton récursif pour résoudre le problème d'ordonnancement d'activités. Nous terminerons le processus de développement d'une solution gloutonne en convertissant l'algorithme récursif en un algorithme itératif. Bien que les étapes que nous allons suivre dans cette section soient légèrement plus compliquées que ce qui est typique lors du développement d'un algorithme glouton, elles illustrent la relation entre les algorithmes gloutonnes et la programmation dynamique.

1. La sous-structure optimale du problème de sélection d'activité

Nous pouvons facilement vérifier que le problème de sélection d'activité présente une sous-structure optimale. Notons S_{ij} l'ensemble des activités qui commencent après la fin de l'activité a_i et qui se terminent avant le début de l'activité a_j. Supposons que nous souhaitions trouver un ensemble maximum d'activités mutuellement compatibles dans S_{ij}, et supposons en outre qu'un tel ensemble maximum soit A_{ij}, qui inclut une certaine activité a_k. En incluant a_k dans une solution optimale, nous nous retrouvons avec deux sous-problèmes : trouver des activités mutuellement compatibles dans l'ensemble S_{ik} (activités qui commencent après la fin de l'activité a_i et qui se terminent avant le début de l'activité a_k) et trouver des activités mutuellement compatibles dans l'ensemble S_{kj} (activités qui commencent après la fin de l'activité a_k et qui se termine avant le début de l'activité a_j). Soit $A_{ik} = A_{ij} \cap S_{ik}$ et $A_{kj} = A_{ij} \cap S_{kj}$, de sorte que A_{ik} contienne les activités dans A_{ij} qui se terminent avant que a_k ne commence et A_{kj} contienne les activités dans A_{ij} qui commencent après la fin de a_k. Ainsi, nous avons $A_{ij} = A_{ik} \cup \{a_k\} \cup A_{kj}$, et donc la taille maximale A_{ij} de l'ensemble d'activités mutuellement compatibles dans S_{ij} se compose de $|A_{ij}| = |A_{ik}| + |A_{kj}| + 1$ activités. L'argument habituel du copier-coller montre que la solution optimale A_{ij} doit également inclure des solutions optimales aux deux sous-problèmes pour S_{ik} et S_{kj}. Si nous pouvions trouver un ensemble A'_{kj} d'activités mutuellement compatibles dans S_{kj} où $|A'_{kj}| > |A_{kj}|$, alors nous pourrions utiliser A'_{kj}, plutôt que A_{kj}, dans une solution au sous-problème pour S_{ij}. Nous aurions construit un ensemble de $|A_{ik}| + |A'_{kj}| + 1 > |A_{ik}| + |A_{kj}| + 1 = |A_{ij}|$ activités compatibles entre elles, ce qui contredit l'hypothèse selon laquelle A_{ij} est une solution optimale. Un argument symétrique s'applique aux activités dans S_{ik}. Cette façon de caractériser la sous-structure optimale suggère que nous pourrions résoudre le problème de sélection d'activité par programmation dynamique. Si nous notons la taille d'une solution optimale pour l'ensemble S_{ij} par $c[i, j]$, alors nous aurions la récurrence

$$c[i, j] = c[i, k] + c[k, j] + 1.$$

Bien sûr, si nous ne savions pas qu'une solution optimale pour l'ensemble S_{ij} inclut l'activité a_k, nous devrions examiner toutes les activités dans S_{ij} pour trouver laquelle choisir, de sorte que

$$c[i,j] = \begin{cases} 0 \; si \; S_{ij} = \emptyset \\ \max\limits_{a_k \in S_{ij}} \{c[i,k] + c[k,j] + 1\} \; si \; S_{ij} \neq \emptyset \end{cases} \quad (2.2)$$

Nous pourrions alors développer un algorithme récursif et le mémoriser, ou nous pourrions travailler de bas en haut et remplir les entrées du tableau au fur et à mesure. Mais nous négligerions une autre caractéristique importante du problème de la sélection d'activités que nous pouvons utiliser avec grand avantage.

2. Faire le choix glouton

Et si nous pouvions choisir une activité à ajouter à notre solution optimale sans avoir à résoudre d'abord tous les sous-problèmes ? Cela pourrait nous éviter d'avoir à considérer tous les choix inhérents à la récurrence (2.2). En fait, pour le problème de sélection d'activité, nous n'avons qu'à considérer un seul choix : le choix glouton. Qu'entendons-nous par le choix glouton pour le problème de sélection d'activités ? L'intuition suggère que nous devrions choisir une activité qui laisse la ressource disponible pour autant d'autres activités que possible. Or, parmi les activités que nous finissons par choisir, l'une d'entre elles doit être la première à terminer. Notre intuition nous dit donc de choisir l'activité en S avec l'heure de fin la plus rapprochée, car cela laisserait la ressource disponible pour autant d'activités qui la suivent que possible. (Si plus d'une activité dans S a l'heure de fin la plus proche, alors nous pouvons choisir n'importe laquelle de ces activités.) En d'autres termes, puisque les activités sont triées dans un ordre monotone croissant par heure de fin, le choix le plus glouton est l'activité a_1. Choisir la première activité à terminer n'est pas la seule façon de penser à faire un choix glouton pour ce problème. Si nous faisons le choix glouton, nous n'avons plus qu'un seul sous-problème à résoudre : trouver des activités qui commencent après la fin de a_1. Pourquoi ne devons-nous pas considérer les activités qui se terminent avant que a_1 ne commence ? Nous avons que $s_1 < f_1$, et f_1 est l'heure de fin la plus proche de toute activité, et donc aucune activité ne peut avoir une heure de fin inférieure ou égale à s_1. Ainsi, toutes les activités compatibles avec l'activité a_1 doivent commencer après la fin de a_1.

