SÉANCE DE SOUTIEN

1 Différents types d'algos

Dans les exercices qui suivent, il faudra trouver

- 2 algos de programmation dynamique,
- 1 algo de type diviser pour régner,
- 1 algo glouton.

Pour calculer les complexités, on supposera dans tous les exercices sauf le 1.3, que les opérations sur les entiers ou sur les réels (addition, multiplication...) se font en $O(1)$.

1.1 Rendre la monnaie

On suppose que l'on a accès à des pièces de valeur $0 < s_1 < \cdots < s_k \in \mathbb{N}$ (on a une infinité de pièces de chaque valeur). Soit $x \in \mathbb{N}$ une somme à atteindre, on veut calculer le nombre de façons différentes d'obtenir x en choisissant des pièces de valeur dans $\{s_1, \cdots, s_k\}$.

1. En prenant $s_1 = 1, s_2 = 3$ et $s_3 = 5$, combien y a-t-il de façon d'obtenir $x = 10$?

   A: *On met en pratique l'algorithme de programmation dynamique décrit dans la question d'après. On construit le tableau des $T(j, x)$ (nombre de façon de faire x avec $s_1, \cdots, s_j$) ligne par ligne en utilisant la formule $T(j+1, x) = T(j, x) + T(j, x - s_{j+1}) + T(j, x - 2s_{j+1}) + \cdots$*

   ```
   |j\x|0|1|2|3|4|5|6|7|8|9|10|
   | 1 |1|1|1|1|1|1|1|1|1|1| 1 |
   | 2 |1|1|1|2|2|2|3|3|3|4| 4 |
   | 3 |1|1|1|2|2|3|4|4|5|6| 7 |
   ```

   Le ligne 2 dépend uniquement de la ligne 1, et la ligne 3 dépend uniquement de la 2.

2. Donner un algorithme pour résoudre ce problème (attention, (s_1, s_2) et (s_2, s_1) donnent la même façon de réaliser $s_1 + s_2$).

   A: *On fait de la programmation dynamique.*

   Soit $T(j, x)$ le nombre de façon de faire x avec des pièces $s_1, \cdots, s_j$. On a $T(j+1, x) = T(j, x) + T(j, x - s_{j+1}) + T(j, x - 2s_{j+1}) + \cdots$ (on distingue selon que l'on prend 0 pièce s_{j+1} pour faire x, ou une pièce s_{j+1}, où 2 pièces s_{j+1}...). Pour les cas de base, on a $T(j, 0) = 1$, $T(j, y) = 0$ si $y < 0$ et $T(1, x) = 1$ si s_1 divise x et 0 sinon. On calcule ensuite $T(j, x')$ pour tout $j \leq k$ et $x' \leq x$ par programmation dynamique. On construit un tableau où chaque ligne dépend seulement de la précédente. La complexité pour calculer $T(k, x)$ est la taille du tableau ($O(kx)$) fois la complexité pour calculer une case du tableau ($O(x)$, il faut potentiellement regarder toutes les cases de la ligne du dessus), c'est à dire $O(kx^2)$. Pour un exemple, voir la question précédente.

3. Refaire la question 1 en utilisant votre algorithme.

1.2 Arbre couvrant de poids minimal

Soit $G = (V, E)$ un graphe connexe non orienté et $w : E \to]0, +\infty[$ une fonction de poids sur les arêtes. l'objectif est de calculer un arbre couvrant de poids minimal. Un arbre couvrant pour le graphe G est un sous ensemble d'arêtes $E' \subset E$ tel que tous les sommets de V appartiennent au moins à une arête de E' (c'est le côté "couvrant") et tel que le graphe (V, E') soit connexe et ne contiennent pas de cycle (c'est le côté "arbre"). On veut donc trouver, parmi tous les arbres couvrants de G, celui dont le poids, c'est à dire $\sum_{e' \in E'} w(e')$, est minimal.

