IFT436 - Algorithmes et structures de données
Notes de cours

Manuel Lafond
Université de Sherbrooke

Table des matières

1 Rappels mathématiques **4**
- 1.1 Ensembles et séquences . . . 5
- 1.2 Fonctions . . . 6
- 1.3 Rappels de base en logique . . . 7
- 1.4 Propriétés de base des logarithmes et exposants . . . 8
- 1.5 Bases de combinatoire . . . 8
- 1.6 Techniques de preuve . . . 10
- 1.7 Résumé . . . 14

2 La notation asymptotique **15**
- 2.1 L'ensemble de fonctions $O(f(n))$ . . . 16
- 2.2 Définition formelle de O . . . 17
- 2.3 Notation asymptotique multivariée . . . 20
- 2.4 La notation Ω . . . 21
- 2.5 La notation Θ . . . 21
- 2.6 Fonctions lisses (* optionnel) . . . 22
- 2.7 Comment montrer que $f(n) \notin O(g(n))$? . . . 23
- 2.8 Résumé . . . 23

3 Le tri **24**
- 3.1 Le tri insertion . . . 25
- 3.2 Le tri fusion . . . 28
- 3.3 Quicksort . . . 31
- 3.4 Borne inférieure du tri par comparaison [section optionnelle] . . . 32
- 3.5 Tri sans comparaisons . . . 35
- 3.6 Le tri comptage . . . 35
  - 3.6.1 Comment trier des clés . . . 36
  - 3.6.2 Stabilité d'un tri . . . 37
- 3.7 Le tri radix . . . 38

3.8 Récapitulatif des tris 41
3.9 Résumé 43

4 Les graphes **44**
4.1 Quelques définitions 45
4.2 Chemins, cycles et connexité 47
4.3 Chemin le plus court 48
4.4 Fouille en largeur 51
4.5 Fouille en profondeur 53
4.6 Arbres 53
4.7 Représentation d'un graphe 55
4.8 Graphes orientés acycliques et tri topologique 57
4.9 Résumé 59

5 Algorithmes gloutons **60**
5.1 Problème de la sélection 61
5.2 Arbres couvrants minimums et algorithme de Prim 63

6 Algorithmes diviser-pour-régner **68**
6.1 Cas d'étude : la recherche dichotomique 70
6.2 Les deux points les plus proches, version 1D 71
6.3 Multiplication de grands entiers 74
6.4 Le Théorème Maître 76
6.5 Les deux points les plus proches, version 2D 77
6.6 Résumé 79

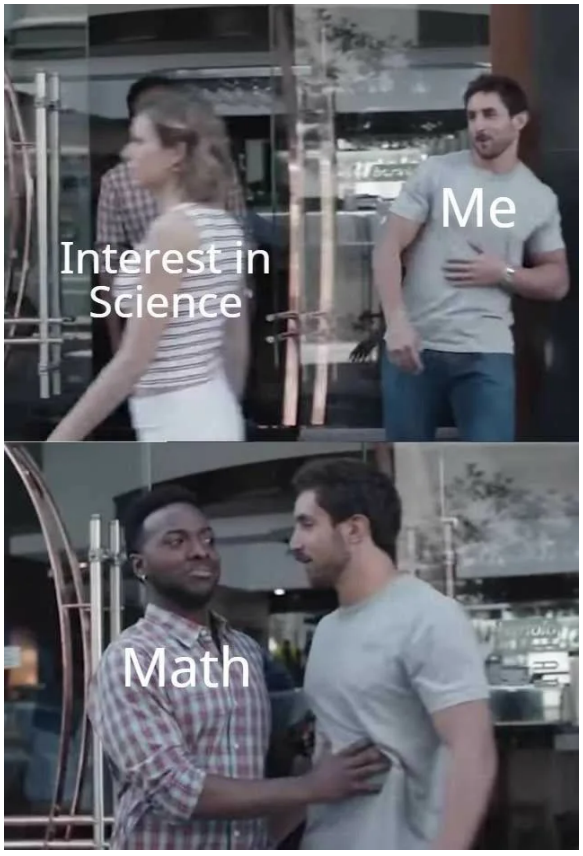
7 Algotihmes exact exponentiels **80**
7.1 Force brute pour le problème de la sous-somme 80
7.2 Recette de résolution de récurrences 84
7.3 Ensemble indépendant maximum dans un graphe 85
7.4 Résumé 86

8 Programmation dynamique **87**
8.1 Fibonacci amélioré 88
8.2 Faire de la monnaie avec un minimum de pièces 90
8.3 Principe d'optimalité de Bellman 94
8.4 Ensemble indépendant dans un arbre 94
8.5 Algorithme de Dijkstra 97
8.6 Algorithme de Floyd-Warshall 100
8.7 Résumé 102

9 Algorithmes probabilistes **103**
9.1 Espérance en temps 103
9.2 Algorithmes Monte Carlo et Las Vegas 106
9.3 Quicksort probabiliste 107
9.4 Algorithme de Karger pour la coupe minimum 111
9.5 Résumé 116

Chapitre 1

Rappels mathématiques



Dans ce chapitre, nous revoyons les bases mathématiques nécessaires à l'analyse algorithmique. Nous révisons les propriétés des logarithmes et exposants, les bases en logique, en combinatoire, puis les techniques de preuve principales.

1.1 Ensembles et séquences

Une *séquence* est une liste ordonnée d'objets, avec répétitions possibles. On dénote souvent une séquence avec des parenthèses. Par exemple, la séquence $S = (1, 1, 2, 3, 5, 8, 13)$ contient les 7 premiers nombres Fibonacci. On peut dénoter une séquence avec les crochets, e.g. $S = [1, 1, 2, 3, 5, 8, 13]$.

En contraste avec les séquences, un *ensemble* est une collection d'objets *non-ordonnée*. Il n'y a pas de répétition dans un ensemble : chaque élément d'un ensemble X est présent une seule fois. Il n'y a pas de contrainte quant à la nature des objets dans X, et ceci dépend du contexte. Un ensemble peut notamment contenir d'autres ensembles. t à la nature des objets dans X, et ceci dépend du contexte. Un ensemble peut notamment contenir d'autres ensembles. On utilise les accolades pour dénoter le contenu d'un ensemble. Par exemple, si X est l'ensemble des 5 premiers entiers pairs positifs, on écrit

$$X = \{2, 4, 6, 8, 10\}$$

Rappelons que même si la ligne ci-haut suggère un ordre, il n'y a pas d'ordre dans un ensemble. Il serait équivalent d'écrire $X = \{4, 8, 10, 2, 6\}$. Un ensemble peut être fini (comme l'exemple ci-dessus) ou infini. Par exemple, l'ensemble de tous les entiers non-négatifs est dénoté

$$\mathbb{N} = \{0, 1, 2, 3, \ldots\}$$

Dans le cas d'un ensemble fini X, la *cardinalité* de X est dénotée par $|X|$ et réfère au nombre d'éléments de X. Dans l'exemple de X ci-dessus, on aurait $|X| = 5$. L'ensemble *vide* est dénoté par $\emptyset$. Notez qu'un ensemble peut contenir un ensemble vide. Si $X = \emptyset$ et $Y = \{\emptyset\}$, on a $|X| = 0$ et $|Y| = 1$.

Pour indiquer qu'un élément x est contenu dans l'ensemble X, on écrit $x \in X$. Si x n'est *pas* dans X, on écrit plutôt $x \notin X$. Si X et Y sont deux ensembles, on écrit $X \subseteq Y$ pour dire que X est un sous-ensemble de Y, c'est-à-dire que tous les éléments de X sont aussi dans Y. On écrit $X \subset Y$ si $X \subseteq Y$ mais que $X \neq Y$ (ce qui se produit quand Y a des éléments que

X n'a pas).

On peut définir un ensemble avec les "..." lorsqu'il est construit à partir d'une suite logique - mais seulement lorsque cette suite est évidente à voir. Par exemple, si X est l'ensemble des nombres pairs entre 0 et $2n$, on peut écrire

$$X = \{0, 2, 4, 6, \ldots, 2n-2, 2n\}$$

On peut aussi utiliser la notation $X = \{x : \text{conditions sur } x\}$ pour définir le contenu de X. Le symbole : veut dire "tels que" et ici, X est construit avec "tous les x tels que la condition sur x est satisfaite". Certains utilisent le "|" au lieu du ":". Voici l'ensemble X des nombres pairs ci-haut définis avec cette notation :

$$X = \{x : 0 \leq x \leq 2n \text{ et } x \text{ est pair}\}$$

Soit X et Y deux ensembles. On peut faire l'union $\cup$, l'intersection $\cap$ et la différence $\setminus$ entre X et Y, définis comme suit :

$$\begin{aligned} X \cup Y &= \{z : z \in X \text{ ou } z \in Y\} \\ X \cap Y &= \{z : z \in X \text{ et } z \in Y\} \\ X \setminus Y &= \{z : z \in X \text{ et } z \notin Y\} \end{aligned}$$

Quelques questions à vous poser : est-il vrai que $X \cup Y = Y \cup X$, $X \cap Y = Y \cap X$ et $X \setminus Y = Y \setminus X$?

1.2 Fonctions

Une *fonction* associe chaque valeur d'un ensemble X avec exactement une valeur d'un ensemble Y. Par exemple, soient les ensembles $\{a, b, c\}$ et $\{0, 1\}$ et disons qu'on associe a et b à 0 et c à 1. On décrit ceci par la fonction $f : \{a, b, c\} \to \{0, 1\}$ définie par $f(a) = f(b) = 0$ et $f(c) = 1$.

De façon générale, on écrit

$$f : X \to Y$$

pour dire que la fonction f associe des éléments de X à des éléments de Y. On appelle X le domaine de f et Y le co-domaine de f. Le domaine peut être infini, par exemple, $f : \mathbb{R} \to \mathbb{R}$ définie par $f(n) = n^2$. Chaque

$x \in X$ doit être associé à exactement un $y \in Y$, dénoté $f(x)$, mais il est possible qu'un $y \in Y$ n'ait aucun x qui lui est associé (comme n^2 ci-haut). Une fonction $f : X \to Y$ est *injective* si $f(x) \neq f(x')$ pour toute paire $\{x, x'\} \subseteq X$ et elle est *surjective* si $\forall y \in Y, \exists x \in X, f(x) = y$. Finalement, f est bijective si elle est à la fois injective et surjective.

1.3 Rappels de base en logique

Un prédicat est, de façon informelle, un énoncé qui est soit vrai, soit faux. Par exemple, le prédicat "$\forall n \in \mathbb{Z}$, $2n$ est un nombre pair" évalue à "vrai", mais "$\forall n \in \mathbb{Z}$, n^2 est un nombre pair" évalue à "faux". Rappelons que $\forall$ veut dire "pour tous les" et $\exists$ veut dire "il existe". Le dernier prédicat est faux parce que $\exists n \in \mathbb{Z}$ tel que n^2 est impair, et donc ce n'est pas vrai pour tous les n (par exemple $n = 1$).

On peut combiner deux prédicats avec "et" et/ou "ou" (dites-le à voix haute). Si p et q sont deux prédicats, $p \wedge q$ veut dire "p et q", alors que $p \vee q$ veut dire "p ou q". On écrit $\neg p$ pour la négation de p. Lorsque nous ferons des preuves par contradiction (ou peut-être par contraposition), il sera très important de savoir exprimer la négation d'un prédicat. On a les règles suivantes :

- $\neg(\forall x, p(x)) \Leftrightarrow \exists x, \neg p(x)$
- $\neg(\exists x, p(x)) \Leftrightarrow \forall x, \neg p(x)$
- $\neg(p \wedge q) \Leftrightarrow \neg p \vee \neg q$
- $\neg(p \vee q) \Leftrightarrow \neg p \wedge \neg q$

où $\Leftrightarrow$ veut dire "est équivalent à" (voir ci-bas).

L'implication s'écrit $p \Rightarrow q$ et veut dire "si p est vrai, alors q est vrai". On écrit $p \Leftrightarrow q$ si $p \Rightarrow q$ et $q \Rightarrow p$, et on dit que p si et seulement si. Dans ce cas, p et q sont équivalents. Notons que lorsque $p \Rightarrow q$, si p est faux, ça n'a aucune implication sur q et si q est vrai, il n'y a aucune implication sur p. Inverser le sens d'une implication est d'ailleurs un sophisme répandu [1]. La norme veut que lorsque p est faux, le prédicat $p \Rightarrow q$ est vrai (parce que $\neg p$ ne vient pas contredire l'implication).

Par contre, si q est faux, on sait que p est faux. D'ailleurs, on a que $p \Rightarrow q \Leftrightarrow \neg q \Rightarrow \neg p$. Notons finalement que $p \Rightarrow q$ est équivalent à $\neg p \vee q$.

1. Voir `https://fr.wikipedia.org/wiki/Affirmation_du_cons%C3%A9quent` pour plus d'informations.

1.4 Propriétés de base des logarithmes et exposants

L'exponentiation et les logarithmes sont inhérents à l'analyse algorithmique. La maîtrise de quelques règles de base permet d'accélérer plusieurs calculs.

On écrit $\log_b a$ pour désigner l'exposant qu'il faut donner à b pour obtenir a. C'est l'inverse de l'exponentiation, et donc $log_b a = x$ si et seulement si $b^x = a$. Dans le cadre de ce cours, si b n'est pas spécifié, on suppose que $b = 2$ (notons que certains écrivent $\lg a = \log_2 a$). Si ce n'est pas déjà le cas, vous devrez apprendre à être à l'aise avec les propriétés des logarithmes et exposants. Pour $a > 0, b > 0, c > 0$ et $a, b, c, k \in \mathbb{R}$, on a :

- $\log_b(a^k) = k \log_b a$
- $a = b^{\log_b a}$
- $\log_c(ab) = \log_c a + \log_c b$
- $\log_c 1 = 0$ et $\log_c 0$ n'est pas défini
- $\log_c(a/b) = \log_c a - \log_c b$
- $\log_b a = \frac{\log_c a}{\log_c b}$
- $a^b a^c = a^{b+c}$
- $(a^b)^c = a^{bc}$
- si $a \leq b^k$, alors $\log_c a \leq \log_c(b^k) = k \log_c b$

1.5 Bases de combinatoire

Permutations

Soit $S = \{s_1, \ldots, s_n\}$ un ensemble. Une *permutation* de S est une séquence ordonnée contenant tous les éléments de S. Par exemple, si $S = \{1, 2, 3\}$, on a 6 permutations possibles : $(1, 2, 3), (1, 3, 2), (2, 1, 3), (2, 3, 1), (3, 1, 2)$ et $(3, 2, 1)$. En général, il y a $n!$ permutations possibles d'un ensemble de n éléments. Ici $n!$ dénote le factoriel de n, où $n! = n \cdot (n-1) \cdot \ldots \cdot (n-2) \cdot 2 \cdot 1$, avec $0! = 1$ comme cas spécial. On peut aussi définir $n!$ récursivement :

$$n! = \begin{cases} 1 & \text{si } n = 0 \\ n \cdot (n-1)! & \text{si } n > 0 \end{cases}$$

Prenez le temps de vous demander pourquoi $n!$ est le nombre de permuations.

Nombre de sous-ensembles

Une autre propriété combinatoire très utile est le nombre de sous-ensembles d'un ensemble S. Il arrive fréquemment que l'on conçoive un algorithme force-brute qui teste tous les sous-ensembles d'un ensemble — comme dans le problème de la sous-somme. Soit $n = |S|$. On peut déduire qu'il y a 2^n sous-ensembles de S comme suit. Soit $S = \{s_1, \ldots, s_n\}$ les éléments de S. Pour construire un sous-ensemble $X \subseteq S$, on a deux choix pour s_1 : soit il est dans X, ou il ne l'est pas. Ensuite, pour s_2, on a les mêmes deux choix, puis deux choix pour s_3, etc. Le nombre de combinaisons de choix est donc $2 \cdot 2 \cdot \ldots \cdot 2$, où on multiplie 2 n fois. On déduit qu'il y a 2^n façons de construire un sous-ensemble différent.

Une autre façon instructive de voir ce résultat est la suivante. On fait correspondre à chaque sous-ensemble $X \subseteq S$ une séquence de n bits dans laquelle le bit i est 1 si $s_i \in X$, et le bit i est 0 si $s_i \notin X$. Chaque sous-ensemble correspond à une séquence distincte de n bits, et chaque séquence de n bits correspond à un sous-ensemble distinct de S. De plus, il y a 2^n chaînes de n bits possibles, car il y a 2 choix pour chaque bit, et donc il y a 2^n sous-ensembles possibles. On a établi une *bijection* entre les sous-ensembles et les chaînes de n bits. En combinatoire, il est fréquent d'utiliser une telle technique : si on ne sait pas comment compter un nombre d'éléments, on peut tenter de les mettre en bijection avec un ensemble qu'on sait comment compter.

Coefficient binomial

Maintenant, pour S un ensemble et un certain k, combien y a-t-il de sous-ensembles de S de taille k ? Si $k = 0$, il y en a un seul, c'est $\emptyset$. Si $k = 1$, il y a n choix possibles. Pour k plus grand, il faut compter.

On écrit $\binom{n}{k}$ pour dénoter le nombre de sous-ensembles de taille k d'un ensemble de taille n. Autrement dit, $\binom{n}{k}$ désigne le nombre de façons de choisir k objets parmi n. On dit d'ailleurs souvent "k parmi n" pour signifier $\binom{n}{k}$. On appelle parfois un choix de k éléments une *combinaison*. Il est bien connu que $\binom{n}{k} = \frac{n!}{k!(n-k)!}$. Ceci est à démontrer en exercice.

On appelle $\binom{n}{k}$ le *coefficient binomial*. Ceci est parce que $\binom{n}{k}$ est le coefficient du terme en x^k lorsqu'on fait l'expansion de $(x+1)^n$. Vous devriez prendre le temps de vous demander pourquoi.

1.6 Techniques de preuve

Nous serons souvent amenés à devoir prouver une affirmation p (i.e. un prédicat), et la plupart du temps nous utiliserons des techniques de preuve qui entrent dans une des catégories suivantes. En algorithmique, les preuves ne nécessitent généralement pas de calculs difficiles. Un niveau élémentaire d'outils mathématiques est souvent très suffisant pour démontrer des résultats algorithmiques. La complexité de ces preuves réside souvent dans l'ingéniosité requise pour trouver les bons calculs à faire, ou la bonne suite de raisonnements qui démontre le résultat. C'est d'ailleurs ce qui fait la beauté de cette discipline !

Notons qu'en algorithmique, les preuves sont destinées à être lues par des personnes, et non des machines. Les preuves sont donc souvent en forme de texte - avec des phrases exprimant la suite de raisonnements qui démontre le résultat voulu. Ceci n'enlève rien à la validité de la preuve. Par contre, lors d'une démonstration, il faut s'assurer de :

- ne pas être au niveau informel ou intuitif. L'argumentation doit démontrer hors de tout doute que l'énoncé à démontrer est vrai ;
- ne pas faire des bonds logiques trop grands ;
- ne pas laisser de cas intraité. Si on oublie de traiter des cas, ce sont souvent ceux qui brisent la démonstration ;

Il faut aussi se méfier des phrases qui commencent par "il est facile de voir que...". Même des docteurs en mathématiques ont commis des erreurs en utilisant

Pour chaque type de preuve, nous donnons quelques exemples canoniques. L'approfondissement de ces techniques devra se faire avec les exercices.

Preuve directe

C'est une argumentation faite à l'aide d'une suite de raisonnements logiques démontrant la véracité de notre affirmation. On pourrait dire en fait dire la même chose de tout type de preuve, mais dans une preuve directe on n'utilise aucune technique ni supposition spéciale, que des arguments directs.

L'exemple canonique des preuves directes est le suivant.

Proposition 1. *La somme de deux entiers pairs est paire.*

Démonstration. Soient n et m deux entiers pairs. Il existe alors deux entiers a et b tels que $n = 2a$ et $m = 2b$. On a que $n + m = 2a + 2b = 2(a + b)$,

et puisque multiplier un entier par 2 le rend pair, on déduit que $n + m$ est pair. □

Voici un autre exemple simple de preuve directe.

Proposition 2. *Pour tout entier* $n \geq 2$, $\binom{n}{2} = \frac{n(n-1)}{2}$.

Démonstration. Rappelons que la quantité $\binom{n}{2}$ réfère au nombre de façons de choisir 2 éléments distincts dans un ensemble X avec $|X| = n$, sans considérer l'ordre. On a n façons de choisir un premier élément, puis $n-1$ façons de choisir le deuxième. Ceci donne lieu à $n(n-1)$ paires ordonnées du style (a, b), où $a, b \in X$. Puisque nous avons compté chaque sous-ensemble $\{a, b\}$ deux fois avec (a, b) et (b, a), l'expression $\frac{n(n-1)}{2}$ nous donne $\binom{n}{2}$. □

Preuve par contre-exemple

Ce type de preuve s'applique lorsqu'on veut montrer qu'il existe un objet qui ne satisfait *pas* une propriété. La plupart du temps, le contre-exemple sert à démontrer qu'un énoncé qu'on pensait vrai est en fait faux. Pour ce faire, on n'a qu'à trouver un exemple concret qui invalide notre hypothèse (ou du moins, on montre qu'un tel contre-exemple existe, même si on n'arrive pas à en construire un de façon explicite). Il faut bien sûr argumenter que notre exemple est bel et bien un contre-exemple, ce qui n'est pas toujours évident.

Un exemple d'application est de montrer qu'un algorithme ne fonctionne pas : il suffit de trouver une seule instance qui fait échouer l'algorithme.

Proposition 3. *Le tri Sous-gradué tel que présenté en classe ne retourne pas toujours un tableau trié.*

Démonstration. Si on donne l'entrée $[3, 1, 2, 4]$ au tri Sous-gradué, au premier tour de boucle, il échangera les valeurs 3 et 4, résultant en le tableau $[4, 1, 2, 3]$. Puisque *nbSwap* devient 1, la valeur 3 restera à la fin du tableau et ne pourra jamais être échangée. On en déduit que le tri Sous-gradué donne la mauvaise sortie sur $[3, 1, 2, 4]$. □

Preuve par induction

La preuve par induction s'applique sur les énoncés qui doivent être prouvés pour une suite d'entiers (possiblement infinie). Souvent, on veut montrer qu'un prédicat $p(i)$ qui dépend de $i \in \mathbb{N}$ est vrai pour tout $i = 1, 2, 3, \ldots$.