De plus, nous avons déjà établi que le problème de sélection d'activité présente une sous-structure optimale. Soit $S_k = \{a_i \in S : s_i \geq f_k\}$ l'ensemble qui commencent après la fin de l'activité a_k. Si nous faisons le choix glouton de l'activité a_1, alors S_1 reste le seul sous-problème à résoudre. La sous-structure optimale nous dit que si a_1 est dans la solution optimale, alors une solution optimale au problème d'origine consiste en l'activité a_1 et toutes les activités dans une solution optimale au sous-problème S_1. Une grande question demeure : notre intuition est-elle correcte ? Le choix glouton – dans lequel nous choisissons la première activité à terminer – fait-il toujours partie d'une solution optimale ? Le théorème suivant montre que c'est le cas.

Théorème

Considérons tout sous-problème non vide S_k, et soit a_m une activité dans S_k avec l'instant de fin le plus proche. Ensuite, a_m est inclus dans un sous-ensemble de taille maximale d'activités mutuellement compatibles de S_k.

Preuve

Soit A_k un sous-ensemble de taille maximale d'activités mutuellement compatibles dans S_k, et soit a_j l'activité de A_k avec le temps de fin le plus tôt. Si $a_j = a_m$, nous avons terminé, puisque nous avons montré que a_m appartient à un sous-ensemble de taille maximale d'activités mutuellement compatibles de S_k. Si $a_j \neq a_m$, soit l'ensemble $A'_k = A_k - \{a_j\} \cup \{a_m\}$ soit A_k mais en remplaçant a_m par a_j. Les activités de A'_k sont disjointes, ce qui suit parce que les activités de A_k sont disjointes, a_j est la première activité de A_k à terminer, et $f_m \leq f_j$. Puisque $|A'_k| = |A_k|$, nous concluons que A'_k est un sous-ensemble de taille maximale d'activités mutuellement compatibles de S_k, et il inclut a_m.

Ainsi, nous voyons que bien que nous puissions résoudre le problème de sélection d'activité avec la programmation dynamique, nous n'en avons pas besoin. (En outre, nous n'avons pas encore examiné si le problème de sélection d'activité a même des sous-problèmes qui se chevauchent.) Au lieu de cela, nous pouvons choisir à plusieurs reprises l'activité qui se termine en premier, ne garder que les activités compatibles avec cette activité et répéter jusqu'à ce qu'il ne reste plus aucune activité. De plus, comme nous choisissons toujours l'activité avec l'heure de fin la plus proche, les heures de fin des activités que nous choisissons doivent strictement augmenter. Nous pouvons considérer chaque activité une seule fois dans l'ensemble, dans l'ordre monotone croissant des temps de fin. Un algorithme pour résoudre le problème de sélection d'activité n'a pas besoin de travailler de bas en haut, comme un algorithme de programmation dynamique basé sur une table. Au lieu de cela, il peut fonctionner de haut en bas, en choisissant une activité à intégrer dans la solution optimale, puis en résolvant le sous-problème consistant à choisir des activités parmi celles qui sont compatibles avec celles déjà choisies. Les algorithmes gloutons ont généralement cette conception descendante : faire un choix puis résoudre un sous-problème, plutôt que la technique ascendante consistant à résoudre des sous-problèmes avant de faire un choix.

3. Un algorithme glouton récursif

Maintenant que nous avons vu comment contourner l'approche de programmation dynamique et utiliser à la place un algorithme glouton descendant, nous pouvons écrire une procédure simple et récursive pour résoudre le problème de sélection d'activité. La procédure RECURSIVEACTIVITY-SELECTOR prend les heures de début et de fin des activités, représentées par les tableaux s et f, l'index k qui définit le sous-problème S_k qu'il doit résoudre, et la taille n du problème d'origine. Il renvoie un ensemble de taille maximale d'activités mutuellement compatibles dans S_k. Nous supposons que les n activités d'entrée sont déjà ordonnées par un temps de fin croissant de façon monotone, conformément à l'inéquation (2.1). Sinon, nous pouvons les trier dans cet ordre en temps $O(n \log n)$, rompant les liens arbitrairement. Pour commencer, nous ajoutons l'activité fictive a_0 avec $f_0 = 0$, de sorte que le sous-problème S_0 est l'ensemble complet des activités S. L'appel initial, qui résout tout le problème, est RECURSIVE-ACTIVITY-SELECTOR$(s, f, 0, n)$.

```
RECURSIVE-ACTIVITY-SELECTOR(s, f, k, n)
1  m = k + 1
2  while m ≤ n and s[m] < f[k]        // find the first activity in S_k to finish
3      m = m + 1
4  if m ≤ n
5      return {a_m} ∪ RECURSIVE-ACTIVITY-SELECTOR(s, f, m, n)
6  else return ∅
```