1. Proposer un algorithme pour résoudre ce problème (au choix parmi programmation dynamique, diviser pour régner ou glouton). Quelle est sa complexité ?

   A: *On peut faire un algorithme glouton, par exemple l'algorithme de Kruskal. L'idée est de commencer par trier les arêtes par poids croissant. Ensuite on construit l'arbre en prenant les arêtes dans l'ordre, du poids le plus faible au plus gros. Si l'arête ne crée par de cycle, on l'ajoute à l'arbre, sinon on passe à la suivante.*

   La complexité de l'algorithme est en $O(m \log(m) + mn) = O(mn)$ où m est le nombre d'arêtes et n le nombre de sommets. Le coût du tri des arêtes est $O(m \log(m))$. Ensuite, pour chaque arête il faut vérifier qu'elle ne crée pas de cycle. Cela peut se faire en parcourant les arêtes déjà sélectionnées, et il y a au plus n arêtes déjà sélectionnées (un arbre a au plus $n - 1$ arêtes), donc ajouter une nouvelle arête coûte $O(n)$ et il faut faire ça m fois, d'où $O(mn)$.

   Preuve de correction de l'algorithme. Déjà, on peut noter que par définition, on obtient une structure sans cycle, car on rejette les arêtes dès qu'elles créent un cycle. Ensuite, on a bien une structure connexe car sinon, comme le graphe d'origine est connexe, on aurait une arête entre les deux composantes qui aurait été rajouté lors de l'algorithme (une arête entre deux composantes connexes ne crée pas de cycle). La structure obtenue par cet algorithme est donc bien un arbre couvrant. Il reste à voir qu'il est minimal. On peut le faire par récurrence, en montrant qu'à chaque étape de l'algorithme il existe un arbre couvrant de poids minimal qui contient les arêtes déjà sélectionnées. Voir Wikipédia pour plus de détails : `https://en.wikipedia.org/wiki/Kruskal%27s_algorithm?oldid=684523029#Proof_of_correctness`.

1.3 Calcul de factorielle

L'objectif de cet exercice est d'estimer le temps de calcul de $n! = n(n-1)\cdots 2$ en fonction du coût de la multiplication de deux entiers (dans cet exercice, on suppose que le coût de la multiplication de deux entiers dépend de la taille des entiers, et n'est pas en $O(1)$). Dans la suite de l'exercice, on suppose que multiplier deux entiers $a, b \leq N$ prend un temps $O(M(N))$. L'algorithme naïf de multiplication que l'on apprend en primaire donne $M(N) = \log(N)^2$, mais on peut faire mieux. Par exemple, en utilisant la transformée de Fourrier rapide, on peut avoir $M(N) = \log(N) \cdot \log\log(N) \cdot \log\log\log(N)$ (ce que l'on arrondira en $\log(N)$). Dans la suite de l'exercice, on utilisera $M(N)$ et on discutera dans la dernière question des complexité que l'on obtient selon le choix de la multiplication choisie.

Plutôt que de donner un algorithme pour calculer la factorielle, on va donner un algorithme un peu plus général, qui étant donné k entiers $a_1, \cdots, a_k \leq n$, calcule le produit de ces k éléments.

1. Donner un algorithme naïf pour calculer le produit des k éléments $a_1, \cdots, a_k$. Quelle est sa complexité ?

   A: *On multiplie les éléments un a un. Le produit des éléments $a_1 \cdot a_2 \cdots a_j$ est inférieur à n^j et $a_{j+1} \leq n \leq n^j$ donc multiplier $a_1 \cdot a_2 \cdots a_j$ et a_{j+1} prend un temps $O(M(n^j))$. Comme $j \leq k$ et que l'on répète la multiplication k fois, on obtiens une complexité totale en $O(k \cdot M(n^k))$. Ce qui fait $O(k^3 \log(n)^2)$ ou $O(k^2 \log(n))$ selon que l'on prend $M(N) = \log(N)^2$ ou $M(N) = \log(N)$.*

2. Donner un algorithme plus malin pour calculer le produit des k éléments (à choisir parmi programmation dynamique, diviser pour régner ou glouton).