L'idée de l'induction est d'abord de montrer que $p(1)$ est vrai. Ensuite, on utilise cette information pour montrer que $p(2)$ est vrai. Puis on utilise $p(2)$ pour montrer que $p(3)$ est vrai, et ainsi de suite. En général, si on arrive à démontrer que $p(1), \ldots, p(i-1)$ sont vrais, on utilise ces faits pour montrer $p(i)$, et ce pour tout i. La seule exception est le cas $p(1)$ pour lequel on ne disposait d'aucune information pour le prouver.

Donc la technique d'induction va comme suit. On commence par prouver un *cas de base*, ci-haut $p(1)$. Ensuite, au lieu de prouver $p(2), p(3), \ldots$ séparément, on donne un argument général pour tout $i > 1$. On suppose que $p(i-1)$ a été prouvé, et on prouve $p(i)$. On appelle $p(i-1)$ notre *hypothèse d'induction*, et cette technique s'appelle l'induction *faible*. Dans l'induction *forte*, on suppose que $p(1), p(2), \ldots, p(i-1)$ forment notre hypothèse d'induction, et les utilise tous (ou plusieurs) pour prouver $p(i)$. Cette différence n'est pas fondamentale, mais on préfère parfois la simplicité de l'induction faible lorsque possible.

Dans les deux cas, une preuve par induction a toujours deux parties. Le cas de base, et le pas d'induction (appelé ainsi parce cette deuxième partie nous permet d'avancer de $i-1$ à i). Notons qu'il y a parfois plusieurs choix pour le paramètre i à utiliser — c'est au prouveur de déterminer celui qui est approprié. Notons aussi que ci-haut, $i = 1$ formait notre cas de base. On peut parfois choisir un autre i, par exemple si notre énoncé ne s'applique que pour tout $n \geq 2$.

Nous utiliserons beaucoup l'induction en algorithmique. Par exemple, on voudra montrer qu'un énoncé est vrai après un premier tour de boucle (cas de base), puis qu'il l'est aussi après le deuxième tour de boucle, le troisième, et ainsi de suite. L'induction est aussi utile pour déterminer certaines complexités, par exemple dans les récurrences et dans les sommations.

La meilleure façon de se familiariser avec l'induction est de faire des exemples. Voici un exemple canonique — c'est le théorème fondamental de l'arithmétique.

Théorème 1. *Tout entier $n > 1$ est soit un nombre premier, ou bien il peut être écrit comme le produit de nombres premiers (avec répétitions possibles).*

Démonstration. Nous prouvons le théorème par induction forte sur n.
Cas de base : $n = 2$. Dans ce cas, 2 est un nombre premier et satisfait l'énoncé du théorème.
Pas d'induction : on suppose par induction que tout entier entre 2 et $n-1$ est soit un nombre premier, ou bien est un produit de nombres premiers. Considérons maintenant n. Si n est un nombre premier, alors il satisfait notre énoncé. Sinon, n est le produit de deux nombres a et b plus grands que 1, et

donc $n = ab$. Il est clair que $2 \leq a \leq n-1$ et $2 \leq b \leq n-1$. Par l'hypothèse d'induction, il existe des nombres premiers $p_1, \ldots, p_i$ et $q_1, \ldots, q_j$ tels que $a = p_1 p_2 \ldots p_i$ et $b = q_1 q_2 \ldots q_j$ (il est possible que $i = 1$ et/ou $j = 1$, ce qui ne pose pas problème). On déduit que $n = p_1 p_2 \ldots p_i q_1 q_2 \ldots q_j$, ce qui montre que n est le produit de nombres premiers. □

Notons que le vrai théorème fondamental de l'arithmétique dit aussi que la décomposition en nombre premier de tout $n > 1$ est unique. L'unicité demande une autre preuve (que vous trouverez facilement sur le web).

Preuve par contradiction

Dans cette technique, on veut prouver p mais ceci s'avère difficile. On essaie alors de prouver sa négation $\neg p$ pour être certain qu'on ne s'est pas trompé sur p, et on voit que lorsqu'on suppose $\neg p$, on démontre des choses qui sont clairement fausses. On en déduit alors que $\neg p$ est impossible, et donc que p doit être vrai. Une variante de cette stratégie est de supposer $\neg p$ et d'en déduire que p est vrai. Ceci donne que $\neg p \Rightarrow p$, ce qui ne fait aucun sens et donc p est vrai.

En utilisant la technique de preuve par contradiction, il faut s'assurer de bien avoir appliqué la négation sur p (et particulier lorsque p contient plusieurs $\forall, \exists, \vee, \wedge, \ldots$).

L'exemple canonique est la suite du théorème 1, le théorème d'Euclide [2]. C'est un bel exemple de réutilisation de théorèmes prouvés antérieurement.

Théorème 2. *Il y a un nombre infini de nombres premiers.*

Démonstration. Pour fins de preuve par contradiction, supposons que l'énoncé est faux. On suppose donc que le nombre de nombres premiers est fini, et donc il existe un ensemble $P = \{p_1, p_2, \ldots, p_k\}$ contenant tous les nombres premiers.

Considérez l'entier $q = p_1 p_2 \ldots p_k + 1$. On sait que q ne peut pas être un nombre premier puisqu'il est plus grand que tous les éléments de P (et P contient tous les nombres premiers). Donc, par le Théorème 1, q est un produit de nombres premiers. Il y a donc $p_i \in P$ qui divise q. Ceci veut dire que $\frac{p_1 p_2 \ldots p_k + 1}{p_i}$ est un entier. Donc, $\frac{p_1 p_2 \ldots p_k}{p_i} + \frac{1}{p_i}$ est un entier. Il est clair que le terme gauche est un entier (car p_i est dans le numérateur), mais

2. Notons que la preuve originale d'Euclide était directe : elle permettait de trouver des nombres premiers de plus en plus grands. Ici, nous présentons une reformulation par contradiction.

il est impossible que $\frac{1}{p_i}$ soit un entier aussi. Puisque l'addition d'un entier avec un non-entier donne un non-entier, nous déduisons que q n'est pas un entier. Nous avons atteint une contradiction. On en déduit que l'ensemble des nombres premiers ne peut pas être fini. □

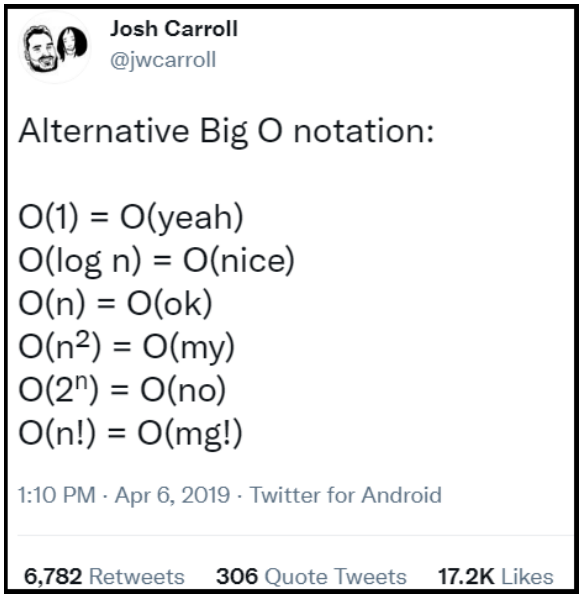
1.7 Résumé

Ce que vous devez savoir :

— les concepts de séquences et d'ensembles, en particulier toutes les notations sur les ensembles
— ce que veulent dire les symboles $\in, \forall, \exists, \wedge, \vee, \neg, \Rightarrow, \Leftrightarrow$. Vous devez aussi savoir appliquer une négation sur un prédicat.
— les propriétés des log et des exposants
— ce que sont les permutations, $n!$ et $\binom{n}{k}$.
— les techniques de preuves standard : directes, par contre-exemple, par induction, par contradiction.
— faites les exercices !

Chapitre 2

La notation asymptotique



La notation grand-O sert principalement à comparer des fonctions et les ordonner en "importance". En fait, O est un ordre partiel sur les fonctions et peut donc servir à nous dire qu'une fonction est "meilleure" une autre (ou pour être plus précis, on devrait dire "pas plus pire"). C'est la raison principale pour laquelle la notation a été inventée. Par exemple, dire que $n \log n \in O(n^2)$ aura l'interprétation "$n \log n$ est meilleure que n^2". Comme nous le verrons, une autre utilité de la notation O est de déterminer le comportement d'une fonction par rapport à un paramètre d'intérêt, souvent n, lorsqu'il prend de grandes valeurs.

Donc à priori, la notation grand-O n'a rien à voir avec l'algorithmique - c'est un concept purement mathématique sur les fonctions. Mais en algorithmique, il s'avère que l'on représente le temps en fonction d'une taille d'entrée. Ces fonctions sont souvent embrouillées par des détails qui contribuent minimalement au temps. On cherche avec la notation O à savoir à quel taux de croissance simplifié notre temps s'apparente "le plus". Plus précisément, on veut voir le ou les termes qui dominent la complexité.

Par exemple, on veut transformer $103n^2 + 76n + 100$ en $O(n^2)$, ou encore $(2^{\log n} + \sqrt{n}) \cdot \frac{1}{8}(\log(n^3) + \log(\log n))$ en $O(n \log n)$ pour rapidement voir le ou les termes qui dominent la complexité. En particulier, on se débarrasse des constantes car leur rôle est de moins en moins important Par exemple, n^2 peut être préférable à $100n$ quand n est petit, mais combien préférez-vous exécuter d'instructions quand $n = 10^7$?

Puisque la notation O est rattachée aux fonctions et non au temps, on pourrait très bien dire que la mémoire utilisée par l'algorithme est $O(n)$, ou bien que le nombre de processeurs utilisés est $O(n^2)$. En fait, on peut placer n'importe quel concept représenté par une fonction quelque part dans la notation O.

2.1 L'ensemble de fonctions $O(f(n))$

Pour entreprendre une couverture un peu plus formelle du sujet, la notation $O(f(n))$ dépend d'une *fonction* $f : \mathbb{N} \to \mathbb{R}^{\geq 0}$ (rappelons que $\mathbb{R}^{\geq 0} = \{x \in \mathbb{R} : x \geq 0\}$). Ici, n n'est pas une quantité spéciale, ni une taille d'entrée, ni un nombre d'imbrications de boucles. On pourrait très bien écrire $O(f(x))$ pour signifier la même chose, la différence étant seulement le nom que l'on donne à la variable de f. Mais que veut dire $O(f(n))$ alors ? La réponse est que $O(f(n))$ désigne l'ensemble des fonctions $g(n)$ qui ne grandissent pas plus vite que $f(n)$.

Donc quand on dit que le temps d'un algorithme A est $O(n^2)$, on veut

dire que le temps pris en fonction de n ne grandit *pas plus vite* que la fonction $f(n) = n^2$. Ceci veut donc dire que O décrit un comportement *maximum*, sans rien indiquer sur son degré de précision. Si votre algorithme s'exécute en temps linéaire, on dit qu'il est $O(n)$ mais il est aussi vrai qu'il est $O(n^{100})$ ou bien $O(2^n)$ car n ne grandit pas plus vite que ces fonctions. Bien sûr, dans les faits on préfère donner une complexité précise.

2.2 Définition formelle de O

Il ne nous reste qu'à définir ce que veut dire "ne grandit pas plus vite" lorsqu'on compare deux fonctions. Intuitivement, $g(n)$ ne grandit pas plus vite que $f(n)$ si, à partir d'un certain n, $g(n) \leq cf(n)$ pour une constante $c \in \mathbb{R}$. Cette constante peut être petite, moyenne, énorme, gargantuesque — ce n'est pas important, mais il faut que ce soit une constante qui ne dépend aucunement du paramètre n.

Nous en arrivons à la définition formelle de $O(f(n))$. Dénotons par $\mathcal{F}$ l'ensemble de toutes les fonctions de $\mathbb{N}$ vers $\mathbb{R}^{\geq 0}$. Pour une fonction $f \in \mathcal{F}$, on a

$$O(f(n)) = \{g \in \mathcal{F} : \exists n_0 \in \mathbb{N}, c \in \mathbb{R}, \forall n \geq n_0, g(n) \leq c \cdot f(n)\}$$

Cette définition formalise précisément l'intuition donnée ci-haut. Remarquons que $O(1)$ désigne l'ensemble des fonctions dont la valeur ne dépasse jamais une constante, peut importe la nature de cette constante.

Avec cette définition, on voit que $O(f)$ est un ensemble de fonctions. On met dans cet ensemble toutes les fonctions g qui, à partir d'un certain n_0, sont toujours plus petites que f à une constante près. Il sera parfois important de s'en remettre à cette définition formelle lorsque des fonctions sont difficiles à placer en termes de O. On appelle O une *notation asymptotique* parce que, d'une certaine façon, elle ne s'intéresse qu'aux comportements des fonctions que lorsqu'elles tendent vers l'infini (ou du moins, toujours pour un n suffisamment grand).

Nous donnons ici quelques exemples de base. Des exemples plus substantiels seront donnés en exercice.

Exemple 1. *Soit $f(n) = n/10$ et $g(n) = 10n$. Montrez que $g(n) \in O(f(n))$.*

Démonstration. On va trouver une constante c telle que $10n \leq cn/10$ pour tout entier $n \geq 1$. Avec un peu d'essai et erreur, on voit que 200 fera

amplement l'affaire. C'est-à-dire, alors $10n < 200 \cdot n/10$ pour tout $n \geq 1$, et donc g ne grandit pas plus vite que f. Il s'ensuit que $g(n) \in O(f(n))$. □

Notons que f et g ne sont que des noms de fonction. Dans l'exemple suivant, on inverse leur rôle.

Exemple 2. *Soit $f(n) = n^2 + 1000n + 5$ et $g(n) = n^2/100$. Montrez que $f(n) \in O(g(n))$.*

Démonstration. On veut éventuellement montrer que $f(n) \leq cg(n)$. Pour ce faire, il est commun de se démarrer une chaîne d'inégalités initiale dans laquelle le terme le plus significatif de f englobe tous les autres. Ceci simplifie les calculs. On note que pour tout $n > 5$, on a

$$f(n) = n^2 + 1000n + 5 < n^2 + 1000n^2 + n^2 = 1002n^2$$

On voit que si on multiplie $g(n)$ par 100201, il dépassera $f(n)$. C'est-à-dire,

$$f(n) < 1002n^2 < 100201n^2/100 = 100201g(n)$$

On a donc démontré que $f(n) < cg(n)$ pour tout entier $n > 5$, avec $c = 100201$. Donc f ne grandit pas plus vite que g et $f(n) \in O(g(n))$. □

Notez que dans les deux exemples précédents, $f(n) \in O(g(n))$ et $g(n) \in O(f(n))$.

Exemple 3. *Soient $f(n) = n^2/1000$ et $g(n) = 1.0001^n$. Montrez que $f(n) \in O(g(n))$.*

Démonstration. Il est vrai que $g(n) = 1.0001^n < n^2/1000$ pour $n \in \{1, 2, \ldots, 172041\}$, mais $1.0001^n > n^2/1000$ pour tout entier $n > 172041$. Donc éventuellement à partir d'un certain n, $n^2/1000 < 1.0001^n$ dépasse $n^2/1000$ et grandit donc plus vite, et $f(n) \in O(g(n))$. □

Dans les exemples, nous avons vu la méthode la plus courante pour prouver que $f(n) \in O(g(n))$: on trouve une constante c et un n_0 de départ pour lesquels la définition ci-haut s'applique. Une autre méthode passe par les limites de fonctions (on suppose que le concept de limite est connu nous ne reverrons pas les limites ici). Nous allons parfois appliquer les règles suivantes, et leur preuve est laissée en exercice.

— si $\lim_{n\to\infty} \dfrac{f(n)}{g(n)} = c$, où $c \in \mathbb{R}^{>0}$, alors $f(n) \in O(g(n))$ et $g(n) \in O(f(n))$ (personne ne domine) ;

— si $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$, alors $f(n) \in O(g(n))$ et $g(n) \notin O(f(n))$ (g domine f) ;
— si $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$, alors $f(n) \notin O(g(n))$ et $g(n) \in O(f(n))$ (f domine g) ;

Cette technique prend tout son sens avec la règle de l'Hôpital, qui dit que $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{f'(n)}{g'(n)}$ (dans la mesure où les limites existent). Voici un exemple d'application sans lequel la preuve est un peu plus complexe (voir exercices).

Exemple 4. *Montrez que* $\log n \in O(\sqrt{n})$.

Démonstration. Tout le monde se rappelle que la dérivée de $\log n$ est $1/(n \ln(2))$ et celle de $\sqrt{n}$ est $1/(2\sqrt{n})$. On calcule

$$\lim_{n\to\infty} \frac{\log n}{\sqrt{n}} = \lim_{n\to\infty} \frac{\ln(2)/n}{1/(2\sqrt{n})} = \lim_{n\to\infty} \frac{2\ln(2)}{\sqrt{n}} = 0$$

Selon les règles, $\log n \in O(\sqrt{n})$. □

Notez que la preuve ci-haut montre aussi que $\sqrt{n} \notin O(\log n)$, ce qui veut dire que $\sqrt{n}$ croît strictement plus vite que $\log n$. Donc si avez le choix entre un algorithme en temps $O(n \log n)$ ou un autre en temps $O(n\sqrt{n})$, vous savez lequel choisir.

Une autre façon de démontrer l'appartenance avec un O est d'y aller par contradiction : on suppose que $g(n) \notin O(f(n))$, on établit ce que ceci impliquerait et on dérive une contradiction. Considérez l'énoncé suivant, qui dit que l'on devrait toujours préférer un algorithme polynomial à un exponentiel. Par exemple, on préfère n^{10000} à 1.000001^n.

Théorème 3. *Pour toutes constantes* $a \in \mathbb{R}, b \in \mathbb{R}^{>1}$, $n^a \in O(b^n)$.

Démonstration. Supposons, pour fins de contradiction, que l'énoncé est faux et que $n^a \notin O(b^n)$. Ceci veut dire que

$$\neg((\exists n_0 \in \mathbb{N}, c \in \mathbb{R}^{>0})(\forall n \in \{n_0, n_0+1, \ldots\}), n^a \leq cb^n)$$

Donc,

$$(\forall n_0 \in \mathbb{N}, c \in \mathbb{R}^{>0})(\exists n \in \{n_0, n_0+1, \ldots\}), n^a > cb^n$$

Dit autrement, on démarre avec la supposition :

$$n^a > cb^n$$

pour n'importe quelle constante c et pour une infinité de valeurs de n. En particulier, ceci est supposé vrai pour $c = 1$. Ceci veut donc dire que

$$\begin{aligned} & n^a > b^n \\ \Rightarrow & \log_b(n^a) > \log_b(b^n) \\ \Rightarrow & a \log_b n > n \log_b b \\ \Rightarrow & a \log_b n > n \\ \Rightarrow & a > \frac{n}{\log_b n} \end{aligned}$$

Ceci doit être vrai pour une infinité de valeurs de n. Toutefois, a est une constante et $\frac{n}{\log_b n}$ tend vers l'infini (parce que $b > 1$). Il est donc impossible que $a > \frac{n}{\log_b n}$ pour une infinité de valeurs de n, ce qui est une contradiction. On peut donc déduire que notre supposition était fausse, et donc que $n^a \in O(b^n)$.

□

2.3 Notation asymptotique multivariée

Comment analysez-vous le temps d'un algorithme qui a une boucle de 1 à n, puis une autre boucle de 1 à m ? Si $n < m$, c'est $O(m)$ mais si $m > n$, c'est $O(n)$. Ou bien on peut étendre la notation O à des fonctions multivariées. Dans ce cas, on dira que $f(n, m) \in O(g(n, m))$ si, pour un n_0 assez grand et un m_0 assez grand, $f(n, m) < cg(n, m)$ pour tout $n \geq n_0$ et $m \geq m_0$. Donc l'intuition reste la même — on simplement deux variables. Formellement, si $\mathcal{F}_2$ désigne l'ensemble des fonctions de $\mathbb{N} \times \mathbb{N}$ vers $\mathbb{R}$, on a $O(f(n, m)) =$

$$\{g \in \mathcal{F}_2 : (\exists n_0, m_0 \in \mathbb{N}, c \in \mathbb{R}^{>0})(\forall n \geq n_0, m \geq m_0) g(n, m) \leq cf(m, n)\}$$

On peut étendre cette définition à un nombre arbitraire de variables. Vous pouvez imaginer que ça ne sera ce qu'il y a de plus élégant. L'important est garder en tête l'intuition habituelle que $f(x_1, \ldots, x_p) \in O(g(x_1, \ldots, x_p))$ si f ne grandit pas plus vite que g à partir d'un certain point, et ce pour

toute combinaison des x_i.

2.4 La notation Ω

La notation O exprime le fait qu'un algorithme ne prendra *jamais plus* qu'une certaine quantité de ressources, souvent le temps. La notation O est donc une sorte de maximum. On veut parfois exprimer le fait que l'algorithme prendra *toujours au moins* une certaine quantité de ressource, i.e. un minimum. On utilise la notation Ω pour ce faire. C'est-à-dire, $f(n) \in \Omega(g(n))$ si, à partir d'un certain n_0, $f(n) > cg(n)$ pour une constante c. Formellement,

$$\Omega(f(n)) = \{g \in \mathcal{F} : \exists n_0 \in \mathbb{R}, c \in \mathbb{R}^{>0}, \forall n \geq n_0, g(n) \geq cf(n)\}$$

Notons que ici, $c > 0$ mais peut être très près de 0. La notation Ω est un peu l'inverse du O, tel que l'on démontre ici. Observez que le théorème suivant demande de démontrer un "si et seulement si", i.e. une équivalence avec le $\Leftrightarrow$. En nous rappelant que $p \Leftrightarrow q$ veut dire $p \Rightarrow q \wedge q \Rightarrow p$, il faut prouver deux implications pour démontrer une équivalence.