Dans un appel récursif donné RECURSIVE-ACTIVITY-SELECTOR(s, f, k, n), la boucle while des lignes 2–3 recherche la première activité dans S_k à terminer. La boucle examine $a_{k+1}, a_{k+2}, \ldots, a_n$, jusqu'à ce qu'elle trouve la première activité a_m compatible avec a_k; une telle activité a $s_m \geq f_k$. Si la boucle se termine parce qu'elle trouve une telle activité, la ligne 5 renvoie l'union de la famille et le sous-ensemble de taille maximale de S_m renvoyé par l'appel récursif RECURSIVE-ACTIVITY-SELECTOR(s, f, m, n). Alternativement, la boucle peut se terminer parce que $m > n$, auquel cas nous avons abordé toutes les activités dans S_k sans en trouver une compatible avec a_k. Dans ce cas, $S_k = \emptyset$, et donc la procédure retourne $\emptyset$ en ligne 6. En supposant que les activités aient déjà été triées par des heures de fin, le temps d'exécution de l'appel RECURSIVE-ACTIVITY-SELECTOR$(s, f, 0, n)$ est $\Theta(n)$, ce que l'on peut voir comme suit. Sur tous les appels récursifs, chaque activité est examinée exactement une fois dans le test de la boucle *tant que* de la ligne 2. En particulier, l'activité est examinée dans le dernier appel effectué dans lequel $k < i$.

4. Un algorithme glouton itératif

Nous pouvons facilement convertir notre procédure récursive en une procédure itérative. La procédure RECURSIVE-ACTIVITY-SELECTOR est presque « tail recursive » : elle se termine par un appel récursif à lui-même suivi d'une opération d'union. C'est généralement une tâche simple que de transformer une procédure tail recursive en une forme itérative ; en fait, certains compilateurs pour certains langages de programmation effectuent cette tâche automatiquement. Tel qu'il est écrit, RECURSIVE-ACTIVITY-SELECTOR fonctionne pour les sous-problèmes S_k, c'est-à-dire les sous-problèmes qui consistent en les dernières activités à terminer.

La procédure GREEDY-ACTIVITY-SELECTOR est une version itérative de la procédure RECURSIVE-ACTIVITY-SELECTOR. Il suppose également que les activités d'entrée sont ordonnées par un temps de fin croissant de façon monotone. Il rassemble les activités sélectionnées dans un ensemble A et renvoie cet ensemble lorsqu'il est terminé.

```
GREEDY-ACTIVITY-SELECTOR(s, f)
1  n = s.length
2  A = {a_1}
3  k = 1
4  for m = 2 to n
5      if s[m] ≥ f[k]
6          A = A ∪ {a_m}
7          k = m
8  return A
```

La procédure fonctionne comme suit. La variable k indexe la plus récente addition à A, correspondant à l'activité a_k dans la version récursive. Puisque nous considérons les activités dans l'ordre de temps de fin croissant de façon monotone, f_k est toujours le temps de fin maximum de toute activité dans A. C'est-à-dire,

$$f_k = \max \{f_i : a_i \in A\} \tag{2.3}$$

Les lignes 2 à 3 sélectionnent l'activité a_1, initialisent A pour contenir uniquement cette activité et initialisent k pour indexer cette activité. La boucle for des lignes 4 à 7 trouve la première activité dans S_k à terminer. La boucle considère chaque activité a_m tour à tour et ajoute a_m à A si elle est compatible avec toutes les activités précédemment sélectionnées ; une telle activité est la première à se terminer en S_k. Pour voir si l'activité a_m est compatible avec toutes les activités actuellement en A, il suffit par l'équation (2.3) de vérifier (à la ligne 5) que son heure de début s_m n'est pas antérieure à l'heure de fin f_k de l'activité la plus récemment ajoutée à A. Si l'activité a_m est compatible, les lignes 6 à 7 ajoutent l'activité a_m à A et définissent k à m. L'ensemble A renvoyé par l'appel GREEDY-ACTIVITY-SELECTOR(s, f) est précisément l'ensemble renvoyé par l'appel RECURSIVE-ACTIVITY-SELECTOR$(s, f, 0, n)$. Comme la version récursive, GREEDY-ACTIVITY-SELECTOR planifie un ensemble de n activités en $\Theta(n)$ temps, en supposant que les activités ont déjà été triées initialement par leurs heures de fin.

II. Éléments de la stratégie gloutonne

L'algorithme glouton obtient une solution optimale à un problème en effectuant une séquence de choix. À chaque point de décision, l'algorithme fait le choix qui semble le meilleur pour le moment. Cette stratégie heuristique ne produit pas toujours une solution optimale, mais comme nous l'avons vu dans le problème de la sélection d'activités, c'est parfois le cas. Cette section aborde certaines des propriétés générales des méthodes gloutonnes. Le processus que nous avons suivi dans la section I pour développer un algorithme glouton était un peu plus complexe que ce qui est typique. Nous sommes passés par les étapes suivantes :

1. Déterminer la sous-structure optimale du problème.
2. Développer une solution récursive. (Pour le problème de sélection d'activité, nous avons formulé la récurrence (2.2), mais nous avons contourné le développement d'un algorithme récursif basé sur cette récurrence.)
3. Montrer que si nous faisons le choix glouton, alors un seul sous-problème subsiste.

4. Prouvez qu'il est toujours prudent de faire un choix glouton. (Les étapes 3 et 4 peuvent se dérouler dans n'importe quel ordre.)
5. Développer un algorithme récursif qui implémente la stratégie gloutonne.
6. Convertir l'algorithme récursif en un algorithme itératif.