   A: *On fait un algo diviser pour régner : on multiplie par récurrence les éléments $a_1, \cdots, a_{k/2}$ et les éléments $a_{k/2+1}, \cdots, a_k$, puis on multiplie les 2 gros éléments obtenus entre eux. Si on note $T(k)$ la complexité pour multiplier k éléments inférieurs à n, on obtient $T(k) = 2T(k/2) + O(M(n^{k/2}))$.*

3. Calculer la complexité de l'algorithme précédent dans le cas où $M(N) = \log(N)^2$ puis dans le cas où $M(N) = \log(N)$.

   A: *Dans le cas où $M(N) = \log(N)^2$, on a l'équation $T(k) = 2T(k/2) + O(k^2)$ (en supposant que n est fixé). La partie récurrence $2T(k/2)$ apporte une complexité k, ce qui est négligeable devant le $O(k^2)$ nécessaire pour multiplier les deux éléments finaux. La complexité est donc $O(k^2)$. Dans le cas où $M(N) = \log(N)$, les deux termes de la récurrence ont le même ordre de grandeur (ils valent $O(k)$ tous les deux). On obtient donc une complexité en $O(k\log(k))$.*

1.4 Coefficients binomiaux

L'objectif de cet exercice est de calculer tous le coefficient binomial $\binom{n}{k}$.

1. On rappelle le triangle de Pascal : $\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}$. Donner un algorithme pour calculer $\binom{n}{k}$ en utilisant le triangle de Pascal. Quelle est sa complexité (en terme d'opérations sur les entiers) ?

   A: *Le triangle de Pascal donne naturellement un algorithme de programmation dynamique. On stocke dans une table les valeurs de $\binom{n'}{k'}$ pour tous les $k' \leq k$ et $n' \leq n$. On initialise les cases $\binom{n'}{0}$ et $\binom{n'}{n'}$ avec 1, puis on construit les autres lignes par lignes. Cela demande une opération sur des entiers pour chaque case. Et il y a $O(kn)$ cases à remplir, d'où une complexité en $O(kn)$.*

2. En utilisant l'algorithme naïf de l'exercice précédent pour calculer le produit de k éléments inférieurs à n et en supposant que la multiplication de deux entiers coûte $O(1)$ quelque soit leur taille, la formule $\binom{n}{k} = \frac{n!}{k!(n-k)!}$ permet d'obtenir un algorithme en $O(k)$ pour calculer le coefficient binomial $\binom{n}{k}$. Est-ce mieux que votre algorithme précédent ? Qu'en est-il si vous voulez calculer tous les coefficients binomiaux $\binom{n'}{k'}$ pour $k' \leq k$ et $n' \leq n$?

   A: *L'algorithme de programmation dynamique est moins performant que l'algorithme direct pour calculer un coefficient binomial ($O(nk)$ vs $O(k)$). Mais il permet d'avoir pour le même prix d'avoir tous les coefficients binomiaux plus petits. Alors qu'avec l'algorithme direct, il faut les recalculer tous, ce qui donne une complexité $O(nk^2)$, moins bonne que l'algorithme de programmation dynamique.*

2 Problèmes NP-complets

Montrer que les problèmes suivants sont NP-complets.

1. 3-SAT-NAE (not all equal) : Étant donné une conjonction de clauses de taille 3, déterminer s'il existe un assignement des variables tel qu'il n'existe aucune clause où tous les littéraux sont égaux.

   A: *3-SAT-NAE est bien dans NP car si on nous donne l'assignement des variables qui convient, on peut le vérifier en temps polynomial.*

   Pour montrer la complétude, on réduit le problème 3-SAT au problème 3-SAT-NAE. Soit ϕ une conjonction de clauses de taille 3 dont on veut savoir si elle est satisfiable (on veut résoudre 3-SAT pour ϕ). On construit ψ qui sera dans 3-SAT-NAE si et seulement si ϕ est dans SAT, ce qui nous donne notre réduction (one-to-one car on fait appel une seule fois à notre boite noire qui résout 3-SAT-NAE pour résoudre 3-SAT).