Théorème 4. *Soient $f, g \in \mathcal{F}$. Alors $f(n) \in O(g(n))$ si et seulement si $g(n) \in \Omega(f(n))$.*

Démonstration. Montrons d'abord que $f(n) \in O(g(n)) \Rightarrow g(n) \in \Omega(f(n))$. Puisque $f(n) \in O(g(n))$, il existe n_0 et c tels que $f(n) \leq cg(n)$ pour tout entier $n \geq n_0$. Donc, $g(n) \geq \frac{1}{c} f(n)$ pour tout entier $n \geq n_0$. En utilisant $\frac{1}{c}$ comme constante, on voit que $g(n) \in \Omega(f(n))$.

Dans l'autre sens, on veut montrer que $g(n) \in \Omega(f(n)) \Rightarrow f(n) \in O(g(n))$. La preuve est presque identique. On sait qu'il existe n_0 et c tels que $g(n) \geq cf(n)$ pour tout entier $n \geq n_0$. Donc $f(n) \leq \frac{1}{c} g(n)$ et il s'ensuit que $f(n) \in O(g(n))$. □

2.5 La notation Θ

Rappelons qu'un algorithme en temps linéaire est $O(n)$, mais aussi $O(2^n)$ puisque le O n'est qu'un maximum. Si on désire aussi indiquer que l'algorithme est $\Omega(n)$, on écrit qu'il prend un temps $\Theta(n)$. Nous préférons utiliser Θ lorsque c'est possible, puisqu'il donne une borne précise du temps pris dans le pire cas au lieu d'un simple maximum ou minimum. L'ensemble des fonctions qui entrent dans θ est

$$\Theta(f(n)) = O(f(n)) \cap O(f(n))$$

Une définition équivalente à l'image des autres données ci-dessus est

$$\Theta(f(n)) = \{g \in \mathcal{F} : \exists n_0 \in \mathbb{R}, c, d \in \mathbb{R}^{>0} \forall n \geq n_0, cf(n) \leq g(n) \leq df(n)\}$$

Donc la notation Θ englobe bien les notions O et Ω.

2.6 Fonctions lisses (* optionnel)

Dans les exercices introductifs, nous avons montré que le tri fusion prenait un temps $O(n \log n)$, mais seulement lorsque n est une puissance de 2. Comment prouver que c'est $O(n \log n)$ pour tout n ? L'induction que l'on a utilisée n'admet pas d'argument simple pour les n entre es puissances de 2. Un outil intéressant pour ce genre de problème consiste en les fonctions *lisses* (*smooth* en anglais). On dit qu'une fonction f est lisse s'il existe $n_0, b \in \mathbb{N}$ tels que pour tout entier $n \geq n_0$, on a les deux conditions

— $f(n) \leq f(n+1)$ [f est éventuellement non-décroissante]

— $f(bn) \in O(f(n))$ [un facteur b ne change rien au taux de croissance]

En particulier, le choix $b = 2$ fera souvent l'affaire. Beaucoup de fonctions que nous rencontrerons sont lisses, par exemple $n^c, \log n, n \log n$. Certaines fonctions qui grandissent trop vite ne le sont pas, comme 2^n ou $n!$.

L'intérêt de cette propriété est la *règle de la lisseté* (traduit de *rule of smoothness*), qui dit la chose suivante :

Théorème 5. *Si $f(n)$ est une fonction lisse et $f(n) \in O(g(n))$ seulement quand n est une puissance de 2, alors $f(n) \in O(g(n))$ de façon inconditionnelle.*

Nous n'avons pas défini ce que voulait dire "$f(n) \in O(g(n))$ seulement quand XYZ". Nous nous en remettons à l'intuition pour cette partie. La règle ci-haut nous économisera beaucoup d'effort, surtout quand nous étudierons les fonctions récursives. Consultez le livre de Brassard et Bratley (Chapitre 3) pour une preuve de cette règle.

Exemple 5. *La fonction $n \log n$ est lisse.*

Démonstration. Posons $f(n) = n \log n$. On montre que $f(2n) = (2n)\log(2n) \in O(n \log n)$. On a que $log(2n) \leq 2 \log n$ pour $n > 4$. Donc $f(2n) < 2n \cdot 2 \log n = 4n \log n \in O(n \log n)$. □

Nous avons donc démontré que le tri fusion prend un temps $O(n \log n)$ de façon inconditionnelle.

2.7 Comment montrer que $f(n) \notin O(g(n))$?

Les règles des limites sont très utiles pour démontrer qu'une fonction n'est pas dans le O d'une autre. Lorsque cette règle n'est pas applicable ou difficile à utiliser, une technique commune est la preuve par contradiction. On suppose que $f(n) \in O(g(n))$ et on dérive quelque chose de faux. Voici un exemple.

Exemple 6. $4^n \notin O(2^n)$

Démonstration. Supposons pour fins de contradiction que $4^n \in O(2^n)$. Alors il existe $n_0 \in \mathbb{N}$ et $c \in \mathbb{R}^{>0}$ tels que $4^n \leq c2^n$ pour tout $n \geq n_0$. En isolant c, ceci veut dire que $\frac{4^n}{2^n} \leq c$ et donc que $2^n \leq c$. Ceci veut dire que c tend vers l'infini lorsque n grandit, ce qui est impossible puisque c est une constante. Cette contradiction implique que $4^n \notin O(2^n)$. □

Dans les exercices, on analyse le cas où on remplace 2 et 4 par n'importe quelles constantes c et $(c + \epsilon)$, où $c > 1$ et $\epsilon > 0$.

2.8 Résumé

Ce que vous devez savoir :

— Comprenez l'intuition derrière O, Ω et Θ, et pourquoi c'est utile en algorithmique.

— Sachez appliquer les définitions correctement. Fixez d'abord n_0 et c, puis ensuite montrez que $f(n) \leq c(n)$ pour tout $n \geq n_0$. Une erreur commune est de changer le c en cours de route (voir exercices).

— Sachez montrer qu'une fonction est O, Ω ou Θ d'une autre fonction ou pas. *****

— Faites les exercices.

***En bref** : Pour montrer qu'une fonction est O, Ω ou Θ d'une autre fonction

$f(n) \in O(g(n))$ prouver $f(n) \leq c(g(n))$

$f(n) \in \Omega(g(n))$ prouver $f(n) \geq c(g(n))$

$f(n) \in \Theta(g(n))$ prouver $c_1(g(n)) \leq f(n) \leq c_2(g(n))$

Chapitre 3

Le tri

En tant que première application de nos nouvelles connaissances en analyse asymptotique, nous allons analyser quelques algorithmes de tri classiques. Commençons par nous rappeler du problème du tri :

Problème du tri (en ordre croissant)
Entrée : une séquence $T = (t_1, \ldots, t_n)$ d'éléments comparables (avec "$\leq$").
Sortie : une permutation $T' = (t'_1, t'_2, \ldots, t'_n)$ des éléments de T telle que $t'_1 \leq t'_2 \leq \ldots \leq t'_n$.

Rappelons qu'une permutation de T consiste en un réordonnancement de ses éléments. Pour nous rapprocher de la notation informatique habituelle, nous allons dénoter un tableau par $T = [t_1, \ldots, t_n]$ (on utilise les crochets au lieu des parenthèses).

3.1 Le tri insertion

Nous connaissons déjà le tri par insertion et le tri fusion. Commençons d'abord par revoir le tri par insertion :

```
fonction triInsertion(T) //T est de taille n
    pour j = 2..n faire
        val = T[j]
        i = j − 1
        tant que i > 0 et T[i] > val faire
            T[i + 1] = T[i]
            i = i − 1
        fin
        T[i + 1] = val
    fin
```

Analysons la complexité de cet algorithme, dans le pire cas. On voudrait idéalement placer cette complexité avec la notation Θ pour bien la cerner. Peut-on d'abord classer son temps en O ? Ceci demande de déterminer le temps *maximum* que peut prendre cet algorithme. On peut démontrer que c'est $O(n^2)$.

Proposition 4. *Le tri par insertion s'exécute en temps $O(n^2)$.*

Démonstration. Pour chaque j de 2 à n, l'algorithme exécute un nombre c constant d'instructions, puis entre dans une boucle imbriquée allant de $j-1$ à 0 dans le pire cas, chaque tour exécutant un autre nombre d constant d'instructions. Pour une valeur de j, on répète ces d instructions au maximum

j fois. Donc pour $j = 2$, on exécute au maximum $c + 2d$ instructions. Pour $j = 3$, on en exécute au maximum $c + 3d$, et en général pour n'importe quel j, on exécute au maximum $c + jd$ instructions.

Le nombre d'instructions exécutées au total sera donc, au maximum

$$\begin{aligned}
\sum_{j=2}^{n}(c + jd) &= \sum_{j=2}^{n} c + \sum_{j=2}^{n} jd \\
&= c(n-1) + d\sum_{j=2}^{n} j \\
&\leq cn + d\sum_{j=1}^{n} j \\
&= cn + d\frac{n(n+1)}{2} \\
&= cn + dn^2/2 + dn/2 \\
&\in O(n^2)
\end{aligned}$$

□

Si certaines opérations sur les sommations faites ci-haut ne vous apparaissent pas comme évidentes, vous devriez prendre un moment pour vous en convaincre.

Cette preuve est beaucoup plus détaillée que ce qui est habituellement fait en algorithmique. La plupart du temps, on ne se soucie pas des blocs d'instructions constants et on argumente plutôt sur la structure du programme. Une argumentation plus concise dirait : "on fait au plus n tours de la première boucle et, pour le j-ième tour de boucle, on exécute une boucle imbriquée avec au plus j tours. Le nombre d'instructions est donc proportionnel (donc à une constante près) à $1+2+3+\ldots+n = n(n+1)/2 \in O(n^2)$".

Notons que nous avons seulement donné une borne maximale sur le nombre d'instructions. La preuve ci-haut n'exclut pas la possibilité que l'algorithme prenne un temps $O(n)$ ou bien $O(n \log n)$. Il faut démontrer que le temps est aussi $\Omega(n^2)$. La plupart du temps, ceci est fait en démontrant que pour une infinité de n, il existe un exemple qui nécessite ce temps.

Proposition 5. *Le tri par insertion s'exécute en temps $\Omega(n^2)$ dans le pire cas.*

Démonstration. Supposons que l'entrée donnée est $T = [n, n-1, \ldots, 2, 1]$.

Pour chaque tour de boucle j, on mettra $val = T[j]$ et il est facile de voir qu'à ce point, $T[j] < T[1], T[2], \ldots, T[j-1]$. La boucle imbriquée fera donc $j-1$ décalages, et ce pour tous les $j \in \{2, \ldots, n\}$. On atteint donc le nombre d'instructions analysé dans la preuve du $O(n^2)$, et on peut déduire que le temps requis est $\Omega(n^2)$. □

Invariants de boucle. Mais au fait, comment savons-nous que le tri par insertion est correct ? Est-il possible que, comme le tri sous-gradué (voir slides d'introduction), cet algorithme se trompe sur certaines entrées ? Nous allons que démontrer que la réponse est non à l'aide des **invariants de boucle**.

Théorème 6. *Le tri par insertion résout le problème du tri.*

Démonstration. Considérez la boucle principale de l'algorithme, qui effectue $n-1$ tours pour i de 2 à n. Nous allons démontrer par induction sur le nombre de tours de boucles effectués l'énoncé suivant : pour tout $j \in \{2, 3, \ldots, n\}$, à la fin du tour de boucle j, le sous-tableau $T[1..j]$ est trié. Puisque, en particulier, cet énoncé est vrai pour $j = n$, ceci démontre qu'à la fin de l'exécution, le tableau sera trié.
Cas de base : après le tour de boucle avec $i = 2$, il est évident que $T[1..2]$ est trié : si on avait $T[1] > T[2]$ au départ, les deux valeurs seront échangées, et rien ne sera échangé sinon.
Induction : on suppose par induction qu'après le tour de boucle avec $j-1$, le sous-tableau $T[1..j-1]$ est trié. Pendant l'itération avec la valeur j, la sous-boucle décale les éléments d'un sous-tableau $T[k..j-1]$ vers la droite, pour un certain k (qui est en fait la valeur de $i+1$ à la fin de la boucle imbriquée). Ceci est fait de façon à ce que $T[k-1] < T[j]$, et avec toutes les valeurs de $T[k..j-1]$ qui sont plus grandes que $T[j]$. Il est donc clair que lorsqu'on fait $T[i+1] = val$, on a $i+1 = k$ et le tableau $T[1..j]$ sera trié, ce qu'on voulait démontrer. □

Notez que pour être stricts, on aurait dû créer un deuxième invariant de boucle pour la boucle imbriquée, et trouver un deuxième énoncé formel à démontrer. Par contre, dans ce cas-ci, on juge que de laisser cette partie informelle est convenable, puisque c'est suffisant pour nous convaincre nous-mêmes ou notre lecteur que l'algorithme est correct. Nous verrons qu'au fur et à mesure que nos algorithmes se compliquent, nous devons nous en tenir aux éléments clés pour démontrer qu'il est correct, sans quoi nos preuves prendraient toutes une trentaine de pages.

Une propriété démontrée par induction sur le nombre de tours de boucle est ce qu'on appelle un *invariant de boucle*. L'idée de l'invariant de boucle est d'identifier une propriété qui : (1) est vraie au premier tour de boucle ; (2) est maintenue tout au long de l'algorithme ; et (3) démontre que l'algorithme est correct. Notre preuve peut s'appliquer avant ou après le tour de boucle - c'est à nous de décider selon le cas. Comme dans l'exemple ci-haut, on veut souvent trouver un invariant qui implique une progression vers une solution à chaque tour de boucle (par exemple, notre tableau est "de plus en plus trié"). Parfois, notre invariant sera par exemple "chaque tour de boucle ne brise pas de solution optimale", ou encore "chaque tour de boucle considère une possibilité de plus".

3.2 Le tri fusion

Rappelons-nous du tri fusion.

fonction *triFusion(T) //T est de taille n*
  si $|T| = 1$ **alors**
    return T
  fin
  $A = triFusion(1..\lfloor n/2 \rfloor)$
  $B = triFusion(\lfloor n/2 \rfloor + 1..n)$
  $a = 1$
  $b = 1$
  $S = [\,]$
  tant que $a \leq |A|$ *ou* $b \leq |B|$ **faire**
    si $b > |B|$ *ou* $A[a] \leq B[b]$ **alors**
      $S.append(A[a])$
      $a++$
    sinon
      $S.append(B[b])$
      $b++$
  fin
  return S

Attardons-nous d'abord au temps du tri fusion. On note la chose suivante : il n'y a pas de pire ou de meilleur cas. Peu importe le tableau T de taille n reçu en entrée, il sera divisé en deux, deux tris récursifs seront appelés et la fusion opèrera sur deux tableaux de même taille en visitant chaque élément des deux tableaux. Ceci veut dire que si on analyse le temps

de n'importe quel cas, on aura à la fois les bornes O et Ω.

On peut exprimer le temps requis avec une récurrence. Il est facile de se convaincre qu'un appel exécutera un nombre d'instructions proportionnel à n pour diviser T et faire la fusion. On borne ce nombre d'instructions par cn, où c est une constante. Pour le temps requis pour les appels récursifs, on observe qu'un appel récursif recevra un tableau de taille $\lfloor n/2 \rfloor$ et l'autre de taille $\lceil n/2 \rceil$. On peut prendre pour acquis que dans le cas spécial où $n = 1$, on n'exécute qu'une instruction. Soit $t(n)$ le temps requis pour trier un tableau de taille n. On peut écrire

$$t(n) = \begin{cases} 1 & \text{si } n = 1 \\ t(\lfloor n/2 \rfloor) + t(\lceil n/2 \rceil) + cn & \text{sinon} \end{cases}$$

Il est embêtant d'analyser des temps avec des fonctions plancher/plafond. On se simplifiera la vie en supposant que n est une puissance de 2 et en utilisant la "lisseté" de la section 2. On n'a plus à se soucier des planchers et plafonds, car les deux appels reçoivent un tableau de taille $n/2$ à chaque fois. Dans le cas où $n = 2^k$, on a donc

$$t(n) = \begin{cases} 1 & \text{si } n = 2^0 \\ 2t(n/2) + cn & \text{si } n = 2^k, k \in \mathbb{N}^{>0} \end{cases}$$

Ceci ressemble à l'exemple fait en classe et dans les exercices, à l'exception que l'on a cn au lieu de n. On peut tout de même faire une preuve par induction aisément.

Proposition 6. *La fonction $t(n)$ telle que définie ci-haut sur n étant une puissance de 2 est égale à $cn \log n + n$.*

Démonstration. On procède par induction sur k, où $n = 2^k$.

Cas de base : $k = 0$. Dans ce cas, $n = 1$ et $cn \log n + n = c \cdot 1 \log 1 + 1 = 0 + 1 = t(1)$, ce qui vérifie le cas de base.

Induction : lorsque $n = 2^k$, on suppose que $t(2^{k-1}) = c2^{k-1} \log 2^{k-1} + 2^k = c(n/2) \log(n/2) + n/2$. Rappelons que $n/2 = 2^{k-1}$. Par la définition de $t(n)$, on a $t(n) = 2t(n/2) + cn$ et, par induction,

$$\begin{aligned} t(n) &= 2t(n/2) + cn \\ &= 2(c(n/2)\log(n/2) + n/2) + cn \\ &= cn\log(n/2) + n + cn \\ &= cn(\log n - 1) + n + cn \\ &= cn(\log n - 1 + 1) + n \\ &= cn\log n + n \end{aligned}$$

ce qu'on voulait démontrer. □

Comment aurait-on pu "deviner" que $t(n) = cn\log n + n$ sans le savoir d'avance ? Ce n'est peut-être pas ce que vous voulez entendre, mais la réponse la plus simple est "par essai et erreur". L'expression a été trouvée en démarrant une preuve par induction sans connaître l'expression recherchée, et les termes sont apparus pour faire fonctionner la preuve. Nous verrons plus tard le théorème maître, qui nous épargne ce travail. En combinant la preuve ci-haut avec les faits que $cn\log n + n \in O(n\log n)$ et que $n\log n$ est une fonction lisse, on obtient le théorème suivant.

Théorème 7. *Le tri fusion s'exécute en temps* $\Theta(n\log n)$.

Maintenant, comment montrer que le tri fusion trie correctement ? Nous n'irons pas dans les détails, mais on peut procéder par induction sur n. Le cas de base $n = 1$ est trivial. Pour l'induction, on peut supposer par induction que A et B sont triés une fois les appels récursifs faits. Ensuite, il faut montrer que la boucle de fusion trie correctement. C'est relativement évident et on s'attarde rarement à ce genre de preuve. Mais si vous voulez démontrer que la fusion fonctionne, vous pouvez le faire avec l'invariant de boucle "après chaque tour de boucle de la fusion, S est trié et toutes ses valeurs sont plus petites que celles de A et de B". Il est facile de voir que cette propriété est maintenue après chaque ajout en fin de S.

Algorithmes de tri surplace. Notez une différence avec le tri par insertion. La fonction triInsertion modifie le tableau T directement et le retourne. Par contre, le tri fusion créé un nouveau tableau temporaire S, le remplit et le retourne. Bien que ça n'ait pas d'impact sur la complexité théorique en temps, on préfère souvent modifier T directement, puisque cela ne demande pas de nouvelle mémoire. En fait, on dit que le triInsertion utilise $O(1)$ mémoire supplémentaire (sans compter T), car la fonction n'uti-

lise qu'un nombre constant de variables. Le tri fusion utilise $\Omega(n)$ mémoire supplémentaire.

Définition 1. *Un algorithme de tri qui déplace les éléments directement dans le tableau d'entrée T, sans créer de tableau temporaire, est appelé un algorithme* surplace.

3.3 Quicksort

Nous présentons maintenant l'algorithme Quicksort, un algorithme très populaire en pratique malgré sa complexité de $O(n^2)$ dans le pire cas. L'idée est la suivante : on choisit une valeur arbitraire de T, disons $T[1]$, que l'on appelle le *pivot*. On créé un tableau A avec tous les éléments plus petits que le pivot, puis on trie A récursivement. Puis on créé un tableau B avec tous les éléments plus grands que le pivot, puis on trie B récursivement. Une fois que c'est fait, on sait que la concaténation de A, du pivot, puis de B donnera un tableau trié.

fonction *quickSort(T) //T est de taille n*
  si $n == 1$ **alors**
    return T
  $pivot = T[1]$
  $A = [\,]$
  $B = [\,]$
  pour $i = 2..n$ **faire**
    si $T[i] \leq pivot$ **alors**
      $A.append(T[i])$
    sinon si $T[i] > pivot$ **alors**
      $B.append(T[i])$
  fin
  $A' = quickSort(A)$
  $B' = quickSort(B)$
  return $A' + [pivot] + B'$ //où "+" dénote la concaténation

Une variante de Quicksort choisit une position de T au hasard et utilise sa valeur comme pivot. L'analyse probabiliste de cette variante montre que le temps espéré de l'algorithme est $O(n \log n)$. Vous verrons cette analyse plus tard. Notons aussi qu'il existe des versions surplace du Quicksort (ce sont celles qui sont généralement implémentées).

Pire cas du Quicksort. Quelle est la complexité de Quicksort, dans le pire cas ? Notons d'abord qu'il faut un temps $\Theta(n)$ pour partitionner le tableau en A et B autour du pivot. Si nous sommes chanceux à chaque choix de pivot, on divisera toujours le tableau en deux parties égales et on retrouvera la même complexité que le tri fusion, qui est $O(n \log n)$.

Intuitivement, c'est ce qu'on peut espérer de mieux. Toujours selon l'intuition, le pire cas devrait alors être lorsqu'il n'y a aucune division. Considérons le cas où $T = [n, n-1, \ldots, 2, 1]$. Au premier appel, on dépense au moins n instructions pour partitionner, le pivot sera n et on aura $A = [\ ]$ et $B = [n-1, n-2, \ldots, 2, 1]$. L'appel récursif sur B prendra au moins $n-1$ instructions pour partionner, et ainsi de suite. On voit qu'au k-ième appel récursif, on reçoit un tableau de taille $n-k$ et qu'il faut un temps $n-k$ pour le partitionner. Le nombre d'instructions total est donc au moins $n+n-1+n-2+\ldots+2+1 \in \Theta(n^2)$. Ceci démontre que dans le pire cas, *quicksort* prend un temps $\Omega(n^2)$.