En parcourant ces étapes, nous avons vu en détail les fondements de la programmation dynamique d'un algorithme glouton. Par exemple, dans le problème de sélection d'activités, nous avons d'abord défini les sous-problèmes S_{ij}, où i et j variaient à la fois. Nous avons alors découvert que si nous faisions toujours le choix glouton, nous pourrions restreindre les sous-problèmes à la forme S_k. Alternativement, nous aurions pu façonner notre sous-structure optimale avec un choix glouton à l'esprit, de sorte que le choix ne laisse qu'un seul sous-problème à résoudre. Dans le problème de sélection d'activités, nous aurions pu commencer par supprimer le deuxième indice et définir les sous-problèmes de la forme S_k. Ensuite, nous aurions pu prouver qu'un choix glouton (la première activité doit se terminer dans S_k), combiné à une solution optimale de l'ensemble restant S_m d'activités compatibles, donne une solution optimale à S_k. Plus généralement, nous concevons des algorithmes gloutonnes selon la séquence d'étapes suivante :

1. Présenter le problème d'optimisation comme un problème dans lequel nous faisons un choix et nous nous retrouvons avec un sous-problème à résoudre.
2. Prouver qu'il existe toujours une solution optimale au problème d'origine qui fait le choix gourmand, afin que le choix gourmand soit toujours sûr.
3. Démontrer la sous-structure optimale en montrant que, après avoir fait le choix glouton, ce qui reste est un sous-problème avec la propriété que si nous combinons une solution optimale du sous-problème avec le choix glouton que nous avons fait, nous arrivons à une solution optimale à l'original problème.

Nous utiliserons ce processus plus direct dans les sections ultérieures de ce chapitre. Néanmoins, sous chaque algorithme glouton, il y a presque toujours une solution de programmation dynamique plus lourde. Comment pouvons-nous savoir si un algorithme glouton résoudra un problème d'optimisation particulier ? Aucun moyen ne fonctionne tout le temps, mais la propriété de choix glouton et la sous-structure optimale sont les deux ingrédients clés. Si nous pouvons démontrer que le problème a ces propriétés, alors nous sommes sur la bonne voie pour développer un algorithme glouton pour lui.

1. Propriété gloutonne

Le premier ingrédient clé est la propriété de choix glouton : nous pouvons assembler une solution globalement optimale en faisant des choix localement optimaux (gloutons). En d'autres termes, lorsque nous réfléchissons au choix à faire, nous faisons le choix qui semble le mieux dans le problème actuel, sans tenir compte des résultats des sous-problèmes. C'est ici que les algorithmes gloutons diffèrent de la programmation dynamique. En programmation dynamique, nous faisons un choix à chaque étape, mais le choix dépend généralement des solutions aux sous-problèmes. Par conséquent, nous résolvons généralement les problèmes de programmation dynamique de manière ascendante, en progressant de sous-problèmes plus petits vers des sous-problèmes plus importants. (Alternativement, nous pouvons les résoudre de haut en bas, mais en mémorisant. Bien sûr, même si le code fonctionne de haut en bas, nous devons toujours résoudre les sous-problèmes avant de faire un choix.) Dans un algorithme glouton, nous

faisons le choix qui nous semble le mieux pour le moment puis nous résolvons le sous-problème qui reste. Le choix fait par un algorithme glouton peut dépendre des choix effectués jusqu'à présent, mais il ne peut dépendre d'aucun choix futur ou des solutions aux sous-problèmes. Ainsi, contrairement à la programmation dynamique, qui résout les sous-problèmes avant de faire le premier choix, un algorithme glouton fait son premier choix avant de résoudre les sous-problèmes. Un algorithme de programmation dynamique procède de bas en haut, alors qu'une stratégie gloutonne progresse généralement de haut en bas, faisant un choix glouton après l'autre, réduisant chaque instance de problème donnée à une plus petite. Bien sûr, nous devons prouver qu'un choix glouton à chaque étape conduit à une solution globalement optimale. Typiquement, comme dans le cas du théorème 16.1, la preuve examine une solution globalement optimale à un sous-problème. Il montre ensuite comment modifier la solution pour substituer le choix glouton à un autre choix, ce qui entraîne un sous-problème similaire, mais plus petit.

Nous pouvons généralement faire le choix glouton plus efficacement que lorsque nous devons considérer un ensemble plus large de choix. Par exemple, dans le problème de sélection d'activités, en supposant que nous ayons déjà trié les activités par ordre croissant de façon monotone des temps de fin, nous devions examiner chaque activité une seule fois. En prétraitant l'entrée ou en utilisant une structure de données appropriée (souvent une file d'attente prioritaire), nous pouvons souvent faire des choix gloutons rapidement, produisant ainsi un algorithme efficace.

2. Sous-structure optimale

Un problème présente une sous-structure optimale si une solution optimale au problème contient en son sein des solutions optimales aux sous-problèmes. Cette propriété est un ingrédient clé pour évaluer l'applicabilité de la programmation dynamique ainsi que des algorithmes gloutonnes. Comme exemple de sous-structure optimale, rappelons comment nous avons démontré dans la section 16.1 que si une solution optimale au sous-problème S_{ij} inclut une activité a_k, alors elle doit également contenir des solutions optimales aux sous-problèmes S_{ik} et S_{kj}. Compte tenu de cette sous-structure optimale, en supposant que si nous savions quelle activité utilise a_k, nous pourrions construire une solution optimale à S_{ij} en sélectionnant a_k ainsi que toutes les activités dans des solutions optimales aux sous-problèmes S_{ij} et S_{kj}. Sur la base de cette observation de sous-structure optimale, nous avons pu concevoir la récurrence (2.2) qui décrit la valeur d'une solution optimale. Nous utilisons généralement une approche plus directe concernant la sous-structure optimale lorsque nous l'appliquons aux algorithmes gloutons. Comme mentionné ci-dessus, nous avons le luxe de supposer que nous sommes arrivés à un sous-problème en ayant fait le choix glouton dans le problème d'origine. Tout ce que nous avons vraiment besoin de faire est d'argumenter qu'une solution optimale au sous-problème, combinée au choix glouton déjà fait, donne une solution optimale au problème d'origine. Ce schéma utilise implicitement l'induction sur les sous-problèmes pour prouver que faire le choix glouton à chaque étape produit une solution optimale.