   La transformation de ϕ à ψ est la suivante. Pour chaque clause $a \vee b \vee c$ de ϕ, on introduit une nouvelle variable w et on ajoute à ψ les clauses $(a \vee b \vee w) \wedge (c \vee -w \vee 0)$. Si ϕ est 3-satisfiable, alors $a \vee b = 1$ ou $c = 1$. Dans le premier cas, on choisi $w = 0$, dans le second $w = 1$ et on obtient une solution pour ψ pour 3-SAT-NAE. Réciproquement, supposons que

ψ soit dans 3-SAT-NAE, alors les deux clauses sont satisfaites, ce qui signifie que soit $a \vee b = 1$ (si $w = 0$), soit $c = 1$ (si $w = 1$) et donc que ϕ est 3-satisfiable. On a donc bien que ϕ est dans SAT ssi ψ est dans 3-SAT-NAE, et ψ peut être obtenu en temps polynomial à partir de ϕ (et est de taille polynomiale en la taille de ϕ), ce qui conclut la réduction.

2. 3-SAT-OIT (one in three) : Étant donné une conjonction de clauses de taille 3, déterminer s'il existe un assignement des variables tel que chaque clause possède exactement un littéral à `vrai` et les deux autres à `faux`.

   ***A:** Comme précédemment, 3-SAT-OIT est bien dans NP.*

   Soit ϕ une conjonction de clauses. Montrons qu'on peut transformer ϕ en une nouvelle formule ψ telle que ϕ est dans 3-SAT ssi ψ est dans 3-SAT-OIT, et que cette transformation peut se faire en temps polynomial.

   Soit $(a \vee b \vee c)$ une clause de ϕ. On la remplace dans ψ par la conjonction $(-a \vee w_1 \vee w_2) \wedge (b \vee w_2 \vee w_3) \wedge (-c \vee w_3 \vee w_4)$, où w_1, w_2 et w_3 sont des nouvelles variables, spécifiques à cette clause. La transformation s'effectue bien en temps polynomial. Il reste à montrer que ϕ est 3-satisfiable ssi ψ est dans 3-SAT-OIT. On remarque que le triplet $(w_1 \vee w_2, w_2 \vee w_3, w_3 \vee w_4)$ peut prendre toutes les valeurs possibles dans $\{0,1\}^3$ sauf la valeur $(0,1,0)$. Donc on pourra trouver des valeurs de w_1, w_2, w_3 et w_4 telles que ψ soit dans 3-SAT-OIT sauf si $(-a, b, -c) = (1,0,1)$, i.e. $(a,b,c) = (0,0,0)$. On en conclut que ψ est dans 3-SAT-OIT ssi ϕ est satisfiable.

3. Max-2-SAT : Étant donné une conjonction de clauses de taille 2 et un entier k, déterminer s'il existe un assignement des variables tel que au moins k clauses soient satisfaites.

   ***A:** Réduction depuis 3-SAT. Une clause $(a \vee b \vee c)$ de 3-SAT est transformée en l'ensemble de clauses suivant $\{w, a, b, c, (-a \vee -b), (-a \vee -c), (-b \vee -c), (-w, a), (-w, b), (-w, c)\}$, où w est une nouvelle variable, qui change pour chaque clause. Si la clause est satisfiable, on peut choisir w pour que 7 des clauses soient satisfaites (et on ne peut pas faire mieux quelque soient les choix de a, b, c et w). Si la clause n'est pas satisfiable, on ne peut pas faire mieux que 6. On choisit donc $k = 7 \times$ nb de clauses. Pour plus de détails sur la réduction, voir* `https://www.nitt.edu/home/academics/departments/cse/faculty/kvi/NPC%203SAT-MAX%202SAT.pdf`.