Peut-on se convaincre que *quickSort* s'exécute aussi en temps $O(n^2)$? C'est-à-dire, on a montré une instance qui nécessite au moins n^2 instructions (à une constante près), ansi peut-on montrer que le temps ne sera jamais plus que n^2 ? L'intuition nous dit que oui, car on a argumenté que le pire cas se résolvait en temps $O(n^2)$ ci-haut. Mais peut-on être certain que c'est le pire cas ? Pour bien répondre à cette question, nous devrons parfaire nos compétences en analyse de récurrences. Nous approfondirons cette question plus tard.

3.4 Borne inférieure du tri par comparaison [section optionnelle]

Un algorithme de tri par comparaison détermine l'ordre des éléments en faisant des comparaisons entre paires d'éléments. En d'autres termes, ces algorithmes émettent des requêtes du style "est-ce que $a < b$?" et prennent des décisions selon la réponse. Par exemple, le tri par insertion procède par comparaison : lors de la boucle d'insertion (la boucle imbriquée), on demande toujours "est-ce que $T[i] < T[i+1]$?" et on déplace des éléments tant que c'est le cas. La plupart des algorithmes de tri sont de ce type, mais nous en verrons d'autres dans la section suivante.

Dans cette section, nous allons montrer que *n'importe quel* tri par comparaison doit prendre un temps $\Omega(n \log n)$. Ceci veut donc dire que les tris connus qui s'exécutent en temps $O(n \log n)$ sont optimaux. Ceci est plutôt remarquable : malgré des années de recherche sur le sujet, le tri est un des

seuls problèmes pour lesquels une borne inférieure de temps non-triviale est connue.

La difficulté de prouver une borne inférieure sur le temps requis est que la preuve doit couvrir tous les algorithmes possibles (et il y en a beaucoup !).

Nous aurons besoin du résultat suivant.

Lemme 1. $\log(n!) \in \Theta(n \log n)$

La preuve se trouve dans le solutionnaire des exercices, série 2.

Rappelons qu'un arbre enraciné est une structure de données contenant des *noeuds*. Chaque noeud a 0, 1 ou plusieurs enfants et chaque noeud a un (et un seul) parent, à l'exception d'un noeud spécial appelé la racine qui n'a pas de parent. Les noeuds ayant 0 enfants sont appelés les *feuilles*, et les autres noeuds sont des noeuds *internes*. La *hauteur* d'un arbre est le nombre maximum de noeuds entre la racine et une feuille (excluant la racine, incluant la feuille). Un *arbre binaire plein* est un arbre dans lequel chaque noeud a 0 ou 2 enfants. Nous aurons aussi besoin du résultat suivant.

Lemme 2. *Un arbre binaire plein avec n feuilles a une hauteur au moins* $\log n$.

Un *arbre de décisions* représente l'exécution d'un algorithme selon les branchements qu'il effectue sur n'importe quelle entrée de taille n. Dans un algorithme, un prédicat p peut-être évalué et l'algorithme prendra deux directions possibles selon le cas où p est vrai, ou faux (ce qu'on appelle un branchement). Par exemple, une instruction "$if(x > y)$" entrera dans le bloc du "if" si le prédicat "$x > y$" évalue à vrai, et sautera par dessus le bloc sinon. Le principe est le même pour une boucle "$while(x < y)$".

Dans un arbre de décision, chaque noeud interne représente un état possiblement atteint par l'algorithme juste avant de prendre une décision de branchement selon la valeur d'un prédicat p. La racine représente l'état atteint avant le tout premier branchement. Les deux enfants d'un noeud interne représentent les deux états que l'on peut atteindre après avoir évalué p. Une feuille représente l'état de l'algorithme juste avant qu'il retourne son résultat. Voir la Figure 9.1 et la légende pour un exemple. On note qu'un arbre de décision est un arbre binaire plein. Aussi, la hauteur d'un arbre binaire plein donne une borne inférieure sur le nombre d'instructions que doit exécuter l'algorithme : s'il y a un chemin entre la racine et une feuille avec h noeuds, c'est que l'entrée qui fait passer l'algorithme par ce chemin demande au moins h comparaisons, donc au moins h instructions.

Dans le cas particulier d'un algorithme de tri par comparaison, l'algorithme doit baser ses décisions en évaluant des prédicats du style "$T[i] <$

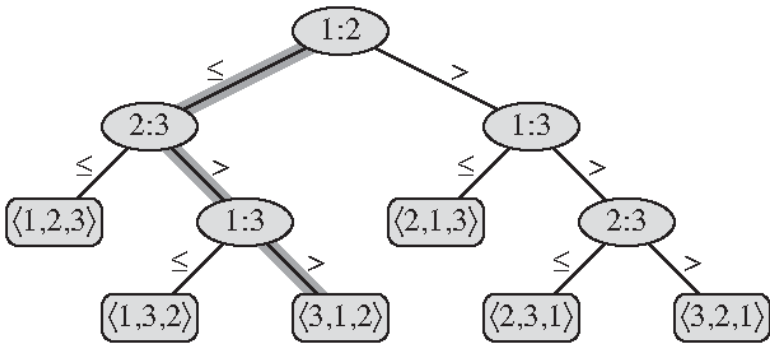


FIGURE 3.1: Un arbre de décision pour le tri avec $n = 3$. Chaque noeud avec des valeurs $a : b$ représente une question "est-ce que $a \leq b$?" et les branches enfantes représentent les deux chemins empruntés selon la réponse.

$T[j]$". On peut donc représenter les noeuds internes d'un arbre de décision par des prédicats de ce genre, et représenter les feuilles par le tableau à retourner (car il fait partie de l'état).

Lemme 3. *Tout algorithme de tri par comparaison nécessite un temps* $\Omega(n \log n)$ *dans le pire cas.*

Démonstration. Soit A un algorithme qui résout le problème du tri correctement, et soit D l'arbre de décision de A sur une entrée de taille n. Selon la définition du problème du tri, la sortie de A sur entrée T doit être une permutation des éléments de T. En d'autres termes, chaque feuille de D représente une permutation des n positions d'un tableau donné en entrée. On peut donc voir chaque feuille comme une permutation de $\{1, 2, \ldots, n\}$. Par exemple, si $n = 3$, la permutation $(2, 1, 3)$ représente une exécution où $T[2]$ s'est retrouvé au début, $T[1]$ en deuxième et $T[3]$ en troisième. Plus généralement, si $P = (p_1, p_2, \ldots, p_n)$ est une permutation de $\{1, 2, \ldots, n\}$, c'est que $T[p_i]$ s'est retrouvé en position i, pour tout $1 \leq i \leq n$.

On prétend qu'il doit y avoir au moins $n!$ feuilles dans D (rappelons que $n!$ est le nombre de permutations de n éléments). Si ce n'est pas le cas, alors il y a une permutation $P = (p_1, p_2, \ldots, p_n)$ qui n'est jamais retournée par A. Il n'est pas difficile de concevoir un tableau T qui nécessite la permutation P pour être trié (je vous laisse y réfléchir). Ceci veut donc dire que A ne retourne pas un tri correct sur entrée T, ce qui contredit son bon fonctionnement.

Donc l'arbre de décision D a au moins $n!$ feuilles et, tel que stipulé dans le Lemme 2, la hauteur de D est au moins $\log(n!) \in \Omega(n \log n)$ (selon le

lemme 1). Puisque la hauteur de D donne un nombre minimum d'instructions pour l'exécution de A, on déduit que A prend un temps $\Omega(n \log n)$ dans le pire cas. □

3.5 Tri sans comparaisons

Si on a des objets dont l'ordre est (relativement) complexe, il est difficile de trier sans comparaison. Par exemple, comment trier des *strings* sans que l'on nous donne une fonction qui détermine qu'une chaîne est plus petite qu'une autre ?

Par contre, la comparaison entre les entiers est très simple, naturelle et il s'avère que nos machines représentent les zones mémoires avec des entiers (dans le modèle RAM, du moins, mais nous n'allons pas nous étendre sur les modèles d'ordinateurs ici). En d'autres termes, il y a certains cas où l'on peut trier des entiers sans faire de comparaisons.

Commençons par un exemple simple : le tri des n premiers entiers. Supposons qu'on vous donne un tableau T et on vous *garantit* qu'il contient tous les entiers de 1 à n, chacun exactement une fois. Il n'est pas très compliqué de trier T en temps $O(n)$.

fonction *triPremiersEntiers(T) //T contient* $\{1, 2, \ldots, n\}$
  $S = [0] * n$ `// S est un tableau avec n zéros`
  pour $i = 1..n$ **faire**
    $S[T[i]] = T[i]$
  fin
  return S

Une question naturelle qui devrait survenir est "mais pourquoi ne pas simplement créer un tableau avec les n premiers entiers et le retourner" ? C'est vrai. Le tri ci-haut prend son sens lorsque vous avez des paires clé-valeur dont les clés sont des entiers, et que vous voulez trier par clé.

3.6 Le tri comptage

De façon plus importante, le principe appliqué ci-haut pour des tableaux T qui contiennent des entiers entre 1 et k, et dans lesquels certaines valeurs peuvent être répétées. C'est ce qu'on appelle le tri comptage. Nous présentons d'abord une version basique.

```
fonction triComptage_basique(T) //T est de taille n
    k = 0
    pour i = 1..n faire
        k = max(k, T[i])
    fin
    S = [0] * k                 // S est un tableau avec k zéros
    pour i = 1..n faire
        S[T[i]] += 1
    fin
    R = [ ]                                  // Le tableau de retour
    pour i = 1..k faire
        pour j = 1..S[i] faire
            R.append(i)
        fin
    fin
    return R
```

L'algorithme triComptage s'exécute en temps $O(n+k)$, où k est la valeur maximum du tableau. Bien sûr, il faut que l'entrée T ne contienne que des entiers non négatifs. Un exercice simple est d'adapter l'algorithme pour supporter les entiers négatifs. Un autre exercice simple est d'améliorer la complexité à $O(n + k')$, où k' est l'écart entre le minimum et le maximum du tableau.

3.6.1 Comment trier des clés

Revenons à la remarque concernant le tri de paires clé-valeur par des clés entières. Ce qui arrive fréquemment, c'est que T soit en fait un tableau d'objets et que l'on veuille utiliser un attribut pour faire le tri. On appelle cet attribut une **clé**. Par exemple, on peut avoir un tableau de paires entiers/strings $T = [(1, "allo"), (4, "salut"), (1, "ciao")]$. Si vous vous rappelez vos structures de données, T peut représenter un dictionnaire. Il se peut qu'on veuille trier les paires par leur premier attribut, la clé, et qu'on veuille aussi garder leur valeur associée. Par exemple, une version triée de T serait $T = [(1, "allo"), (1, "ciao"), (4, "salut")]$.

On peut facilement adapter tous les algorithmes de tri vus jusqu'à maintenant Dans ce cas, le triComptage tel que présenté ci-haut ne fonctionnera pas : l'algorithme insère de nouveaux entiers dans un nouveau tableau retourné. Il existe plusieurs façons de modifier le triComptage pour tenir compte de ce problème. Une de ces façons est de calculer, pour chaque

entier i présent dans les clés de T, l'intervalle des positions destinées à entreposer les i dans la sortie. Dans l'exemple ci-haut, ou voudra calculer que les éléments avec la clé 1 sont destinés aux positions 1 à 2, les éléments avec la clé 2 n'ont aucune destination et les éléments avec la clé 4 sont destinés à la position 3.

```
fonction triComptage(T) //T est de taille n
    k = 0
    pour i = 1..n faire
    |   k = max(k, T[i])
    fin
    S = [0] * k                 // S est un tableau avec k zéros
    pour i = 1..n faire
    |   S[T[i]]+=1
    fin
    pour i = 2..n faire
    |   S[i] += S[i − 1]
    fin
    R = [0] * n                             // Le tableau de retour
    pour i = n..1 faire
    |   R[S[T[i]]] = T[i]
    |   S[T[i]]-=1
    fin
    return R
```

Nous vous recommandons de faire la trace de cet algorithme sur un exemple pour bien cerner ce qu'il fait. L'intuition derrière cet algorithme est la suivante : $S[i]$ compte d'abord le nombre de i dans T. La deuxième boucle modifie ensuite S pour dire que la valeur 1 doit aller aux positions $(0, S[1]]$, que 2 doit aller aux positions $(S[1], S[2]]$ et ainsi de suite. Ici, par $(a, b]$, on veut dire les positions entre a et b en excluant a mais en incluant b.

3.6.2 Stabilité d'un tri

Remarquez une différence fondamentale avec le tri comptage basique : les éléments insérés dans R sont les éléments de T, et non des entiers nouvellement créés. De plus, les éléments dont la clé est égale apparaissent en sortie dans le même ordre relatif que dans l'entrée. Toujours avec le même exemple plus haut, c'est $T = [(1, "allo"), (1, "ciao"), (4, "salut")]$ qui sera retourné par ce tri, et non $T = [(1, "ciao"), (1, "allo"), (4, "salut")]$ (parce que dans l'entrée, $(1, "allo")$ apparaît avant $(1, "ciao")$. Cette propriété s'appelle la

stabilité et, dans certains contextes, peut être désirable.

Définition 2. *Un algorithme de tri est* stable *si, dans la sortie, les éléments égaux apparaissent dans le même ordre que dans l'entrée.*

3.7 Le tri radix

Le tri radix utilise une représentation des objets à trier par une séquence de caractères, puis trie selon le dernier caractère, puis selon l'avant-dernier, et ainsi de suite. Par exemple, si on a des entiers à trier, on prendra leur représentation en binaire et on triera selon leur bit le plus à droite, puis selon le deuxième le plus à droite, etc. Ceci concorde avec la définition de radix, qui "base d'un système numérique".

Prenons par exemple la liste d'entiers représentés en binaire

1001
0101
0010
1011

Contrairement à la description donnée ci-haut, le plus naturel est d'abord trier selon le bit le plus significatif (le plus à gauche). Donc on obtiendrait la liste

0101
0010
1011
1001

Ce n'est pas encore complètement trié. Il faudrait faire une sous-liste avec les éléments commençant par 0 et trier cette sous-liste, puis trier la liste avec les éléments commençant par un 1. Dans l'exemple, on devrait donc trier $(0101, 0010)$ et $(1011, 1001)$ séparément. C'est faisable de façon récursive, mais ceci va créer beaucoup de sous-listes à la fin et il y a une

façon plus pratique [1].

Au lieu de trier selon les bits de gauche à droite, on va aller de droite à gauche, donc du bit le moins significatif au plus significatif. En faisant ça, l'algorithme fonctionne et il n'y a pas de sous liste à gérer. Par contre, il est important que le tri utilisé soit stable. Dans l'exemple, quand on trie selon le dernier bit, les trois éléments avec un 1 au dernier bit seront déplacés mais ils apparaissent dans le même ordre qu'à l'origine. L'exemple ci-dessous illustre l'avant à gauche, et l'après à droite.

010$\underline{1}$	0010
001$\underline{0}$	0101
101$\underline{1}$	1011
100$\underline{1}$	1001

On trierait ensuite selon l'avant-dernier bit, toujours de façon stable.

00$\underline{1}$0	0101
01$\underline{0}$1	1001
10$\underline{1}$1	0010
10$\underline{0}$1	1011

Puis, selon la prochaine position

0$\underline{1}$01	1001
1$\underline{0}$01	0010
0$\underline{0}$10	1011
1$\underline{1}$11	0101

Et finalement, selon le bit le plus significatif

1. Ce tri a été inventé à l'époque des cartes perforées (et même avant en fait). Il fallait un opérateur pour gérer les sous-listes manuellement, ce qui demandait un certain effort.

$\underline{1}001$	0010
$\underline{0}010$	0101
$\underline{1}011$	1001
$\underline{0}101$	1011

et magie ! La liste est triée.

L'algorithme du tri radix est alors très simple, dans la mesure où il a accès à un tri stable. On va supposer que l'on a accès à une fonction $triComptage(T, pos)$ qui effectue un tri stable en utilisant la valeur à position pos comme clé (vous ne devriez pas avoir de difficulté à voir qu'une légère modification au pseudo-code donné plus haut devrait suffire). En particulier, dans le cas où on trie des entiers en binaire, il n'y a que deux valeurs possibles dans le tableau des comptes, et $triComptage$ prendra un temps $O(n + 2) = O(n)$.

fonction *triRadix(T) //T est de taille n*
  $maxpos$ = position la plus à droite des éléments de T
  pour $pos = maxpos..1$ **faire**
    $triComptage(T, pos)$
  fin

Notez que l'on peut trier des entiers dans leur représentation décimale, des chaînes de caractères, ou bien n'importe quel ensemble d'éléments dont l'ordre dépend de positions de moins en moins significatives.

Complexité. Plusieurs variables entrent en ligne de compte dans le tri radix. Supposons que l'on trie des éléments qui dépendent de w positions, et que chaque position a d valeurs possibles. On fera alors w appels au tri comptage, et chaque appel se fera en temps $O(n + d)$. La complexité est donc de $O(w(n + d))$.

Dans nos architectures d'ordinateurs habituelles, les entiers sont stockés sur w bits, qui est habituellement une constante (e.g. 32 ou 64). En considérant aussi $d = 2$, ceci veut dire que le tri radix est $O(n)$ dans ces cas. Par contre, le tri fusion ou encore quickSort sont reconnus pour mieux performer en pratique. Une des raisons (parmi d'autres) est que $\log n < 64$ la plupart du temps, et donc $n \log n$ se compare bien à $w(n + d)$.

Exactitude. Comment peut-on se convaincre que le tri radix est correct ? On va argumenter (informellement ici) que les éléments sont d'abord triés en considérant la dernière position seulement, puis triés selon les deux dernières positions prises ensemble, puis selon les trois dernières positions ensemble, et ainsi de suite.

On déclare donc l'invariant de boucle "après avoir trié selon la position j, les éléments sont triés selon les positions $j, j+1, \ldots, maxpos-1, maxpos$". Ici, "selon les positions X" veut dire que l'on considère la position j comme la plus significative, et $maxpos$ la moins significative. Si ceci est vrai pour toutes les positions $j \in \{1, \ldots, maxpos\}$, ceci voudra dire qu'à la fin, T sera trié en considérant toutes les positions. Notre l'invariant de boucle démontre bel et bien ce qu'on veut. Notez que notre induction se fait "à l'envers". Notre cas de base sera $j = maxpos$ car l'algorithme débute par la fin, et on va vouloir prouver le cas j en supposant que notre invariant est vrai pour les positions $\{j+1, \ldots, maxpos\}$.

Pour voir que l'invariant de boucle est maintenu, on observe qu'il est d'abord vrai quand $j = maxpos$ car l'algorithme trie selon cette position au début. Pour $j < maxpos$, on suppose que T est trié selon les positions $\{j+1, j+2, \ldots, maxpos\}$. En triant selon la j-ième position, on obtient un tableau que l'on appellera ici T'. Les éléments de T' sont ordonnés selon la position j. Par exemple dans le cas binaire, les éléments avec 0 en position j viennent avant ceux avec 1. Considérons maintenant tous les éléments X ayant une même valeur x en position j. Les éléments de X sont consécutifs dans T', et on voudrait que X soit trié selon les positions $\{j+1, \ldots, maxpos\}$. Mais selon notre invariant de boucle, c'était le cas dans T, et ce sera aussi le cas dans T' puisqu'on utilise le tri comptage, qui est stable. L'algorithme est donc correct.

3.8 Récapitulatif des tris

Le tableau à la page suivante résume les tris qui ont été vus (en plus du tri monceau — si vous ne le connaissez pas, cherchez-le !) :

En bref :

Algo	Pire cas	Meilleur cas	Avantages	Inconvénients
Insertion	$O(n^2)$	$O(n)$	Il trie surplace, facile à implémenter, utilise peu de mémoire. Très rapide si le tableau est déjà trié (ou presque).	Pas optimal, d'autres tris existent en temps $O(n \log n)$.
Fusion	$O(n \log n)$	$O(n \log n)$	Il trie en $O(n \log n)$, facile à implémenter, permet à votre prof de s'en servir comme exemple d'analyse par induction.	Utilise $O(n)$ mémoire supplémentaire pour créer un 2e tableau solution (c'est faisable sur place, mais pas évident du tout). Si le tableau est déjà trié, prend quand même $O(n \log n)$.
Quicksort	$O(n^2)$	$O(n \log n)$	Performe bien en pratique ($n \log n$ avec forte probabilité), surtout si on prend le temps d'implémenter une version sur place.	Le pire cas est $O(n^2)$.
Comptage	$O(n + k)$	$O(n + k)$	Tri sans comparaison, très utile pour trier des entiers de taille au plus k.	Utilise $O(k)$ et $O(n)$ mémoire supplémentaire. Pas utile du tout si k est grand.
Radix	$O(w(n + d))$	$O(w(n + d))$	Utile si on trie des éléments qui peuvent être représentés en chaînes de w éléments avec d valeurs possibles.	Pas très utile si w et d sont grands. En pratique, fait trop de récursions.
Monceau	$O(n \log n)$	$O(n \log n)$	Utilise les propriétés du monceau (insertion $O(\log)$, getMin $O(1)$, supprimerMin $O(\log n)$)	Nécessite $O(n)$ espace supplémentaire pour représenter les éléments en monceau. Demande aussi d'implémenter une structure additionnelle.

3.9 Résumé

Ce que vous devez savoir :

— Comprenez les principes du triInsertion, triFusion, quickSort, triComptage, triRadix.
— Comprenez surtout les principes qui nous ont permis d'analyser ces tris.
— Rappelez-vous que l'on peut trier en temps $O(n \log n)$.
— Sachez ce qu'est un tri surplace et un tri stable.
— Sachez que le tri comptage peut trier en temps $O(n)$ si vous avez des valeurs entre 1 et $O(n)$ à trier.
— Faites les exercices.

Chapitre 4

Les graphes

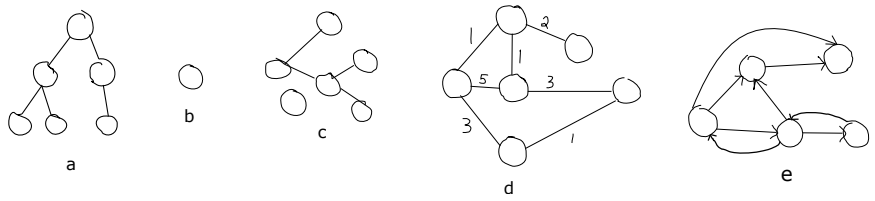


FIGURE 4.1: Des exemples de graphes. Ici, a, b et c illustrent des graphes simples, d est un graphe valué et e est un graphe orienté. Notez que dans un graphe orienté, deux arêtes à contre-sens sont possibles.