III. Exemple classique : Problème de séquencement de travaux avec délais

Étant donné un éventail de travaux où chaque travail a une date limite et un profit associé si le travail est terminé avant la date limite. Il est également donné que chaque travail prend une seule unité de temps, de sorte que le délai minimum possible pour tout travail est de 1. Comment maximiser le profit total si un seul travail peut être planifié à la fois.

Input: Four Jobs with following deadlines and profits

JobID	Deadline	Profit
A	4	20
B	1	10
C	1	40
D	1	30

Output: Following is maximum profit sequence of jobs C, A

PARTIE III : PROGRAMMATION DYNAMIQUE OU ALGORITHMES GLOUTONS ?

Parce que les stratégies de programmation dynamique et algorithmes gloutons exploitent toutes deux une sous-structure optimale, vous pourriez être tenté de générer une solution de programmation dynamique à un problème lorsqu'une solution convenue suffit ou, à l'inverse, vous pourriez penser à tort qu'une solution gloutonne fonctionne alors qu'en fait une solution de programmation dynamique est requise. Pour illustrer les subtilités entre les deux techniques, étudions deux variantes d'un problème d'optimisation classique.

Le problème du sac à dos variante *tout ou rien* est le suivant. Un voleur dévalisant un magasin trouve n objets. Le $i^{ème}$ élément vaut v_i dollars et pèse w_i livres, où v_i et w_i sont des nombres entiers. Le voleur veut emporter une charge aussi précieuse que possible, mais il peut transporter au plus W livres dans son sac à dos, pour un certain nombre entier W. Quels objets doit-il emporter ? (Nous appelons cela le problème du sac à dos variante tout ou rien car pour chaque objet, le voleur doit le prendre ou le laisser derrière lui ; il ne peut pas prendre une fraction d'un objet ou prendre un objet plus d'une fois).

Dans le problème du sac à dos fractionnaire, la configuration est la même, mais le voleur peut prendre des fractions d'objets, plutôt que d'avoir à faire un choix binaire (0-1) pour chaque article. Vous pouvez considérer un élément du problème du sac à dos variante tout ou rien comme un lingot d'or et un élément du problème du sac à dos fractionnaire comme de la poussière d'or. Les deux problèmes de sac à dos présentent la propriété de sous-structure optimale. Pour le problème 0-1, considérez la charge la plus précieuse qui pèse au plus W livres. Si nous retirons l'objet j de cette charge, la charge restante doit

être la charge la plus précieuse pesant au plus $W - w_j$ que le voleur peut prendre sur les $n - 1$ objets d'origine à l'exclusion de j. Pour le problème fractionnaire comparable, considérons que si nous retirons un poids w d'un élément j de la charge optimale, la charge restante doit être la charge la plus précieuse pesant au plus $W - w$ que le voleur peut prendre des $n - 1$ éléments d'origine plus $w_j - w$ livres de l'article j. Bien que les problèmes soient similaires, nous pouvons résoudre le problème du sac à dos fractionnaire par une stratégie gloutonne, mais nous ne pouvons pas résoudre le problème 0-1 par une telle stratégie. Pour résoudre le problème fractionnaire, nous calculons d'abord la valeur par livre $\frac{v_i}{w_i}$ pour chaque élément. Obéissant à une stratégie gloutonne, le voleur commence par prendre le plus possible de l'objet ayant la plus grande valeur par livre. Si la réserve de cet objet est épuisée et qu'il peut encore en transporter plus, il prend autant que possible de l'objet avec la valeur la plus élevée par livre suivante, et ainsi de suite, jusqu'à ce qu'il atteigne sa limite de poids W. Ainsi, en triant les éléments par valeur par livre, l'algorithme glouton s'exécute en $O(n \log n)$.

Pour voir que cette stratégie gloutonne ne fonctionne pas pour le problème du sac à dos 0-1, considérons l'instance de problème illustrée à la figure 3 (a). Cet exemple contient 3 articles et un sac à dos pouvant contenir 50 livres. L'article 1 pèse 10 livres et vaut 60 dollars. L'article 2 pèse 20 livres et vaut 100 dollars. L'article 3 pèse 30 livres et vaut 120 dollars. Ainsi, la valeur par livre de l'article 1 est de 6 dollars par livre, ce qui est supérieur à la valeur par livre de l'article 2 (5 dollars par livre) ou de l'article 3 (4 dollars par livre). La stratégie avide, par conséquent, prendrait le point 1 en premier. Comme vous pouvez le voir dans l'analyse de cas de la figure 3 (b), cependant, la solution optimale prend les éléments 2 et 3, laissant l'élément 1 de côté. Les deux solutions possibles qui prennent l'élément 1 sont toutes deux sous-optimales. Pour le problème fractionnaire comparable, cependant, la stratégie gloutonne, qui prend l'élément 1 en premier, donne une solution optimale, comme le montre la figure 3 (c). Prendre l'élément 1 ne fonctionne pas dans le problème 0-1 car le voleur est incapable de remplir son sac à dos à pleine capacité et l'espace vide réduit la valeur effective par livre de sa charge. Dans le problème 0-1, lorsque nous considérons s'il faut inclure un article dans le sac à dos, nous devons comparer la solution du sous-problème qui inclut l'article avec la solution du sous-problème qui exclut l'article avant de pouvoir faire le choix. Le problème formulé de cette manière donne lieu à de nombreux sous-problèmes qui se chevauchent - une caractéristique de la programmation dynamique, et en effet, comme l'exercice 16.2-2 vous demande de le montrer, nous pouvons utiliser la programmation dynamique pour résoudre le problème 0-1.

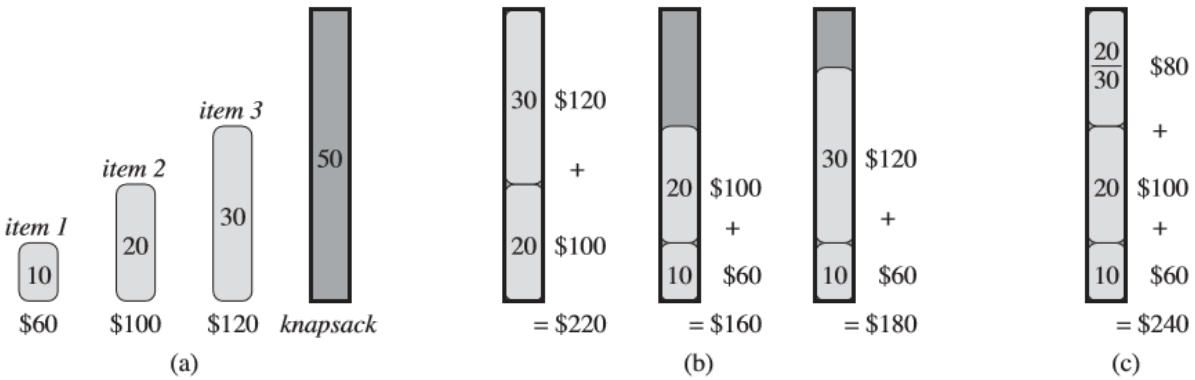


Figure 3 Un exemple montrant que la stratégie gloutonne ne fonctionne pas pour le problème du sac à dos 0-1. (a) Le voleur doit sélectionner un sous-ensemble des trois objets indiqués dont le poids ne doit pas dépasser 50 kg. (b) Le sous-ensemble optimal comprend les éléments 2 et 3. Toute solution avec l'élément 1 est sous-optimale, même si l'élément 1 a la plus grande valeur par livre. (c) Pour le problème du sac à dos fractionnaire, prendre les éléments dans l'ordre de la plus grande valeur par livre donne une solution optimale.

CONCLUSION

En résumé, pour notre thème de programmation dynamique et algorithmes gloutons nous avons abordé tout d'abord les grandes lignes de ce qu'est la programmation dynamique et les algorithmes gloutons, après, nous avons étudié quelques exemples, la comparaison de la programmation dynamique et algorithmes gloutons en a suivi, et enfin, nous avons situé la programmation dynamique dans les domaines de recherche. Pour les différents exemples abordés, nous avons calculé les différentes complexités algorithmiques, celles-ci varient d'un problème à l'autre. Il en ressort que la programmation dynamique peut être vue comme une amélioration de la méthode diviser pour régner, typiquement conçu pour résoudre les problèmes d'optimisation. Elle permet d'obtenir à coup sûr une solution optimale contrairement à d'autre méthode algorithmique résolvant également le même type de problème en occurrence la méthode gloutonne. Pourquoi dit-on que la programmation dynamique est une méthode algorithmique bien trop lourde pour trouver les meilleures solutions d'un problème ?

Bibliographie

- Introduction to Algorithms Third Edition, Thomas H. Cormen, Clifford Stein, Charles E. Leiserson, Ronald Rivest, 2009
- The Algorithm Design Manuel, Steven Skiena,1997
- Algorithm Design, John Kleinberg, Eva Tardos,2005

Webographie

- https://www.geeksforgeeks.org/dynamic-programming/
- https://www.geeksforgeeks.org/greedy-algorithms/

TRAVAUX DIRIGES

1. PROBLÈME DE RENDU DE MONNAIE

Considérons le problème de rendre la monnaie pour n FCFA en utilisant le plus petit nombre de pièces. Supposons que la valeur de chaque pièce est un nombre entier.

a. Décrivez un algorithme glouton pour effectuer un changement composé de pièces de monnaie $P = \{10, 5, 1\}$. Montrez que votre algorithme donne une solution optimale.
b. Supposons que les pièces disponibles soient dans les dénominations qui sont des puissances de c, c'est-à-dire que les dénominations sont $c^0, c^1, \dots, c^k$ pour certains entiers $c > 1$ et $k > 1$. Montrez que l'algorithme glouton donne toujours une solution optimale.
c. Donnez un ensemble de dénominations de pièces pour lesquelles l'algorithme glouton ne donne pas une solution optimale.
d. Donnez un ensemble de dénominations de pièces pour lesquelles l'algorithme glouton ne trouve pas de solution.

2. VARIANTE DU PROBLÈME DE RENDU DE MONNAIE

Nous considérons des pièces de monnaie de 1, 2, et 5 FCFA. Notons $N(x)$ le nombre minimum de pieces pour obtenir x centièmes.

1. Quelle est la valeur de $N(0), N(1), N(2), N(3), N(4)$ *et* $N(5)$.
2. Donner un algorithme qui calcule $N(x)$ et sa complexité en termes d'opération.