Les graphes sont des structures de données omniprésentes en algorithmique. Ils sont constitués de sommets et d'arêtes, où les arêtes lient deux sommets. Les graphes servent donc à représenter des liens entre des ensembles d'éléments. Par exemple, on peut représenter un réseau social par un graphe où les sommets sont les personnes, et où une arête entre les sommets *Alice* et *Bob* indique que ces deux personnes sont amies. On peut aussi représenter un réseau routier par un graphe (comme les diverses applications *maps* le font). Les sommets représentent les intersections ou bien les emplacements, et les arêtes représentent des rues. Dans ce cas, on peut associer un *poids* à l'arête pour indiquer la longueur de la rue ainsi qu'une *direction* s'il y a un sens unique. Autre exemple, en bio-informatique les sommets peuvent être des protéines au sein d'un organisme et les arêtes représentent une interaction quelconque.

4.1 Quelques définitions

Un graphe est constitué de sommets et d'arêtes. Chaque arête associe deux sommets. On écrit souvent $G = (V, E)$ pour dénoter un graphe G dont les sommets sont V et les arêtes sont E. On définit $n = |V|$ et $m = |E|$, i.e. n est le nombre de sommets et m le nombre d'arêtes. On écrit parfois $V(G)$ et $V(E)$ pour dénoter les sommets et les arêtes de G, respectivement. Un graphe peut être orienté ou non-orienté - c'est-à-dire, ses arêtes peuvent avoir une direction ou non. Les arêtes qui ont une orientation sont parfois appelées des arcs pour rendre la distinction claire. Formellement, V peut être n'importe quoi, mais E doit désigner un ensemble de paires de sommet. Si le graphe est non-orienté, on a $E \subseteq \{\{u, v\} : u \in V, v \in V, u \neq v\}$ et s'il est orienté, $E \subseteq \{(u, v) : u \in V, v \in V, u \neq v\}$. Certains types de graphes permettent une arête de u vers u, ou bien plusieurs arêtes entre deux sommets. Nous n'allons pas allouer ces cas pour l'instant. Les graphes

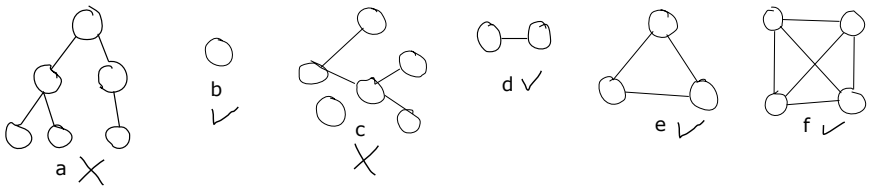


FIGURE 4.2: Graphes complets et non-complets. Notez que b est complet parce qu'il est vrai qu'entre pour toute paire de noeuds, il y a une arête.

sans boucles et sans arêtes multiples sont appelés des graphes *simples*.

Les arêtes d'un graphe peuvent avoir un poids. Ceci se définit par une *fonction de valuation* $f : E \to \mathbb{R}$ qui nous dit quel poids a chaque arête. Par exemple, dire $f(\{u, v\}) = 5$ indique que l'arête qui lie u et v a un poids de 5 (on aurait $f((u, v))$ dans le cas orienté). Un graphe peut être donné avec sa valuation, dans quel cas on écrit $G = (V, E, f)$ pour indiquer que G est valué par f.

Dans ces notes, si on utilise un graphe G sans préciser s'il est orienté ou non, on supposera par défaut qu'il est non-orienté.

Supposons que $G = (V, E)$ est non-orienté. Pour une arête $\{u, v\} \in E$, on écrira parfois $uv \in E$ pour ne pas alourdir la notation avec des accolades. Mais notez que uv et vu désignent la même arête. Si $uv \in E$, on dit que u est *adjacent* à v et que v est adjacent à u. On dira aussi que u est *incident* à l'arête uv (et v est incident à uv). Pour $u, v \in V$, on dit que v est un *voisin* de u si $uv \in E$. Le *degré* d'un sommet u, dénoté $deg(u)$, est le nombre de voisins de u, i.e. $deg(u) = |\{v \in V : \{u, v\} \in E\}|$.

Un graphe est appelé *complet* si chaque arête possible est présente, i.e. $E = \{\{u, v\} : u, v \in V \text{ et } u \neq v\}$. Question quiz : combien d'arêtes a un graphe complet à n sommets ? Il y en a $\binom{n}{2}$, car c'est le nombre de paires que l'on peut former.

Un graphe $G' = (V', E')$ est un *sous-graphe* de $G = (V, E)$ si $V' \subseteq V$ et $E' \subseteq \{uv \in E : u, v \in V'\}$. Pour $X \subseteq V$, le graphe *induit* par X est dénoté $G[X]$, et consiste en le graphe $G = (X, \{uv \in E : u, v \in X\})$.

Voici un petit résultat qui pourra être pratique.

Lemme 4. *Pour tout graphe G, $\sum_{v \in V(G)} deg(v) = 2m$.*

Démonstration. Quand on fait la somme des degrés, chaque arête uv est comptée exactement 2 fois : une fois quand on ajoute $deg(u)$, et une fois quand on ajoute $deg(v)$. Donc cette somme est égale à 2 fois le nombre d'arêtes, qui est $2m$. □

4.2 Chemins, cycles et connexité

Un chemin dans un graphe $G = (V, E)$ est une séquence de sommets $C = (v_1, v_2, \ldots, v_k)$ tous différents et tels que $v_i v_{i+1} \in E$ pour tout $i \in \{1, \ldots, k-1\}$. Donc un chemin nous dit comment nous rendre de v_1 à v_k sans repasser deux fois au même endroit. On appelle donc C un chemin entre v_1 et v_k. Si C contient des sommets répétés, on appelle cela une *marche* (walk, dans la littérature) [1].

La *longueur* d'un chemin est le nombre d'arêtes sur ce chemin. Donc un chemin $C = (v_1, \ldots, v_k)$ a une longueur $k - 1$. La *distance* entre deux sommets u et v est la longueur minimum d'un chemin entre u et v. Donc parmi tous les chemins possibles entre u et v, celui qui a le moins d'arêtes donne la distance. On dénote cette distance par $dist(u, v)$. Si le graphe est valué, la longueur du chemin est la somme des poids de ses arêtes, et la distance est encore la longueur minimum possible.

Un *cycle* est une séquence de sommets $C = (v_1, v_2, \ldots, v_k, v_1)$ telle que $(v_1, \ldots, v_k)$ est un chemin et $v_k v_1 \in E(G)$. Donc un cycle nous dit comment partir de v_1 et y revenir sans repasser deux fois au même endroit (sauf v_1). La longueur d'un cycle est son nombre d'arêtes, qui cette fois-ci est égal à son nombre de sommets.

Un graphe est *connexe* si, pour toute paire de sommets $u, v \in V$, il existe au moins un chemin entre u et v.

Si un graphe n'est pas connexe, alors il est constitué de l'union de plusieurs graphes connexes. Ces sous-graphes sont appelés des *composantes connexes*. Pour être formel, une composante connexe C de G est un sous-graphe induit de G qui est connexe et qui est maximal. Ici par *maximal*, on veut dire que si on ajoute un sommet à C, alors le graphe devient non-connexe.

1. Dans la littérature, certains auteurs permettent à un chemin de répéter des sommets. Ils/elles appellent un chemin sans répétition un chemin *simple*. Pour nous un chemin ne répète pas de sommets et une marche en répète.

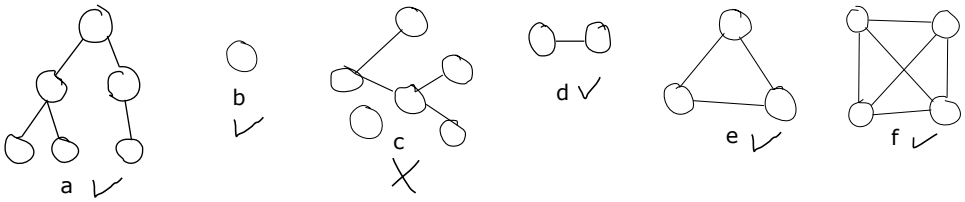


FIGURE 4.3: Graphes connexes et non-connexes. Notez que b est connexe parce qu'il est vrai qu'entre pour toute paire de noeuds, il existe un chemin.

On peut se demander combien d'arêtes il faut à un graphe pour qu'il soit connexe. On démontre un résultat plus général. Par exemple, prenez le théorème suivant avec $k = n - 2$ arêtes. Alors le graphe a au moins $n - k = n - (n - 2) = 2$ composantes connexes. En d'autres termes, il faut au moins $n - 1$ arêtes pour qu'un graphe soit connexe.

Théorème 8. *Un graphe avec k arêtes et n sommets, où $0 \leq k \leq n - 1$, a au moins $n - k$ composantes connexes.*

Démonstration. On démontre par induction sur k. Comme cas de base, quand $k = 0$, chaque sommet forme une composante connexe et il y en a donc $n = n - 0 = n - k$.

On suppose par induction que tout graphe ayant n sommets et $k-1$ arêtes a au moins $n - (k - 1) = n - k + 1$ composantes connexes. Soit $G = (V, E)$ un graphe avec $k > 0$ arêtes, et G' le sous-graphe obtenu en retirant une arête uv choisie arbitrairement (formellement, $G' = (V, E \setminus \{uv\})$. Alors G' a $k-1$ arêtes et, par induction, a au moins $n-k+1$ composantes connexes.

Il y a deux cas possibles. Cas 1 : si u et v sont dans la même composante connexe de G', le nombre de composantes connexes de G reste au moins $n-k+1 \geq n-k$, car réinsérer uv ne peut pas créer des nouvelles composantes. Case 2 : si u et v sont dans deux composantes connexes différentes C_1 et C_2 de G'. Lorsqu'on réinsère uv, ceci a pour effet de "réunir" C_1 et C_2 en une seule composante, et le nombre de composantes diminue donc de 1. Donc G a au moins $n - k + 1 - 1 = n - k$ composantes, tel que désiré. □

En remplaçant k par $n - 2$, on a le corollaire suivant.

Corollaire 1. *Tout graphe connexe a au moins $n - 1$ arêtes.*

4.3 Chemin le plus court

Considérez le problème suivant :

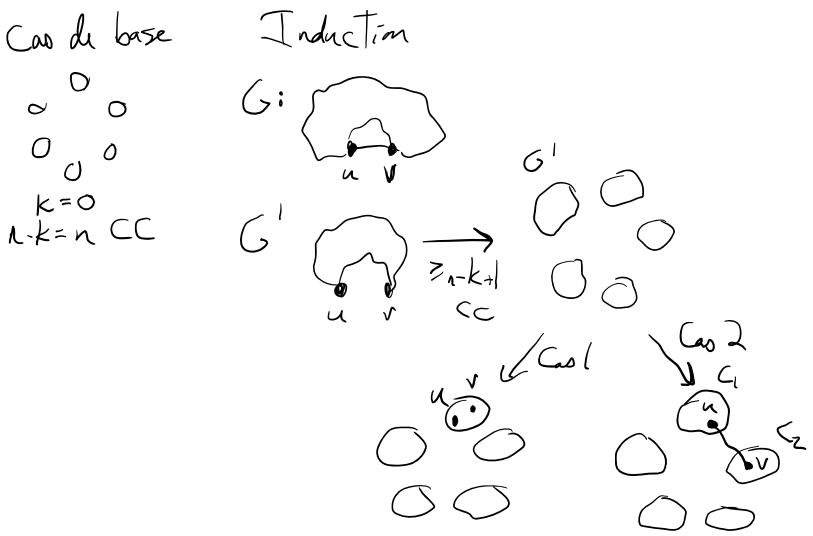


FIGURE 4.4: Une représentation visuelle de la preuve. Ici, CC veut dire "composante connexe".

Problème du chemin le plus court
Entrée : un graphe $G = (V, E)$ et deux sommets $u, v \in V$.
Sortie : un chemin de longueur minimum entre u et v, s'il en existe un.

Nous avons un algorithme en temps $O(n+m)$ qui peut trouver un chemin de longueur minimum. L'idée est de démarrer de u, de visiter ses voisins, puis les voisins des voisins, etc. Une file s'occupera de gérer dans quel ordre on visite les voisins des voisins. Chaque fois qu'on visite un sommet, on se rappelle de qui nous y a mené. On répète jusqu'à ce qu'on ait trouvé v, et on reconstruit le chemin.

```
fonction plusCourtChemin(G = (V, E), u, v) //V est de taille n
    visités[u] = ∅
    file = [u]
    fini = False
    tant que ¬file.vide() et ¬fini faire
        w = file.pop()
        pour chaque voisin z de w faire
            si z ∉ visités alors
                visités[z] = w
                file.append(z)
                si z == v alors
                    fini = True
        fin
    fin
    si v ∉ visités alors
        return "Aucun chemin n'existe"
    C = (v)                    // on va reconstruire le chemin
    w = v
    tant que w ≠ u faire
        w = visités[w]
        C.insérerDébut(w)
    fin
    return C
```

Nous n'allons pas nous étendre sur l'exactitude de cet algorithme. Mais quelle est la complexité ? Il y a 2 niveaux de boucle, chacune s'exécutant au pire n fois. Donc ça doit être $O(n^2)$, non ? Oui, c'est vrai parce que O est un comportement maximum, mais on peut aussi dire que c'est $O(n+m)$ en raffinant l'analyse.

Tout d'abord, l'algorithme fait des utilisations implicites de structures de données. La gestion de la file peut se faire en temps $O(1)$ par opération. Il faut ajouter et accéder aux éléments du dictionnaire *visités*. On peut l'implémenter en table de hachage pour que ces opérations prennent un temps $O(1)$ (avec forte probabilité, détail que nous ignorons ici). On peut aussi supposer que $V = \{1, 2, \ldots, n\}$ et implémenter *visités* comme un tableau de n éléments. Ceci demande un tout petit peu de travail, mais c'est faisable et chaque opération se fera en temps $O(1)$. Il n'est pas trop difficile de voir que les autres instructions peuvent toutes se faire en temps $O(1)$. Il faut se convaincre que l'on peut exécuter "pour chaque voisin z de w" en

temps constant à chaque tour de boucle — c'est-à-dire, peut-on énumérer les voisins aussi facilement ? On va le supposer pour l'instant, voir la prochaine section pour les détails.

Attardons-nous d'abord au nombre de fois que l'on entrera dans la première boucle "tant que", au total dans toute l'exécution. On voit qu'un sommet n'entrera jamais deux fois dans une file. La raison est que l'on n'enfile jamais des sommets *visités*, et chaque fois qu'on enfile un élément, on le marque comme visité. Donc chaque *pop*() de la file donne un sommet différent à travers toute l'exécution, et on entre au maximum n fois dans la boucle.

Combien de fois entrera-t-on dans la deuxième boucle (la "pour chaque") au total ? Pour chaque sommet w considéré, on passe dans cette boucle $deg(w)$ fois, une fois pour chaque voisin. On vient d'argumenter qu'on ne repasse jamais deux fois avec le meme w, donc le nombre de fois que l'on entre dans cette boucle est au maximum $\sum_{w \in V} deg(w) = 2m$. Ceci est le nombre total de fois que l'on entre dans cette boucle, indépendamment de la première boucle.

On vient d'argumenter que les instructions au sein de la première boucle s'exécutent $O(n)$ fois, et celles au sein de la deuxième boucle s'exécutent $O(m)$ fois. Donc on se retrouve avec une complexité de $O(n + m)$.

4.4 Fouille en largeur

L'algorithme *cheminPlusCourt* peut être utilisé pour trouver tous les sommets que l'on peut atteindre à partir d'un sommet u. Considérez l'algorithme suivant, qui est presque identique au précédent, mais avec la différence notable que l'on n'a pas de sommet cible v.

```
fonction fouilleEnLargeur(G = (V, E), u) //V est de taille n
    visités = {u}
    file = [u]
    tant que ¬file.vide() et ¬fini faire
        w = file.pop()
        pour chaque voisin z de w faire
            si z ∉ visités alors
                visités.insert(z)
                file.append(z)
        fin
    fin
    return visités
```

Ce parcours s'appelle une fouille en largeur. On peut s'en servir pour identifier toutes les composantes connexes d'un graphe. En particulier, ceci permet de vérifier si un graphe est connexe.

```
fonction composantesConnexes(G = (V, E)) //V est de taille n
    àVisiter = V
    CC = {}  // La liste des composantes connexes (données
     par leurs sommets)
    tant que ¬(àVisiter.vide()) faire
        u =n'importe quel sommet de àVisiter
        X = fouilleEnLargeur(G, u)
        CC.append(X)
        àVisiter = àVisiter \ X
    fin
    return CC
```

La complexité de cet algorithme est essentiellement la somme des complexités des appels à $fouilleEnLargeur$, qui totalise $O(\sum_{i=1}^{k}(n_i + m_i))$, où ici n_i et m_i sont le nombre de sommets et d'arêtes de la composante connexe i.

Puisque chaque sommet et arête n'est compté qu'une fois, ceci sera $O(n + m)$. Notons que si $àVisiter$ est implémenté en table de hachage, la suppression des éléments de X prendra un temps $O(|X|)$ avec forte probabilité, ce qui ne change pas la complexité.

4.5 Fouille en profondeur

Un autre algorithme bien connu pour parcourir un graphe est la fouille en profondeur. On visite un sommet, puis ses voisins non-visités de façon récursive. L'aspect "profondeur" vient du fait que si on démarre la fouille avec u, on va s'éloigner de plus en plus de u dans la récursion (du moins, c'est le cas si G est un arbre).

L'appel initial devrait se faire avec un u arbitraire et $visités = \{u\}$. Le paramètre R devrait contenir u initialement, et sera rempli des sommets visités en guise de retour.

fonction *fouilleEnProfondeur*$(G = (V, E), u, visités, R)$
  pour chaque *voisin* v *de* u **faire**
    si $v \notin visités$ **alors**
      $visités.insert(v)$
      $R.append(v)$
      $fouilleEnProfondeur(G, v, visités, R)$
  fin

On voit que *visités* et R jouent le même rôle, i.e. ils contiennent les sommets vus jusqu'à maintenant. On les distingue ici car *visités* est souvent implémenté en table de hachage ou en tableau, et R en liste ordonnée parce qu'on veut les sommets dans un ordre précis. La fouille en profondeur est d'ailleurs particulièrement utilisée dans les arbre pour énumérer les noeuds dans un ordre prévisible.

4.6 Arbres

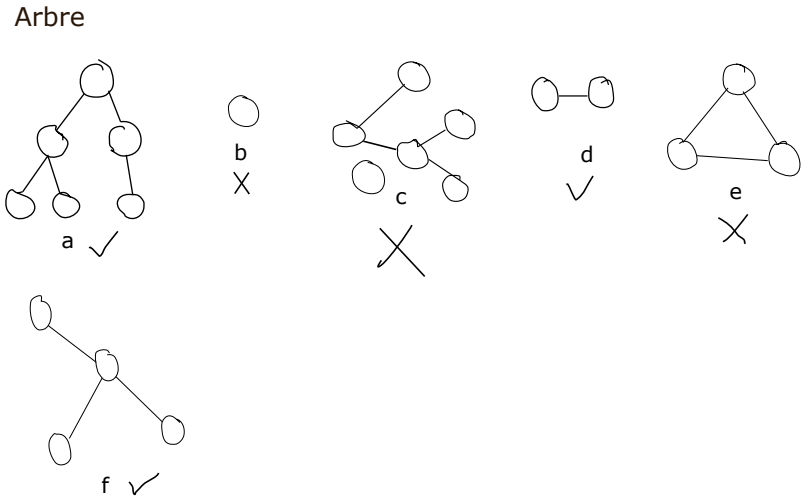
Un *arbre* est un graphe qui est connexe et qui ne contient pas de cycle. Ceci est tellement fondamental que l'on va le répéter en définition.

Définition 3. *Un arbre est un graphe connexe qui ne contient pas de cycle.*

Il y a plusieurs définitions équivalentes à celle ci-haut. Nous en présentons deux autres ici.

Théorème 9. *Soit $G = (V, E)$. Alors les énoncés suivants sont logiquement équivalents :*

1. *G est connexe et ne contient pas de cycle.*
2. *G est connexe et a $n - 1$ arêtes.*



Démonstration. On démontre l'équivalence en montrant qu'un énoncé implique l'autre.

(1) $\Rightarrow$ (2) : supposons que G est connexe et n'a pas de cycle. On veut montrer que G a $n-1$ arêtes. On le démontre par induction sur n, le nombre de sommets. Comme cas de base, quand $n = 1$, il n'y a qu'un seul graphe, il est connexe sans cycle et a effectivement $n - 1 = 0$ arêtes.

On suppose donc par induction que tout graphe avec moins de n sommets connexe et sans cycle a $n-1$ arêtes. Soit $uv \in E(G)$, et soit G' le sous-graphe obtenu de G en retirant uv. Il ne peut pas y avoir de chemin entre u et v dans G' : s'il y avait un tel chemin $C = (u, \ldots, v)$, alors dans G, $(u, \ldots, v, u)$ formerait un cycle alors qu'on suppose que G n'a pas de cycle. Donc G' n'est pas connexe et a donc plusieurs composantes connexes. Il doit y avoir deux telles composantes C_1 et C_2, car le retrait d'une seule arête ne peut pas en créer plus. Soit c_1 et c_2 le nombre de sommets de C_1 et C_2, respectivement. On sait que $c_1 + c_2 = n$ et que $C_1 < n$ et $C_2 < n$, car une contient u et l'autre v. Aussi, C_1 et C_2 n'ont pas de cycle, car retirer une arête ne peut pas créer de cycle. On peut donc utiliser l'induction sur C_1 et C_2. On déduit que C_1 a $c_1 - 1$ arêtes et C_2 en a $c_2 - 1$. Par ailleurs, en comptant uv, G a $c_1 - 1 + c_2 - 1 + 1 = c_1 + c_2 - 1 = n - 1$ arêtes. Ceci conclut cette partie de la preuve.