3. MISE EN CACHE HORS LIGNE

Les ordinateurs modernes utilisent un cache pour stocker une petite quantité de données dans une mémoire rapide. Même si un programme peut accéder à de grandes quantités de données, en stockant un petit sous-ensemble de la mémoire principale dans le cache—une mémoire petite mais plus rapide—le temps d'accès global peut considérablement diminuer. Lorsqu'un programme informatique s'exécute, il crée une séquence $\langle r_1, r_2, \dots, r_n \rangle$ de n demandes de mémoire, où chaque demande concerne un élément de données particulier. Par exemple, un programme qui accède à 4 éléments distincts $\{a, b, c, d\}$ pourrait faire la séquence de requêtes $\langle d, b, d, b, d, a, c, d, b, a, c, b \rangle$. Soit k la taille du cache. Lorsque le cache contient k éléments et que le programme demande le $(k+1)^{er}$ élément, le système doit décider, pour cette demande et pour chaque demande suivante, quels k éléments conserver dans le cache. Plus précisément, pour chaque requête r_i, l'algorithme de gestion de cache vérifie si l'élément r_i est déjà dans le cache. Si c'est le cas, alors nous avons un accès au cache ; sinon, nous avons un manque de cache. En cas d'échec du cache, le système récupère r_i dans la mémoire principale et l'algorithme de gestion du cache doit décider s'il faut conserver r_i dans le cache. S'il décide de conserver r_i et que le cache contient déjà k éléments, il doit alors expulser un élément pour faire de la place à r_i. L'algorithme de gestion du cache supprime les données dans le but de minimiser le nombre d'échecs de cache sur l'ensemble de la séquence de requêtes. Généralement, la mise en cache est un problème en ligne. C'est-à-dire que nous devons prendre des décisions sur les données à conserver dans le cache sans connaître les demandes futures. Ici, cependant, nous considérons la version hors ligne de ce problème, dans laquelle on nous donne à l'avance la séquence entière de n requêtes et la taille de cache k, et nous

souhaitons minimiser le nombre total d'échecs de cache. Nous pouvons résoudre ce problème hors ligne grâce à une stratégie simple appelée plus loin dans le futur, qui choisit d'expulser l'élément du cache dont le prochain accès dans la séquence de requêtes est le plus éloigné dans le futur.

a. Écrivez du pseudo-code pour un gestionnaire de cache qui utilise la stratégie la plus éloignée du futur. L'entrée doit être une séquence $\langle r_1, r_2, \ldots, r_n \rangle$ de demandes et une taille de cache k, et la sortie doit être une séquence de décisions concernant l'élément de données (le cas échéant) à expulser à chaque demande. Quel est le temps d'exécution de votre algorithme ?
b. Montrer que le problème de mise en cache hors ligne présente une sous-structure optimale.
c. Prouvez que le plus loin dans le futur produit le nombre minimum possible d'échecs de cache.

4. FRACTION ÉGYPTIENNE

Chaque fraction positive peut être représentée comme la somme de fractions unitaires uniques. Une fraction est une fraction unitaire si le numérateur est 1 et le dénominateur est un entier positif, par exemple $\frac{1}{3}$ est une fraction unitaire. Une telle représentation est appelée fraction égyptienne car elle était utilisée par les anciens Égyptiens.

Voici quelques exemples :

La représentation de la fraction égyptienne de $\frac{2}{3}$ est $\frac{1}{2} + \frac{1}{6}$

La représentation de la fraction égyptienne du $\frac{6}{14}$ est $\frac{1}{3} + \frac{1}{11} + \frac{1}{231}$

La représentation de la fraction égyptienne du $\frac{12}{13}$ est $\frac{1}{2} + \frac{1}{3} + \frac{1}{12} + \frac{1}{156}$

Nous pouvons générer des fractions égyptiennes en utilisant l'algorithme Glouton. Pour un nombre donné de la forme $\frac{nr}{dr}$ où $dr > nr$, trouver d'abord la plus grande fraction unitaire possible, puis se reproduire pour la partie restante. Par exemple, considérons 6/14, nous trouvons d'abord un plafond de 14/6, c'est-à-dire 3. Ainsi, la première fraction unitaire devient 1/3, puis se répète pour (6/14 - 1/3) c'est-à-dire 4/42.

5. LES POLICIERS ATTRAPENT LES VOLEURS

Étant donné un tableau de taille n qui a les spécifications suivantes :

Chaque élément du tableau contient soit un policier, soit un voleur.

Chaque policier ne peut attraper qu'un seul voleur.

Un policier ne peut pas attraper un voleur qui se trouve à plus de K unités du policier.

Nous devons trouver le nombre maximum de voleurs qui peuvent être attrapés.

Exemples :

Entrée : $arr[\,] = \{'P', 'T', 'T', 'P', 'T'\}$, $K = 1$.

Sortie : 2.

Ici maximum 2 voleurs peuvent être attrapés, d'abord un policier attrape le premier voleur et le deuxième policier peut attraper le deuxième ou le troisième voleur.

Entrée : $arr[\,] = \{'T','T','P','P','T','P'\}$, $K = 2$.

Sortie : 3.

Entrée : $arr[\,] = \{'P','T','P','T','T','P'\}$, $K = 3$.

Sortie : 3.

6. LA PLUS LONGUE SOUS-SÉQUENCE PALINDROMIQUE

Un palindrome est une chaîne non vide sur un alphabet qui se lit de la même manière en avant et en arrière. Des exemples de palindromes sont toutes les chaînes de longueur 1, *civic*, *racecar* et *aibohphobia*. Donnez un algorithme efficace pour trouver le palindrome le plus long qui soit une sous-séquence d'une chaîne d'entrée donnée. Par exemple, étant donné l'entrée *character*, votre algorithme doit renvoyer *carac*. Quel est le temps d'exécution de votre algorithme ?

7. LA DISTANCE D'EDITION

Afin de transformer une chaîne source de texte $x[1..m]$ en une chaîne cible $y[1..n]$, nous pouvons effectuer diverses opérations de transformation. Notre objectif est, étant donné x et y, de produire une série de transformations qui changent x en y. Nous utilisons un tableau z—supposé être suffisamment grand pour contenir tous les caractères dont il aura besoin—pour contenir les résultats intermédiaires. Initialement, z est vide, et à la fin, nous devrions avoir $z[j] = y[j]$ pour $j = 1, 2, \ldots, n$. Nous maintenons les indices actuels i dans x et j dans z, et les opérations sont autorisées à modifier z et ces indices. Initialement, $i = j = 1$. On est obligé d'examiner chaque caractère de x lors de la transformation, ce qui signifie qu'à la fin de la séquence d'opérations de transformation, on doit avoir $i = m + 1$.

Nous pouvons choisir parmi six opérations de transformation :

- **Copier** un caractère de x vers z en réglant $z[j] = x[i]$ puis en incrémentant à la fois i et j. Cette opération examine $x[i]$.
- **Remplacer** un caractère de x par un autre caractère c, en définissant $z[j] = c$, puis en incrémentant à la fois i et j. Cette opération examine $x[i]$.
- **Supprimer** un caractère de x en incrémentant i mais en laissant j seul. Cette opération examine $x[i]$.
- **Insérer** le caractère c dans z en définissant $z[j] = c$ puis en incrémentant j, mais en laissant i seul. Cette opération n'examine aucun caractère de x.
- **Twiddle** (c'est-à-dire échanger) les deux caractères suivants en les copiant de x à z mais dans l'ordre inverse ; nous le faisons en définissant $z[j] = x[i + 1]$ et $z[j + 1] = x[i]$ puis en définissant $i = i + 2$ et $j = j + 2$. Cette opération examine $x[i]$ et $x[i + 1]$.
- **Tuer** le reste de x en définissant $i = m + 1$. Cette opération examine tous les caractères de x qui n'ont pas encore été examinés. Cette opération, si elle est effectuée, doit être l'opération finale.

Par exemple, une façon de transformer la chaîne source *algorithm* en chaîne cible *altruistic* consiste à utiliser la séquence d'opérations suivante, où les caractères soulignés sont $x[i]$ et $z[j]$ après l'opération :

Operation	x	z
initial strings	algorithm	_
copy	algorithm	a_
copy	algorithm	al_
replace by t	algorithm	alt_
delete	algorithm	alt_
copy	algorithm	altr_
insert u	algorithm	altru_
insert i	algorithm	altrui_
insert s	algorithm	altruis_
twiddle	algorithm	altruisti_
insert c	algorithm	altruistic_
kill	algorithm_	altruistic_

Notez qu'il existe plusieurs autres séquences d'opérations de transformation qui transforment *algorithm* en *altruistic*. Chacune des opérations de transformation a un coût associé. Le coût d'une opération dépend de l'application spécifique, mais nous supposons que le coût de chaque opération est une constante qui nous est connue. Nous supposons également que les coûts individuels des opérations de copie et de remplacement sont inférieurs aux coûts combinés des opérations de suppression et d'insertion ; sinon, les opérations de copie et de remplacement ne seraient pas utilisées. Le coût d'une séquence donnée d'opérations de transformation est la somme des coûts des opérations individuelles de la séquence. Pour la séquence ci-dessus, le coût de la transformation de *algorithm* en *altruistic* est

$$\big(3 \cdot cout(copy)\big) + cout(replace) + cout(delete) + \big(4 \cdot cout(insert)\big) + cout(twiddle) + cout(kill)$$

Étant donné deux séquences $x[1..m]$ et $y[1..n]$ et un ensemble de coûts d'opération de transformation, la distance d'édition de x à y est le coût de la séquence d'opérations la moins chère qui transforme x en y. Décrivez un algorithme de programmation dynamique qui trouve la distance d'édition de $x[1..m]$ à $y[1..n]$ et imprime une séquence d'opérations optimale. Analysez le temps d'exécution et les besoins en espace de votre algorithme.

8. PROBLÈME DU SAC À DOS

Considérons les deux variantes du problème du sac à dos suivantes :

Variante tout ou rien

Un voleur dévalisant un magasin trouve n objets, le i^{ieme} de ces objets valant v_i FCFA et pesant w_i kilos, v_i et w_i étant des entiers. Le voleur veut bien évidement emporter un butin de plus grande valeur possible mais il ne peut porter que W kilos dans son sac à dos. Quels objets devrait-il prendre ?

Variante fractionnaire

Le problème est le même que le précèdent, sauf qu'ici peut ne prendre qu'une fraction d'un objet et n'est plus obligé de prendre l'objet tout entier comme précédemment, ou de le laisser.

1. Proposer un algorithme glouton optimal pour la **variante fractionnaire**.
2. Quelle est sa complexité ?
3. L'algorithme précèdent reste-t-il optimal pour **la variante tout ou rien** ? Justifier votre réponse.
4. Proposer un algorithme de programmation dynamique pour la **variante tout ou rien**.