(2) $\Rightarrow$ (1) : On veut montrer que si G est connexe et a $n-1$ arêtes, alors G n'a pas de cycle. Supposons que G a un cycle $C = (v_1, v_2, \ldots, v_k, v_1)$. Soit

G' le graphe obtenu en enlevant v_1v_k. Il y a toujours un chemin entre v_1 et v_k. Ceci veut dire que G' est encore connexe, mais a $n-2$ arêtes. Ceci contredit le théorème 8. Puisque l'on a attein une contradiction, on déduit que G n'avait pas de cycle. □

Dans un arbre, un sommet qui n'a qu'un seul voisin est appelé une *feuille*, et sinon c'est un *noeud interne*. Notez qu'un arbre doit avoir au minimum 2 feuilles, et qu'il existe des arbres avec deux feuilles (laissé en exercice).

Si vous avez étudié les structures de données au préalables, vous avez probablement entendu parler d'arbres binaires de recherche, d'arbres AVL, arbres rouge-noir, etc. Ces arbres ont tous une *racine*, alors que les arbres tels que définis ci-haut n'ont pas de sommet spécial appelé la racine. Il y a donc une distinction à faire entre les arbres enracinés et non-enracinés. La définition ci-haut désigne un arbre non-enraciné.

Un arbre est enraciné s'il contient un noeud spécial appelé la racine. Si on a un arbre non-enraciné, on peut le transformer en enraciné en prenant n'importe quel sommet et en l'appelant la racine. Au final, c'est le même graphe, mais son interprétation change.

Appelons la racine r. Les voisins de r sont appelés ses enfants. Pour un noeud $v \neq r$, il n'y a qu'un seul voisin v' de v tel que $dist(v', r) < dist(v, r)$. Ce sommet v' est appelé le *parent* de v. Les autres voisins sont appelés des *enfants*. Notez que les feuilles d'un arbre sont les sommets qui n'ont pas d'enfant.

Comme nous le verrons plus tard, les arbres, enracinés ou non, ont plusieurs intérêts algorithmiques.

4.7 Représentation d'un graphe

Il y a plusieurs façons de stocker un graphe en mémoire. La complexité des opérations communes peut dépendre de cette représentation. Les deux options les plus communes sont les suivantes.

Liste d'adjacences. On représente les adjacences dans une liste de listes. La liste 1 contient tous les voisins du sommet 1, la liste 2 tous les voisins du sommet 2, et en général la liste j contient tous les voisins du sommet j.

Donc pour obtenir le k-ième voisin de i, on écrirait $listeAdj[i][k]$. Notons que pour attribuer un poids aux arêtes/arcs, $listeAdj[i][k]$ pourrait être une paire constituée d'un sommet et d'une valeur.

Matrice d'adjacences. On a une matrice M de taille $n \times n$ dans laquelle $M_{i,j} = 1$ si j est un voisin de i, et $M_{i,j}$ est 0 sinon. Si les arêtes ont des poids, on peut remplacer le 1 par le poids. Une matrice d'adjacence est typiquement représentée par un tableau 2D dynamique.

Différences de complexité. Le tableau suivant montre les complexités des opérations communes sur des graphes. La plupart du temps, nous n'allons pas faire trop d'ajouts/suppression sur des sommets et arêtes. La liste d'adjacence sera donc souvent plus avantageuse, puisqu'elle permet d'itérer à travers les voisins d'un sommet plus rapidement.

	Liste d'adjacence	Matrice d'adjacence
Ajouter un sommet (sans voisins)	$O(1)$. On ajoute une entrée à la fin de la liste	$O(n^2)$. On doit recréer toute la matrice
Supprimer un sommet	$O(n+m)$. Il faut passer à travers chaque sous-liste pour enlever le sommet. La somme des tailles de chaque sous-liste est 2m.	$O(n^2)$. On doit recréer toute la matrice
Ajouter une arête uv.	$O(1)$. On peut insérer en fin des deux listes de u et v.	$O(1)$. On pose $M_{u,v} = 1$.
Supprimer une arête uv.	$O(deg(u) + deg(v))$. On doit supprimer dans deux sous-listes.	$O(1)$. On pose $M_{u,v} = 0$.
Vérifier si uv est une arête.	$O(deg(u))$. Il faut passer la liste des voisins de u.	$O(1)$. On regarde $M_{u,v}$.
Obtenir le k-ième voisin de u.	$O(1)$. On retourne $liste[u][k]$.	$O(n)$. Il faut passer toute la rangée de u et compter les voisins rencontrés.

Si on en revient à la complexité de la fouille en largeur, on voit que si on a une liste d'adjacence, implémenter une boucle du type "pour chaque voisin z de w" prend un temps $O(1)$ pour passer d'un z au suivant. Par contre, ce n'est pas le cas pour la matrice d'adjacence, sauf que cette dernière a l'avantage de permettre de savoir si uv est une arête en temps $O(1)$.

Dans la plupart des cas, nos graphes ne seront pas dynamiques et ne demanderont pas de faire d'ajout/suppression de somets et d'arêtes. Dans ces situations, on pourra supposer que l'on a accès aux *deux* représentation, et on pourra itérer à travers les sommets en temps $O(1)$ par sommet, et

demander si uv est une arête ou non en temps $O(1)$. L'étude des complexités optimales sur des graphes dynamiques (i.e. qui changent souvent) est un sujet de recherche actif.

Notons que d'autres représentations sont possibles, par exemple la matrice d'incidence ou la table d'arêtes. Leur utilité est limitée et nous n'en discuterons pas ici.

Graphes orientés. Notez que la liste d'adjacence et la matrice d'adjacence peuvent aussi représenter un graphe orienté. Dans la liste d'adjacence, il suffit de poser que $liste[u]$ contient la liste des voisins *sortants* de u. Il serait aussi concevable que $list[u]$ contienne *deux* listes : une pour les voisins entrants et une pour les sortants. Dans la matrice d'adjacence, $M_{u,v} = 1$ indique que (u, v) est un arc (donc en particulier, on pourrait avoir $M_{u,v} = 1$ et $M_{u,v} = 0$).

4.8 Graphes orientés acycliques et tri topologique

Une application classique des graphes orientés est de "trier" les sommets par ordre de priorité. C'est-à-dire, supposons que l'on veut organiser des tâches interdépendantes, avec des dépendances du style "tâche A doit être complétée avant la tâche B".

On peut représenter cette situation avec un graphe orienté $G = (V, E)$, où V est l'ensemble des tâches et $(u, v) \in E$ si u doit être complété avant v. Notons d'abord que s'il y a un cycle orienté, vous êtes mal-pris, car il n'y a aucune façon de satisfaire l'ordre des tâches. En supposant qu'il n'y a pas de cycle orienté, comment déterminer un ordre des tâches qui satisfait les contraintes de précédence ? C'est ce que le *tri topologique* fait. Nous devons d'abord introduire quelques définitions supplémentaires.

Si $G = (V, E)$ est un graphe orienté, on dénote, pour $u \in V$, $N^+(u) = \{v \in V : (u, v) \in E\}$ et $N^-(u) = \{v \in V : (v, u) \in E\}$. On appelle $N^+(u)$ le voisinage sortant de u et $N^-(u)$ le voisinage entrant. Un *cycle orienté* est une séquence de sommets $(v_1, v_2, \ldots, v_k, v_1)$ telle que $v_i \neq v_j$ pour tout $1 \leq i < j \leq k$ et $(v_i, v_{i+1}) \in E$ pour $i \in \{1, 2, \ldots, k-1\}$, ainsi que $(v_k, v_1) \in E$. On a la définition suivante :

Définition 4. *Un* graphe orienté acyclique *est un graphe orienté dans lequel il n'y a aucun cycle orienté.*

Notons que cette définition s'apparente à celle des arbres dans le cas non-orienté. La différence ici est qu'il peut apparaître des cycles quand on ignore la direction des arcs.

Un tri topologique demande d'ordonner les sommets de façon à n'avoir des arcs que vers "l'avant".

Définition 5. *Soit $G = (V, E)$ un graphe orienté acyclique. Un* tri topologique *de G est une permutation $(v_1, v_2, \ldots, v_n)$ de V telle que si $(v_i, v_j) \in E$, alors $i < j$.*

L'algorithme de Kahn trouve le tri topologique d'un graphe orienté, s'il en existe un.

fonction *triTopoKahn($G = (V, E)$) //V est de taille n, E de taille m*

```
R = ()                                  // La séquence à retourner
S = {u ∈ V : |N⁻(u)| = 0}                // Les sommets sans arc entrant.
tant que S ≠ ∅ faire
    s = un élément de S
    S.erase(s)
    R.append(s)
    pour chaque t ∈ N⁺(s) faire
        Enlever l'arc (s, t) de G
        si |N⁻(t)| == 0 alors
            S.append(t)
    fin
fin
si |R| ≠ n alors
    return "Pas de tri topologique possible"
return R
```

L'idée est d'ajouter les sommets sans voisin entrant d'abord. Puisque les priorités de ces sommets seront satisfaites, on peut retirer leurs arêtes incidentes. De là, on peut regarder quels nouveaux sommets n'ont pas de voisin entrant, et on continue. Si on ne réussit pas à ajouter tous les sommets, c'est qu'il devait y avoir un cycle orienté quelque part.

Nous vous laissons le soin de tester cet algorithme sur des exemples afin de vous convaincre qu'il fonctionne correctement. Quant à la complexité, elle n'est pas immédiate. Comment fait-on pour construire S au départ ? Si on a une liste d'adjacence, on peut parcourir chaque arc de la structure et conserver un compteur sur le nombre d'arcs entrant pour chaque sommet. Ceux que l'on retient dans S sont les sommets avec un compteur 0. Ceci peut se faire en temps $O(n + m)$. Si on a une matrice d'adjacence, il faudra

un temps $O(n^2)$.

La première boucle ne sera jamais exécutée plus de n fois, une fois par sommet. Les lignes $S.erase(s)$ et $R.append(s)$ peuvent s'implémenter en temps $O(1)$ si R et S sont implémentés comme des files. Ensuite, de façon similaire à ce qui a été vu précédemment, on ne passera jamais plus de m fois dans la deuxième boucle. Donc si le contenu de cette boucle pouvait prendre un temps constant, la complexité serait de $O(n + m)$. Toutefois, enlever un arc (s, t) d'une liste d'adjacence peut prendre un temps $\Theta(deg(s))$, et on peut concevoir un pire cas dans lequel la complexité devient $O(n^2)$ avec la liste d'adjacence.

Il est possible d'obtenir une complexité $O(n + m)$ dans le pire cas en maintenant un compteur de voisins entrants au lieu de supprimer des arêtes. Ceci ne demande qu'une légère modification à l'algorithme.

4.9 Résumé

Ce que vous devez savoir :

— Connaissez les définitions principales concernant les graphes.
— Sachez parcourir un graphe et sachez surtout les adapter à vos besoins. Par exemple, comment le parcours en largeur peut servir à trouver un chemin plus court entre deux sommets.
— Rappelez-vous de ce qu'est un arbre.
— Connaissez les deux représentations d'un graphe.
— Connaissez le tri topologique.
— Faites les exercices.

Chapitre 5

Algorithmes gloutons

Nous allons maintenant étudier plusieurs techniques générales de création d'algorithmes. L'une des plus intuitives est la technique des algorithmes gloutons. On les appelle aussi parfois les algorithmes voraces, ou *greedy* en anglais. Le principe de base est simple : on construit notre solution une partie à la fois. À chaque étape où l'on doit choisir la prochaine partie, on prend celle qui est la "plus prometteuse", c'est-à-dire la partie qui est localement optimale.

Cette approche ne fonctionne pas toujours. En fait, nous avons déjà vu en devoir des exemples d'algorithmes gloutons qui ne fonctionnent pas. Dans le problème de la sous-somme, on a vu que l'approche gloutonne d'ajouter le maximum des entiers d'entrée à notre sous-somme peut produire des erreurs. On a aussi vu que l'algorithme glouton pour le commis voyageur qui choisit toujours la ville disponible la plus proche ne fonctionne pas.

Toutefois, certains problèmes admettent une solution gloutonne. Lorsque c'est le cas, l'avantage est que l'on obtient un algorithme très simple à implémenter. L'inconvénient est qu'il est parfois difficile de démontrer que l'algorithme fonctionne correctement. L'exemple canonique d'un algorithme glouton est celui pour le *problème de la sélection*.

5.1 Problème de la sélection

Supposez que vous allez à un festival, et qu'il y a plusieurs activités en parallèle. Chaque activité a une heure de début et une heure de fin. Vous voulez profiter au maximum des festivités en participant à un maximum d'activités. Par contre, vous ne pouvez pas entreprendre deux activités dont les temps se chevauchent. On peut représenter les activités par des intervalles, et le but est d'en choisir un maximum qui n'entrent pas en conflit.

Problème de la sélection
Entrée : un ensemble $A = \{[s_1, t_1] \ldots, [s_n, t_n]\}$ de paires d'entiers (où $[s_i, t_i]$ représente l'intervalle $\{s_i, s_i + 1, \ldots, t_i\}$).
Sortie : un sous-ensemble $S \subseteq A$ de taille maximum tel que pour tout $[s_i, t_i], [s_j, t_j] \in S$ avec $i \neq j$, on a $[s_i, t_i] \cap [s_j, t_j] = \emptyset$.

Quelle activité devrait-on considérer en premier ? Intuitivement, on devrait prendre l'activité dont la fin arrive le plus tôt, car elle nous laissera un maximum de temps pour en prendre d'autres par la suite. Ceci suggère l'idée suivante : on trie A selon les temps de fin. Puis, on construit S itérativement en y plaçant d'abord l'activité de temps de fin minimum. Ensuite, on ajoute la prochaine activité de fin minimum qui ne crée aucun conflit, et ainsi de

suite.

fonction *activitésGloutonnes*($A = \{[s_1, t_1], \ldots, [s_n, t_n]\}$)
  Trier A selon les temps de fin t_i
  $S = [A[1]]$
  pour $i = 2..n$ **faire**
    si $A[i].debut > S[|S|].fin$ **alors**
      $S.append(A[i])$
  fin

Plutôt simple, non ? Mais comment sait-on que cet algorithme fonctionne ? Une technique générale pour démontrer qu'un algorithme glouton fonctionne est la suivante. On prend une solution optimale, disons qu'on l'appelle S^*, et on compare la solution S qu'on obtient avec S^*. On compare les éléments de S et S^* un à un jusqu'à ce qu'on rencontre la première différence. À ce moment, on montre que l'on pourrait modifier S^* avec notre choix et ainsi obtenir une solution de valeur égale ou meilleure. Puisqu'on peut répéter ces modifications autant de fois qu'on veut, ceci veut dire qu'on transformer S^* en S en conservant l'optimalité.

Théorème 10. *L'algorithme activitésGloutonnes résout correctement le problème de la sélection.*

Démonstration. Soit $S = \{[s'_1, t'_1], \ldots, [s'_k, t'_k]\}$ l'ensemble des activités retournées par activitésGloutonnes. On suppose que les éléments de S sont triés par temps de fin. Soit $S^* = \{[s^*_1, t^*_1], \ldots, [s^*_h, t^*_h]\}$ l'ensemble des activités d'une solution optimale, triées par temps de fin (il n'y a pas de mal à trier S^* pour les fins de notre preuve). Maintenant, soit $[s'_i, t'_i]$ le premier intervalle de S tel que $[s'_i, t'_i] \neq [s^*_i, t^*_i]$. C'est-à-dire, $[s'_1, t'_1] = [s^*_1, t^*_1], [s'_2, t'_2] = [s^*_2, t^*_2], \ldots, [s'_{i-1}, t'_{i-1}] = [s^*_{i-1}, t^*_{i-1}]$, et le premier élément différent arrive à l'indice i. Si un tel indice n'existe pas, on a $S = S^*$ et S est optimal, tel que désiré.

Nous allons montrer qu'on peut tout simplement remplacer $[s^*_i, t^*_i]$ par $[s'_i, t'_i]$ dans S^*. Il suffit de montrer qu'aucun conflit ne sera créé. Notons d'abord que $s^*_j \geq t^*_i$ pour tout $j > i$. Aussi, puisque notre algorithme glouton choisit les activités en priorisant les temps de fin, on sait que $t'_i \leq t^*_i$. Notez alors que $[s'_i, t'_i]$ n'est pas dans S^*, puisque les éléments de S^* sont triés selon les temps de fin.

On a que $s^*_j \geq t^*_i \geq t'_i$ pour tout $j > i$. Donc il n'y a pas de conflit dans la solution alternative $\hat{S} = (S^* \setminus \{[s^*_i, t^*_i]\}) \cup \{[s'_i, t'_i]\}$. De plus, $\hat{S}$ a autant d'éléments que S^* et est donc aussi une solution optimale (notons que pour

dire que $\hat{S}$ a autant d'éléments, il a fallu mentionner que $[s'_i, t'_i]$ n'était pas dans S^*).

On peut répéter l'argument ci-haut autant de fois qu'on veut. À chaque itération, on obtient une solution optimale $\hat{S}$ qui a plus d'éléments en commun avec S. En répétant au plus h fois, on voit que l'on obtient S, ce qui veut dire que S est aussi une solution optimale. □

5.2 Arbres couvrants minimums et algorithme de Prim

Revenons maintenant aux graphes. Soit $G = (V, E, f)$ un graphe valué non-orienté. On suppose que G est connexe. Disons que G est un réseau informatique et qu'on veut sécuriser quelques liens qui seront utilisés pour des échanges confidentiels. Le sous-graphe obtenu avec les liens sécurisés devrait aussi être connexe. Ici $f(uv)$ représente le coût de sécuriser le lien entre u et v, et on voudrait avoir des lignes sécurisées à coût minimum.

Un *sous-graphe couvrant* d'un graphe $G = (V, E, f)$ est un sous-graphe $G' = (V, E')$ de G tel que G' est connexe et contient tous les sommets de G. On suppose que $f(uv) > 0$ pour tout $uv \in E$. Le coût de G' est $\sum_{e \in E'} f(e)$, et un sous-graphe couvrant est minimum si son coût est minimum. Il n'est pas difficile de voir qu'un sous-graphe couvrant minimum doit être un arbre : s'il y a un cycle dans G', on peut enlever une arête du cycle, rester connexe et avoir un coût plus petit. C'est pourquoi on parle souvent d'*arbre couvrant minimum*.

Définition 6. *Un arbre couvrant minimum (ACM) d'un graphe valué $G = (V, E, f)$ est un sous-graphe $G' = (V, E')$ de G tel que G' est un arbre contenant tous les sommets de G, et dont $\sum_{e \in E'} f(e)$ est minimum.*

Il s'avère que l'on peut trouver un arbre couvrant minimum de façon gloutonne. On construit l'arbre une arête à la fois. On commence par ajouter un sommet arbitraire, puis à chaque étape on ajoute le nouveau sommet à notre arbre qui nécessite l'arête de moindre coût pour ce faire. Cet algorithme a été découvert en 1930 par Jarník et redécouvert par Prim en 1957. Comme c'est souvent le cas en science, c'est ce dernier nom qui est resté et qui a baptisé ce qu'on appelle l'*algorithme de Prim*.

```
fonction primACM(G = (V, E, f))
    s = n'importe quel sommet de V
    T = ({s}, ∅)                    // Notre arbre partiel initial
    tant que |T| < n faire
        x = nil    // xy est la l'arête à insérer, f(nil) = ∞
        y = nil                     // On aura x ∈ V \ T et y ∈ V(T)
        pour chaque u ∈ V \ V(T) faire
            pour chaque v ∈ T faire
                si f(uv) < f(xy) alors
                    x = u
                    y = v
            fin
        fin
        Ajouter le sommet x à T et l'arête xy
    fin
    return T
```

Il est remarquable que cet algorithme fonctionne. On va le démontrer avec la même approche que précédemment.

Théorème 11. *L'algorithme de Prim retourne un arbre couvrant minimum.*

Démonstration. Soient $E' = \{e_1, e_2, \ldots, e_{n-1}\}$ les arêtes de l'arbre T retourné par l'algorithme de Prim, dans l'ordre où elles ont été ajoutées. Soit E^* l'ensemble des arêtes d'un sous-arbre couvrant minimum, avec $T^* = (V, E^*)$ l'arbre correspondant. Soit e_i la première arête de E' qui n'est pas dans E^* (i.e. $e_1, \ldots, e_{i-1}$ sont dans E^* mais pas e_i). On va montrer qu'on peut remplacer une arête de T^* par e_i. L'idée est simple : si on ajoute e_i à T^*, on obtient un cycle. Ce cycle doit contenir une arête au moins aussi coûteuse que e_i. On peut retirer cette arête coûteuse et obtenir un arbre couvrant de valeur égale ou moindre.

Considérons l'arbre partiel $T = (V', \{e_1, \ldots, e_{i-1}\})$ que l'on avait dans l'algorithme avant d'ajouter l'arête e_i. Soit $e_i = uv$ tel que $u \in V'$ et $v \notin V'$. Dans T^*, il y a un chemin unique C de u à v, et C n'utilise pas l'arête uv. De plus, il doit y avoir une arête yz de C telle que $y \in V'$ et $z \notin V'$. Puisque Prim choisit l'arête sortant de V' de coût minimum, on a $f(yz) \geq f(uv)$. Si on considère le sous-graphe $\hat{T} = (V, (E^* \setminus \{yz\}) \cup \{uv\})$, il est facile de voir que $\hat{T}$ est connexe. De plus $\hat{T}$ a $n-1$ arêtes, et donc c'est un arbre. Le coût de $\hat{T}$ est égal ou inférieur au coût de T^*, puisqu'on a remplacé yz par uv.

On peut répéter l'argument ci-haut jusqu'à ce que l'arbre couvrant minimum $\hat{T}$ contienne toutes les arêtes de E', ce qui veut dire que E' doit être

de coût minimum. □

Nous n'avons pas discuté de la complexité de l'algorithme de Prim. La version présentée ci-haut prendra un temps $O(n^3)$. Il est possible d'améliorer cette complexité en maintenant une liste triée M contenant les prochaines arêtes à considérer pour continuer de construire T (avec possiblement des arêtes entièrement comprises dans T que l'on a laissé traîner). L'idée est d'abord, lorsqu'on ajoute s, d'ajouter ses arêtes incidentes à M de façon triée. Lorsqu'on doit choisir une arête, on prend l'arête minimum uv de M. Si les deux sommets de uv sont déjà dans T, on jette uv et on recommence. Lorsqu'on ajoute un v à T, on ajoute ses arêtes incidentes sortant de T à M. On devrait en principe retirer les arêtes de M incidentes à v et contenues dans T, mais on ne s'en préoccupe pas ici.

L'idée clé ici est qu'on peut utiliser un monceau pour M au lieu d'une liste triée. Chaque arête sera insérée au maximum deux fois dans M, une fois pour chaque bout, et donc $|M| \in O(m)$. Chaque insertion et retrait du minimum se fait en temps $O(\log m)$, et il y aura au maximum $O(m)$ insertions et retraits. La complexité devient $O(m \log m)$ (en supposant que $m \geq n$).

fonction *primACM(*$G = (V, E, f)$*)*
    $s =$ n'importe quel sommet de V
    $M = Monceau()$
    pour chaque *voisin* t *de* s **faire**
        Ajouter st de poids $f(st)$ à M
    fin
    $T = (\{s\}, \emptyset)$ `// Notre arbre partiel initial`
    tant que $|T| < n$ **faire**
        $uv = M.popMin()$ `// Retire le min de` M `et le retourne`
        tant que u *et* v *sont dans* T **faire**
            $uv = M.popMin()$
        fin
        Ajouter v et uv dans T
        pour chaque *voisin* w *de* v **faire**
            Ajouter vw dans M
        fin
    fin
    return T

En bref :

— Principe général des algorithmes gloutons : Il fait le choix le plus prometteur localement.

— Fonctionnement de l'algorithme glouton de PrimACM : Ajoute le nouveau sommet visité dont l'arête est la plus prometteuse.

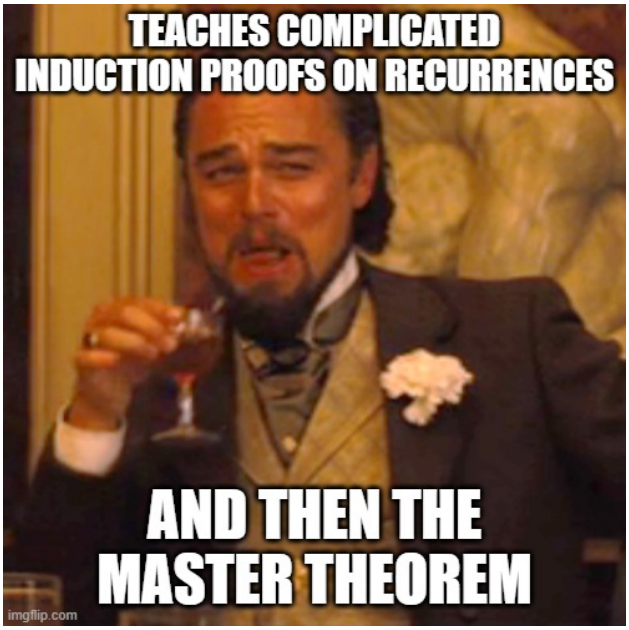
— PrimACM en exécution : `https://www.youtube.com/watch?v=cplfcGZmX7I`

“”Recette” classique de preuve pour un algorithme glouton

— Trouver une première différence à la position i entre notre solution A' et l'optimal A^*
— Remplacer l'élément $A^*[i]$ par $A'[i]$
— Montrer que $A^*[i]$ reste optimale après cette modification.
— Déduire qu'en répétant cette opération, on terminera avec un A^* optimal identique à notre A', et donc notre A' est optimal.

Chapitre 6

Algorithmes diviser-pour-régner



Le principe général d'un algorithme diviser-pour-régner peut être expliqué comme suit :

— Séparer l'instance donnée en entrée en plusieurs sous-parties ;
— Résoudre chaque partie séparément (souvent de façon récursive) ;
— Combiner les sous-solutions obtenues pour produire une solution globale.

Nous avons déjà vu un exemple classique d'algorithme de ce type : le tri fusion. Dans cet algorithme, les parties sont les deux sous-tableaux créés, ils sont triés récursivement puis combinés avec la fusion.

Donc avions vu que le tri fusion donnait lieu à une *récurrence*, c'est-à-dire une fonction f qui s'appelle elle-même. Plus précisément, dénotons par $f(n)$ le temps maximum pris par le tri fusion pour trier n éléments. Chaque appel récursif prend un temps au maximum $f(n/2)$ et la fusion se fait en temps $O(n)$ (notez ici qu'on suppose que n est une puissance de 2, et on ignore le fait qu'un des tableaux peut avoir une taille $\lceil n/2 \rceil$). On peut donc exprimer

$$f(n) \leq \begin{cases} O(1) & \text{si } n = 1 \\ 2f(n/2) + O(n) & \text{sinon} \end{cases}$$

Observez aussi que l'on incorpore la notation O au sein même de nos expressions — en particulier, on additionne $O(n)$ à autre chose. Ceci est ambigu, car O dénote un ensemble de fonctions. C'est toutefois une notation standard, et l'interprétation qu'on en fait est que $x + O(f(n))$ veut dire $x+$ "un terme qui ne dépasse pas $cf(n)$ pour une constante c".

Maintenant, comment sait-on que $f(n) \in O(n \log n)$? On peut le démontrer par induction. On veut montrer qu'il existe n_0 et c tels que $f(n) \leq cn \log n$ pour tout $n \geq n_0$. Il sera pratique de réécrire $f(n)$ avec des constantes concrètes. Par la définition de O, on a

$$f(n) \leq \begin{cases} c_1 & \text{si } n = 1 \\ 2f(n/2) + c_2 n & \text{sinon} \end{cases}$$

pour des constantes c_1 et c_2.

On prend $n = 2$ comme cas de base ($n = 1$ ne fonctionne pas car $n \log n = 0$ dans ce cas, mais ce n'est pas important car on peut poser $n_0 = 2$). On veut montrer que $f(n) \leq cn \log n$, mais on ne connaît pas c et il est difficile de le deviner d'avance. Une stratégie est de "traîner" un c inconnu, et de décider de sa valeur à la fin pour que la preuve fonctionne. C'est peut-être contre-intuitif, mais c'est une façon commune de prouver ce genre d'énoncé.

Donc quand $n = 2$, selon notre écriture de f, on a $f(n) \leq 2f(1) + c_2 \cdot 2 \leq 2c_1 + 2c_2 = 2(c_1 + c_2)$. On veut que $2(c_1 + c_2) \leq cn \log n = c2 \log 2 = 2c$, ce qui est vrai pour toute constante $c \geq c_1 + c_2$.

On suppose donc par induction qu'il existe une constante $c \geq c_1 + c_2$ telle que pour tout $m < n$ avec m est une puissance de 2, $f(m) \leq cm \log m$. Selon la définition de f, on a $f(n) \leq 2f(n/2) + c_2 n$. Par hypothèse d'induction, on a

$$\begin{aligned} f(n) &\leq 2f(n/2) + c_2 n \\ &\leq 2c(n/2)\log(n/2) + c_2 n \\ &= cn(\log n - 1) + c_2 n \\ & cn \log n - cn + c_2 n \end{aligned}$$

Pour compléter la preuve, on voudrait que $cn \log n - 2cn + c_2 n \leq cn \log n$. Ceci est automatiquement vrai si on choisit $c \geq c_2$. Donc en posant $c = c_1 + c_2$, notre cas de base et notre induction fonctionnent, ce qui conclut la preuve.

Nous verrons en exercice qu'une façon d'enlever la contrainte de la puissance de 2, une façon est de diviser f en cas de bas, cas pair et cas impair.

6.1 Cas d'étude : la recherche dichotomique

La recherche dichotomique est souvent présentée comme un algorithme diviser-pour régner. Voici une version récursive de l'algorithme, qui retourne *true* si un élément x est dans T et $false$ sinon. Ici, on suppose que T est trié.

fonction *rechercheDicho*(T, x)
  si $|T| = 1$ **alors**
    return $(T[1] == x)$
  $mid = |T|/2$
  si $T[mid] == x$ **alors**
    return *true*
  sinon si $x < T[mid]$ **alors**
    return *rechercheDicho*$(T[1..mid - 1], x)$
  sinon si $x > T[mid]$ **alors**
    return *rechercheDicho*$(T[mid + 1..|T|], x)$

Cet algorithme est perçu comme une méthode diviser-pour-régner car on

sépare le tableau en deux parties, et on résout les parties en n'en considérant qu'une seule, et la "combinaison" de solutions ne fait que retourner la solution de l'appel récursif. On peut exprimer le temps maximum $f(n)$ requis par cet algorithme comme suit :

$$f(n) \leq \begin{cases} O(1) & \text{si } n = 1 \\ f(n/2) + O(1) & \text{sinon} \end{cases}$$

ou, si on préfère,

$$f(n) \leq \begin{cases} c_1 & \text{si } n = 1 \\ f(n/2) + c_2 & \text{sinon} \end{cases}$$

pour des constantes c_1, c_2. Notez qu'on suppose toujours que n est une puissance de 2.

On "sait" que $f(n) \leq c \log n$ pour une certaine constante c, mais comment le démontrer ? Par induction bien sûr. On prend $n = 2$ comme cas de base. Selon la définition, $f(2) = f(1) + c_2 = c_1 + c_2$. On veut que $c_1 + c_2 \leq c \log(2) = c$, ce qui est vrai si on choisit $c \geq c_1 + c_2$.

Pour l'induction, on suppose que $f(m) \leq c \log m$ pour une constante c. En utilisant cette hypothèse, on obtient

$$\begin{aligned} f(n) &\leq f(n/2) + c_2 \\ &\leq c \log(n/2) + c_2 \\ &= c \log n - c + c_2 \end{aligned}$$

et on a $c \log n - c + c_2 \leq c \log n$ en choisissant $c \geq c_1 + c_2$. On peut donc trouver une constante qui font que notre cas de base et l'induction fonctionnent, montrant que $f(n) \leq c \log n$.

6.2 Les deux points les plus proches, version 1D

On vous donne n points $P = [p_1, \ldots, p_n]$ en 1 dimension, triés de gauche à droite (donc P est un un tableau d'entiers trié), et on vous demande de trouver les deux points de P les plus proches (donc p_i, p_j tels que $|p_i - p_j|$ est minimum).

Il n'est pas difficile de concevoir un algorithme en temps linéaire qui parcourt de gauche à droite, qui compare chaque p_i avec p_{i+1} et qui retourne le minimum. Pour des fins pédagogiques, nous présentons ici la ver-

sion diviser-pour-régner. L'idée est d'abord de traiter $P_1 = [p_1, \ldots, p_{n/2}]$ et $P_2 = [p_{n/2+1}, \ldots, p_n]$ séparément. Ensuite, les deux points les plus proches sont soit dans P_1, soit dans P_2, ou bien un est dans P_1 et l'autre dans P_2. Les deux premiers cas ont été vérifiés récursivement, et pour le troisième il suffit de comparer $p_{n/2}$ et $p_{n/2+1}$.

fonction *pointsPlusProches1D*($P = [p_1, \ldots, p_n]$)
  si $|P| = 1$ **alors**
    return *nil*
  si $|P| = 2$ **alors**
    return $(P[1], P[2])$
  $P_1 = [p_1, \ldots, p_{n/2}]$
  $P_2 = [p_{n/2+1}, \ldots, p_n]$
  $(x_1, y_1) = pointsPlusProches(P_1)$
  $(x_2, y_2) = pointsPlusProches(P_2)$
  retourner la paire de points de distance minimum parmi
    $(x_1, y_1), (x_2, y_2)$ et $(p_{n/2}, p_{n/2} + 1)$

Vous devriez prendre un moment pour vous convaincre que cet algorithme fonctionne. Quelle est sa complexité ? Comme c'est l'habitude, on va exprimer le temps $f(n)$ maximum que cet algorithme prend. On a

$$f(n) \leq \begin{cases} O(1) & \text{si } n = 1 \text{ ou } 2 \\ 2f(n/2) + O(1) & \text{sinon} \end{cases}$$

ou encore

$$f(n) \leq \begin{cases} c_1 & \text{si } n = 1 \text{ ou } 2 \\ 2f(n/2) + c_2 & \text{sinon} \end{cases}$$

Nous n'avons pas encore rencontré cette récurrence. Une façon de se donner une intuition est de représenter l'arbre enraciné de récursion de cet algorithme. À la racine, on a l'appel initial qui prend un temps $O(1)$ sans compter les appels récursifs. On ajoute deux enfants à la racine, un pour chaque appel récursif. Chaque enfant représente un temps $O(1)$ et on leur ajoute chacun deux enfants, correspondant aux appels récursifs faits sur ces enfants. On continue jusqu'à ce qu'on atteigne les cas terminaux, qui deviennent des feuilles dans l'arbre.

La complexité de l'algorithme est $O(1)$ multiplié par le nombre de sommets de cet arbre (car chaque appel récursif est représenté par un sommet). Combien y a-t-il de sommets ? Intuitivement, l'arbre a une hauteur $O(\log(n))$

car n est divisé par 2 à chaque récursion. Chaque sommet interne de l'arbre a deux enfants, et il est bon de savoir qu'un arbre binaire de hauteur $O(\log n)$ a $O(n)$ sommets. On peut donc deviner que la complexité de l'algorithme est $O(1) \cdot O(n) = O(n)$.

Tout ceci n'est qu'intuitif et ne constitue pas une preuve formelle. Par contre, l'idée des arbres de récursion peut parfois être utile pour "deviner" la valeur de f, surtout quand le temps passé dans chaque récursion est $O(1)$.

Maintenant, montrons que $f(n) \in O(n)$, i.e. que $f(n) \leq cn$ pour une constante c. On procède par induction sur n. On va tenter d'appliquer les concepts ci-haut directement et ceci *échouera*. Nous montrerons ensuite comment régler ce problème.

Une première tentative.

On veut montrer que $f(n) \leq cn$ pour une certaine constante c. Pour le cas de base, on prend $n = 1$ et on voit que $f(1) = c_1 \leq cn$, dans la mesure où on choisit $c \geq c_1$. Pour l'induction, on suppose que $f(m) \leq cm$ pour une constante c (à déterminer). En utilisant l'hypothèse d'induction, on obtient

$$\begin{aligned} f(n) &\leq 2f(n/2) + c_2 \\ &\leq 2c(n/2) + c_2 \\ &= cn + c_2 \end{aligned}$$

On a maintenant un problème. Peut importe le c qu'on choisit, on n'aura jamais $f(n) \leq cn$ à cause du terme c_2 qui s'y ajoutera. Il faut trouver une façon de se débarrasser de ce c_2 dans notre induction. Pour ceci, il faut modifier notre énoncé légèrement.

Une deuxième tentative.

On va prouver que pour $n \geq 2$, on a $f(n) \leq c(n-1)$ pour une constante c. Ceci implique $f(n) \leq cn$, ce qu'on veut démontrer.

On procède par induction, avec $n = 2$ comme cas de base. Par définition, on a $f(n) \leq c_1$. On aura $c_1 \leq c(2-1)$ si on choisit n'importe quel $c \geq c_1$.

On suppose que $f(m) \leq c(m-1)$ pour tout $m < n$. En utilisant l'hypothèse d'induction, on obtient

$$\begin{aligned} f(n) &\leq 2f(n/2) + c_2 \\ &\leq 2c(n/2 - 1) + c_2 \\ &= cn - 2c + c_2 \\ &= cn - c - c + c_2 \\ &= c(n-1) - c + c_2 \end{aligned}$$

et $c(n-1) - c + c_2 \leq c(n-1)$ si on prend $c \geq c_2$. Donc $c = c_1 + c_2$) satisfait notre cas de base et l'induction, ce qui montre que $f(n) \in O(n)$.

6.3 Multiplication de grands entiers

Supposons que l'on veut multiplier deux grand entiers x et y représentés en base 10 et nécessitant n chiffres. Ce n'est pas une opération en temps constant — nos architectures ne permettent de multiplier des entiers que sur un petit nombre de bits. On va s'intéresser au nombre d'opérations à faire pour multiplier x et y par rapport à n. Notons que faire l'addition $x+y$ peut se faire facilement en temps $O(n)$, mais la multiplication n'est pas si évidente.

Vous avez sans doute appris à multiplier $x \cdot y$ en multipliant chaque chiffre de y avec chaque chiffre de x, en parcourant de droite à gauche. Ceci s'appelle la longue multiplication, car elle prend un temps $O(n^2)$, voir `https://en.wikipedia.org/wiki/Multiplication_algorithm#Long_multiplication`.

On peut faire mieux avec ce qu'on appelle l'algorithme de Karatsuba. L'idée est de diviser x et y en sous-multiplications et de combiner les résultats. On peut supposer que n est pair en ajoutant, au besoin, un 0 au début de x et y. On sépare d'abord x en x_1 et x_2, où x_1 contient les $n/2$ premiers chiffres de x (de gauche à droite) et x_2 contient les chiffres $n/2 + 1$ à n. On fait de même avec y pour former y_1 et y_2.

Par exemple, si $x = 536353$ et $y = 113243$, on aura $x_1 = 536$, $x_2 = 353$, $y_1 = 113$ et $y = 243$.

On note que

$$\begin{aligned} x &= x_1 \cdot 10^{n/2} + x_2 \\ y &= y_1 \cdot 10^{n/2} + y_2 \end{aligned}$$

Maintenant, on peut observer que le produit $x \cdot y$ peut être décomposé

comme suit :

$$\begin{aligned} x \cdot y &= (x_1 \cdot 10^{n/2} + x_2)(y_1 \cdot 10^{n/2} + y_2) \\ &= 10^n x_1 y_1 + 10^{n/2}(x_1 y_2 + x_2 y_1) + x_2 y_2 \end{aligned}$$

Les additions et les multiplications par des puissances de 10 peuvent se faire en temps linéaire. Cette décomposition nécessite 4 multiplications avec des entiers sur $n/2$ chiffres, que l'on peut faire récursivement.

Si $f(n)$ est le temps pour multiplier des entiers à n chiffres, on aura $f(n) = 4f(n/2) + O(n)$. En utilisant les techniques vues ci-haut, il est possible de démontrer que $f(n) \in O(n^2)$.

Donc, on n'a rien gagné...

Il faut travailler un peu plus. Le terme du milieu peut être représenté comme suit :

$$x_1 y_2 + x_2 y_1 = (x_1 + x_2)(y_1 + y_2) - x_1 y_1 - x_2 y_2$$

et au final, on obtient

$$x \cdot y = 10^n x_1 y_1 + 10^{n/2}((x_1 + x_2)(y_1 + y_2) - x_1 y_1 - x_2 y_2) + x_2 y_2$$

L'observation principale à faire est qu'il n'y a que trois multiplications à faire ici. Si on évite de calculer le même produit deux fois, on a besoin de $x_1 y_1$, $(x_1 + x_2)(y_1 + y_2)$ et $x_2 y_2$. Chacune des trois sous-multiplications se fait avec des entiers de taille $n/2$, et la récurrence devient

$$f(n) \leq \begin{cases} O(1) & \text{si } n = 1 \\ 3f(n/2) + O(n) & \text{sinon} \end{cases}$$

Si vous faites l'arbre de récurrence de f, vous verrez que sous la racine, on a 3 noeuds prenant un temps $n/2$, puis au niveau en dessous, 9 noeuds avec un temps $n/4$, et plus généralement, au niveau k on a 3^k noeuds prenant un temps $n/2^k$. Puisque l'arbre a une hauteur $O(\log(n))$, la complexité peut être (grossièrement) estimée à $\sum_{k=1}^{\log n} 3^k \cdot n/2^k$. Il est possible de démontrer que cette expression est $\Theta(n^{\log 3})$, donc à peu près $n^{1.59}$.

C'est un peu embêtant d'avoir à faire tout ce travail à chaque récurrence que l'on rencontre. Il s'avère que l'on a une formule magique simple pour évaluer n'importe quelle récurrence du type $f(n) = af(n/b) + g(n)$. Cette

formule s'appelle le *théorème maître*.

6.4 Le Théorème Maître

Le théorème maître donne l'ordre Θ d'une équation de récurrence ayant la forme

$$f(n) = af\left(\frac{n}{b}\right) + O(g(n))$$

où $g(n)$ est de la forme n^c, ou plus généralement de la forme $n^c \log^k n$, pour des constantes $c \geq 0$ et $k \geq 0$ (donc notez que c et k peuvent être 0). Ici, a est le nombre de sous-problèmes créés à chaque récursion d'un algorithme diviser-pour-régner, b contrôle la taille des sous-problèmes et $g(n)$ est le temps passé pour faire le traitement dans un appel récursif. Le théorème maître stipule qu'il n'y a que trois possibilités :

— si $g(n) \in \Theta(n^c)$ et $c < \log_b a$, alors $f(n) \in \Theta(n^{\log_b a})$
— si $g(n) \in \Theta(n^c \log^k n)$ et $c = \log_b a$, alors $f(n) \in \Theta(n^c \log^{k+1} n)$
— si $g(n) \in \Theta(n^c)$ et $c > \log_b a$, alors $f(n) \in \Theta(n^c)$

Nous n'allons pas démontrer ce théorème, mais vous pouvez concevoir l'idée principale : il faut généraliser l'analyse de l'arbre de récursion à des valeurs arbitraires a, b et $g(n)$. Notons que les deux premiers cas correspondent à un gain obtenu grâce à la technique diviser-pour-régner, tandis que dans le troisième cas, la division n'a rien donné (comme dans notre première tentative de multiplication).

Par exemple, dans le tri fusion, on avait $f(n) = 2f(n/2) + O(n)$, et donc $a = b = 2$, $g(n) = n^1$ et donc $c = 1$. Puisque $c = \log_b a$, le deuxième cas s'applique et on obtient $f(n) = \Theta(n \log n)$. Dans l'exemple des multiplications, on avait $f(n) = 3f(n/2) + O(n)$. Ici, $c = 1 < \log_2 3$, et donc le premier cas s'applique, ce qui donne $n^{\log_2 3}$.

Maintenant que nous avons le théorème maître, plus besoin de se soucier de démontrer des récurrences, non ? Pourquoi s'embêter avec des preuves par induction et de l'analyse d'arbres de récurrence ? La réponse est simple : pour que l'on ait au moins une idée d'où provient ce théorème, et pour que l'on ait des outils pour analyser des récurrences qui n'entrent pas dans le moule du théorème. D'ailleurs, discutons de fonctions de récurrence dans lesquelles on ne divise pas l'entrée, mais on la réduit d'une taille constante.

6.5 Les deux points les plus proches, version 2D

Nous avons maintenant le problème suivant. On nous donne une liste de points non-triée $P = [p_1, \ldots, p_n]$ en deux dimensions, i.e. $p_i = (x_i, y_i)$ pour chaque $i \in \{1, \ldots, n\}$. On nous demande de trouver les deux points p_i et p_j de distance minimum, où la distance entre deux deux points $p_i = (x_i, y_i)$ et $p_j = (x_j, y_j)$ est définie par $dist(p_i, p_j) = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}$. C'est ce qu'on appelle la distance Euclidienne, qui représente la longueur du trait si on dessine une ligne entre les deux points (exercice : quel célèbre théorème nous permet de déduire cette expression pour la longueur du trait ?).

La stratégie diviser-pour-régner peut s'appliquer comme suit :

— Trier P selon les coordonnées en x ;
— Trouver, de façon récursive, les deux points les plus proches parmi $P_1 = [p_1, \ldots, p_{n/2}]$;
— Trouver, de façon récursive, les deux points les plus proches parmi $P_2 = [p_{n/2} + 1, \ldots, p_n]$;
— Soient p_i, p_j les deux points les plus proches jusqu'a maintenant, et soit $d = dist(p_i, p_j)$;
— Vérifier si deux points entre P_1 et P_2 sont plus proches que p_i et p_j.

Mais comment réaliser cette dernière étape ? Si on compare chaque paire de points, on aura un temps $O(n^2)$ et on n'aura rien gagné. Il faut utiliser le fait qu'on a déjà trouvé deux points à distance d. Donc, ce qu'on peut faire, c'est pour chaque point p_k dans P_1, de vérifier s'il y a un point dans P_2 de distance moins que d. Les autres points ne nous intéressent pas.

La première observation à faire est que l'on peut tracer une ligne verticale qui sépare $p_{n/2}$ et $p_{n/2+1}$, n'importe où, disons au milieu à la position $x = \ell$. On peut imaginer une "bande" verticale avec les extrémités aux coordonnées $\ell - d, \ell + d$ qui contient tous les points candidats intéressants. C'est-à-dire, tous les points de P_1 dont la coordonnée x est plus petite que $\ell - d$ peuvent être éliminés, de même que tous les points de P_2 dont la coordonnée x dépasse $\ell + d$. On suppose que l'on a effacé tous ces points de P_1 et P_2. Maintenant, considérons un point $p_k = (x_k, y_k)$ de P_1. S'il y a un point p_h de P_2 à distance au plus d, alors p_h doit se trouver dans la bande et, de plus, être à distance y au maximum d de y_k. Donc p_h doit être dans le rectangle dont les coins supérieur-gauche et inférieur-droit sont $(\ell - d, y_k - d)$ et $(\ell + d, y_k + d)$. L'aire de ce rectangle est $(2d)^2 = 4d^2$ (en fait c'est un carré). Appelons ce rectangle R_k. Il s'avère qu'il ne peut pas y avoir beaucoup de points de P_2 dans R_k, tel que stipulé par le lemme qui suit.

Lemme 5. *Soit p_k un point de P_1. Sachant que tous les points de P_2 sont*

à distance d ou plus, il y a au plus 32 points de P_2 contenus dans R_k. De la même façon, il y a au plus 32 points de P_1 contenus dans R_k.

Démonstration. On donne la preuve pour P_2. La preuve pour P_1 est identique.

Supposons que pour tout point p_h de P_2, on dessine un cercle de rayon $d/2$ centré à p_h, et appelons ce cercle C_h. On note que les cercles autour des points de P_2 n'ont pas d'intersection, sauf peut-être à un point de leur circonférence.

Si p_h est dans R_k, considérons l'aire de l'intersection entre R_k et C_h. Il n'est pas difficile de voir que R_k doit contenir au moins le $1/4$ de C_h (au moins, faites un dessin pour vous en convaincre). Donc l'intersection est d'au moins $\pi(d/2)^2/4 = \pi d^2/16 \geq d^2/8$ (pour simplifier les calculs). Si R_k contient plus de 32 points de P_2, la somme des intersections sera plus grande que $4d^2$, qui est l'aire de R_k. Ceci est impossible, puisque les cercles des points de P_2 ont une intersection vide (sauf peut-être un point). Donc il y a au plus 32 points qui sont à distance au plus d de p_k. □

Une analyse plus raffinée permet de réduire 32 à 6, mais de toute façon, $32 \in \Theta(6) = \Theta(1)$.

Comment utiliser cette information ? Une fois que l'on a tous les points dans notre bande, on peut les trier par coordonnée y en ordre croissant. On parcourt ces points dans l'ordre et, pour chaque point p rencontré, on le compare avec ses 64 précédents et 64 suivants dans la liste. De cette façon, on a la garantie que l'on va passer par les 32 points les plus proches en y de P_1 possiblement dans la boîte, et les 32 points les plus proches en y de P_2 possiblement dans la boîte. Si on rencontre deux points à distance moins de d, il n'y a rien à changer, mis à part mettre à jour le d et se rappeler de conserver les deux points minimums.

Pour résumer, la dernière étape de l'algorithme décrit ci-haut se fait comme suit :

- définir $\ell = x_{n/2} + (x_{n/2+1} - x_{n/2})/2$ (prendre $\ell = x_{n/2}$ fonctionnerait aussi) ;
- retirer de P_1 tous les points p_k tels que $x_k < \ell - d$, et retirer de P_2 tous les points p_h tels que $x_h > \ell + d$ (se fait en temps $O(n)$) ;
- soit $P' = [p'_1, \ldots, p'_m]$ la liste des points obtenus en triant les points restants $P_1 \cup P_2$ selon leur coordonnée y (se fait en temps $O(n \log n)$) ;
- pour chaque p'_k dans P', trouver p'_h dans $P'[p'_{k-64}, \ldots, p'_{k+64}]$ qui minimise $dist(p'_k, p'_h)$. Si leur distance est plus petite que le minimum courant, mettre à jour le minimum courant à (p'_k, p'_h) (se fait en temps $O(n)$).

Cette étape prend donc un temps $O(n \log n)$. La récurrence du temps maximum $f(n)$ de cette fonction est

$$f(n) \leq \begin{cases} O(1) & \text{si } n = 1, 2 \\ 2f(n/2) + O(n \log n) & \text{sinon} \end{cases}$$

Selon le théorème maître, $a = b = 2$, $c = k = 1$, et on a $c = \log_b a$. Le temps de cet algorithme est $O(n \log^2 n)$, ce qui est nettement mieux que $O(n^2)$. Il est possible d'améliorer cette complexité à $O(n \log n)$. L'idée est de demander aux appels récursifs de retourner leurs points en ordre des y. L'étape du tri ne prend alors qu'un temps $O(n)$. Les détails sont laissés en exercice.

6.6 Résumé

Ce que vous devez savoir :

— Sachez représenter le temps d'un algorithme récursif par une récurrence.
— Sachez appliquer l'induction sur une récurrence pour démontrer son ordre de grandeur.
— Sachez appliquer le théorème maître.
— Faites les exercices.

Chapitre 7

Algotihmes exact exponentiels

Dans ce chapitre, nous considérons des algorithmes récursifs qui, au lieu de diviser leur entrée, leur *soustrait* un certain nombre d'éléments. Ceci est souvent utile pour entreprendre une force brute sur toutes les solutions possibles. Par contre, ces idées prennent presque toujours un temps exponentiel. Commençons avec le problème de la sous-somme.

7.1 Force brute pour le problème de la sous-somme

Considérons l'exemple suivant. On se rappelle que dans le problème de la sous-somme, on reçoit un ensemble d'entiers $S = \{s_1, \ldots, s_n\}$ et un entier cible t, et on veut savoir s'il existe $S' \subseteq S$ tel que $\sum_{s \in S'} s = t$. On suppose qu'on veut seulement $true$ ou $false$ comme réponse. L'algorithme suivant résout ce problème.

fonction *sousSomme*(S, t)
  si $|S| == 0$ **alors**
    return $false$
  si $t == 0$ **alors**
    return $true$
  return $(sousSomme(S[2..n], t) \vee sousSomme(S[2..n], t - S[1]))$

Ici, le $\vee$ représente le "où" logique. L'idée est que pour $S[1]$, on a deux possibilités : soit cet élément contribue à la somme, ou non. Le premier appel considère que $S[1]$ ne participe pas, et le deuxième considère qu'il participe.

Il n'est pas difficile de voir que cet algorithme prend un temps $O(2^n)$. Ceci est parce que l'algorithme crée un arbre de récursion de profondeur $\theta(n)$ dans le pire cas, et chaque noeud de cet arbre a 2 enfants. Le nombre d'appels récursifs est donc $O(2^n)$, et chaque appel s'exécute en temps $O(1)$, donnant la complexité $O(2^n)$.

Une façon plus générale d'analyser cette compléité qui nous servira à d'autres cas est de faire une analyse de récurrence *homogène linéaire* (voir ci-bas).

C'est-à-dire, si on représente le temps maximum $f(n)$ requis par cet algorithme, et qu'on suppose pour simplifier que $f(0) = 1$, on aura

$$f(n) = \begin{cases} 1 & \text{si } n = 0 \\ 2f(n-1) + O(1) & \text{sinon} \end{cases}$$

Le temps représenté par $f(n)$ peut être décrit comme le nombre de noeuds de l'arbre de récursion correspondant, fois $O(1)$ qui représente le temps passé par noeud. On va se concentrer sur le nombre de noeud de l,arbre de récursion, ce qui en revient à ignorer le terme $O(1)$ de la récurrence dans notre analyse. Soit $g(n)$ le nombre de noeuds de l'arbre de récursion de cet algorithme. Nous avons

$$g(n) = 2g(n-1)$$

avec $g(0) = 1$ comme cas initial. Ceci est une égalité, et on voudrait résoudre pour $g(n)$, i.e. trouver une ou des fonctions qui satisfont cette égalité. On peut y aller par essai et erreur. Si on essaie par exemple avec $g(n) = n$, on a bien sûr $g(n) \neq 2g(n-1)$ car $n \neq 2(n-1)$ en général. Par contre, on pourrait deviner que $g(n) = 2^n$ fonctionne, car $2^n = 2(2^{n-1})$. C'est donc une solution. Toutefois, $c2^n$ est aussi une solution pour toute constante c, car $c2^n = 2(c2^{n-1})$.

En fait, il est possible de démontrer que toutes les solutions sont de la forme

$$g(n) = c2^n$$

Mais que vaut c ? Dans notre cas, ceci ne change rien, car notre algo sera $O(2^n)$. Si on veut vraiment trouver c, on peut utiliser le fait que $g(0) = 1$. Si on remplace n par 0 dans l'expression ci-dessus, on a

$$g(0) = c2^0 = c$$

et puisque $g(0) = 1$, on a $c = 1$. Donc $g(n) = 2^n$ et ceci représente la complexité de notre algorithme.

Fibonacci

La séquence des nombres de Fibonacci est définie par

$$fib(n) = \begin{cases} 0 & \text{si } n = 0 \\ 1 & \text{si } n = 1 \\ fib(n-1) + fib(n-2) & \text{sinon} \end{cases}$$

Les quelques premiers termes de la séquence sont $0, 1, 1, 3, 5, 8, 13, 21$. On va analyser l'algorithme naïf suivant qui calcule le n-ième nombre de Fibonacci :

fonction *calculFib(n)*
  si $n == 0$ *ou* $n == 1$ **alors**
    return n
  return $calculFib(n-1) + calculFib(n-2)$

Le temps maximum $f(n)$ requis par cet algorithme peut s'écrire avec

$$f(n) = \begin{cases} 1 & \text{si } n = 0, 1 \\ f(n-1) + f(n-2) + O(1) & \text{sinon} \end{cases}$$

Encore une fois, on va calculer le nombre de noeuds de l'arbre de récursion et s'attarder à la fonction

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

avec $g(0) = g(1) = 1$.

On cherche une solution à cette égalité. On peut deviner qu'il y a une solution de la forme $g(n) = r^n$, car cette fonction est "probablement" exponentielle. On va donc chercher une solution de cette forme, avec r comme valeur inconnue. S'il y en a une, on aura

$$r^n = r^{n-1} + r^{n-2}$$

On a une égalité à résoudre. En divisant par r^{n-2} de chaque côté, on obtient

$$\begin{aligned} r^2 &= r^1 + r^0 \\ &= r + 1 \end{aligned}$$

ce qui mène à

$$r^2 - r - 1 = 0$$

En trouvant les zéros de $r^2 - r - 1$ (avec le classique $\frac{-b \pm \sqrt{b^2-4ac}}{2a}$, on obtient les deux solutions

$$\begin{aligned} r_1 &= \frac{1+\sqrt{5}}{2} \\ r_2 &= \frac{1-\sqrt{5}}{2} \end{aligned}$$

Mais quel r_i devrait-on prendre pour notre $g(n)$? Il est possible de montrer que toutes les solutions pour $g(n) = g(n-1) + g(n-2)$ ont la forme

$$g(n) = c_1 \left(\frac{1+\sqrt{5}}{2}\right)^n + c_2 \left(\frac{1-\sqrt{5}}{2}\right)^n$$

où c_1 et c_2 sont des constantes pouvant prendre n'importe quelles valeurs. Puisque s'intéresse seulement au O, on peut prendre la plus grosse racine et poser que $g(n) \in O(\left(\frac{1+\sqrt{5}}{2}\right)^n)$, ce qui est environ $O(1.618^n)$.

Si on veut trouver la valeur exacte de c_1 et c_2, il faut utiliser $g(0)$ et $g(1)$. Ceci donne lieu à

$$\begin{aligned} g(0) &= 1 = c_1 + c_2 \\ g(1) &= 1 = c_1 \left(\frac{1+\sqrt{5}}{2}\right) + c_2 \left(\frac{1-\sqrt{5}}{2}\right) \end{aligned}$$

On a deux équations et deux inconnues. On peut trouver $c_1 = \frac{1+\sqrt{5}}{2\sqrt{5}}$ et $c_2 = \frac{-(1-\sqrt{5})}{2\sqrt{5}}$, et donc

$$g(n) = \frac{1+\sqrt{5}}{2\sqrt{5}}\left(\frac{1+\sqrt{5}}{2}\right)^n + \frac{-(1-\sqrt{5})}{2\sqrt{5}}\left(\frac{1-\sqrt{5}}{2}\right)^n$$

ce qu'on peut simplifier à

$$g(n) = \frac{1}{\sqrt{5}}\left(\frac{1+\sqrt{5}}{2}\right)^{n+1} - \frac{1}{\sqrt{5}}\left(\frac{1-\sqrt{5}}{2}\right)^{n+1}$$

En ce qui nous concerne, $g(n) \in O\left(\left(\frac{1+\sqrt{5}}{2}\right)^n\right)$ et donc notre algorithme prend un temps exponentiel (le terme $O(1)$ de f ne change pas cette complexité).

7.2 Recette de résolution de récurrences

La méthodologie ci-haut peut se généraliser à des récurrences de la forme

$$f(n) = a_1 f(n-1) + a_2 f(n-2) + \ldots + a_k f(n-k)$$

où k est une constante et $a_1, \ldots, a_k$ sont aussi des constantes.

On les appelle des *récurrences homogènes linéaires* : homogène parce que tous les termes dépendent de f, et linéaire, car chaque occurrence de f n'est affectée que par une constante. Par exemple, $f(n) = 10f(n-1) + 5f(n-3)$ est homogène et linéaire. Mais $f(n) = 5f(n-1) + 4$ n'est pas homogène à cause du terme 4, et $f(n) = f(n-1)f(n-2)$ n'est pas linéaire. Nous omettons le traitement de récurrences non-homogènes ou non-linéaires, qui amènent leur lot de complexité.

Ici, nous allons nous contenter de donner la recette pour des récurrences homogènes linéaires dans lesquelles $a_1 \geq 0, \ldots, a_k \geq 0$, qui sont les plus fréquentes en algorithmique.

1. chercher une solution de la forme $f(n) = r^n$
2. poser
$$r^n = a_1 r^{n-1} + a_2 r^{n-2} + \ldots + a_k r^{n-k}$$
3. envoyer tous les termes à gauche et diviser par r^{n-k}, ce qui donne
$$r^k - a_1 r^{k-1} - a_2 r^{k-2} - \ldots - a_k = 0$$

4. trouver les racines de cette expression (les zéros) : il y en aura k. Soient $r_1, r_2, \ldots, r_k$ les racines de ce polynome.
5. soit r_i la racine réelle maximum. Alors $f(n) \in O(r_i^n)$.

Notez que si certains des a_i sont négatifs, cette recette fonctionne *sauf* si certaines des racines a_i sont identiques. Dans ce cas, consultez la documentation appropriée !

7.3 Ensemble indépendant maximum dans un graphe

Soit G un graphe. Un sous-ensemble $X \subseteq V(G)$ est appelé un *ensemble indépendant* si, pour tout $u, v \in X$ distincts, $uv \notin E(G)$. On va donc s'intéresse au problème de trouver un ensemble indépendant de taille maximum dans un graphe.

On peut créer un algorithme qui effectue une force brute sur chaque sous-ensemble, et qui retourne le sous-ensemble maximum qui est un ensemble indépendant. Ceci prendre un temps $O(2^n \cdot n^c)$, avec $c \in O(1)$. On peut toutefois faire mieux.

Considérez l'algorithme suivant. Ici, $G - v$ dénote le graphe duquel on retir v, et $G - N(v) - v$ le graphe duquel on retire v et tous ses voisins $N(v)$.

fonction *indset*$(G = (V, E))$
  si $|E| = 0$ **alors**
    return V
  Soit v un sommet de G de degré maximum
  $I_1 = indset(G - v)$
  $I_2 = indset(G - N(v) - v) \cup \{v\}$
  si $|I_1| > |I_2|$ **alors**
    return I_1
  sinon
    return I_2

Le principe de l'algorithme est que pour le v choisi, il y a 2 cas. Soit v est dans la solution optimale, ou bien v n'est pas dans la solution optimale. Si v n'y est pas, on n'a qu'à retirer v et trouver une solution optimale sur $G - v$. Si v est dans la solution optimale, alors on ne peut pas ajouter ses voisins (par définition d'un ensemble indépendant). Dans ce cas, il faut chercher une solution sur $G - N(v) - v$.

Pour la complexité, on note qu'il faut un temps $O(n)$ dans chaque appel pour trouver v et calculer $G - v$ et $G - N(v) - v$. L'appel $G - v$ se fait

sur un graphe avec $n-1$ sommets. On note que le v choisi a au moins un voisin, car sinon on aurait $|E| = 0$. Ceci veut dire que $G - N(v) - v$ a $n-2$ sommets ou moins. La complexité de cet algorithme est donc représenté par la récurrence

$$f(n) \leq f(n-1) + f(n-2) + O(n)$$

on a déjà vu que le nombre de noeuds de l'arbre de récursion de cette fonction est $O(1.618^n)$. Le temps total de l'algorithme est borné par $O(1.618^n \cdot n)$.

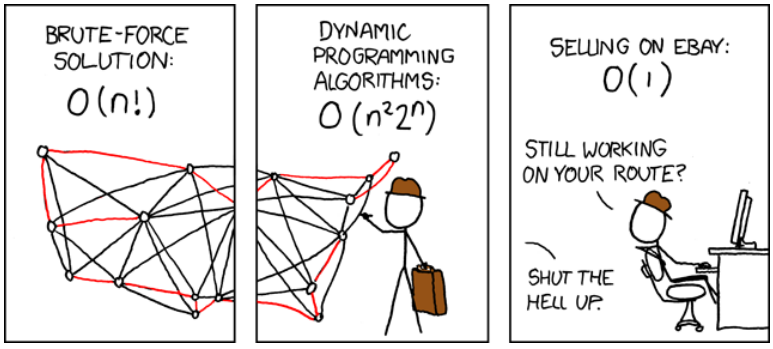
7.4 Résumé

Ce que vous devez savoir :

— Sachez reconnaître les récursions qui "divisent" versus celles qui "soustraient".
— Sachez exprimer la récurrence d'un algorithme récursif exponentiel.
— Sachez évaluer une telle récurrence.
— Sachez designer de "bons" algorithmes récursifs exponentiels. Chaque appel devrait soustraire autant que possible.

Chapitre 8

Programmation dynamique



Le principe de la programmation dynamique est de calculer une solution à une instance I en fonction des solutions à des sous-instances $I_1, \ldots, I_k$. Ceci peut se faire de façon récursive ou non. D'un point de vue, on commence par vouloir calculer I, puis on calculer récursivement $I_1, \ldots, I_k$, et on "combine" ces sous-solutions. Ceci s'apparente à l'approche diviser-pour-régner. Par contre, la programmation dynamique peut être vue comme plus générale, car elle n'exige pas de "diviser" I. Il faut seulement être en mesure de définir une solution à I en fonction de sous-instances pertinentes, divisées ou non. De plus, un aspect important de la programmation dynamique est qu'une fois qu'une solution $sol(I)$ est calculée pour une instance I, cette solution est conservée en mémoire pour réutilisation future. L'optimisation de l'algorithme de Fibonacci présenté plus bas illustre bien l'avantage de cette approche.

En programmation dynamique, on peut aussi faire l'inverse : on commence par calculer des solutions pour des petites sous-instances, et on utilise ces solutions pour calculer des instances de plus en plus grandes. L'approche récursive est dite *top-down* (de haut en bas) tandis que la programmation dynamique est dite *bottom-up* (de bas en haut).

Le mot "programmation" ici ne fait pas référence à l'écriture de code. Le terme réfère plutôt à une programmation comme dans un événement, ou plus généralement à une liste ou une table d'informations. Le lien avec la technique présentée ici est que l'on va presque toujours construire une table (une "programmation") de solutions rencontrées jusqu'à maintenant, et cette table va grandir de façon dynamique.

8.1 Fibonacci amélioré

Revenons avec l'exemple du calcul de la fonction de Fibonacci. Considérez l'algorithme suivant, qui implémente l'idée du chapitre précédent, mais qui stocke chaque valeur de $f(n)$ calculée dans une variable globale F pour réutilisation future. On suppose que F est un dictionnaire dans lequel $F[n] = nil$ si n n'est pas une clé.

```
global F = dict()
fonction calculFib2(n)
    si n == 0 ou n == 1 alors
        return n
    si F[n] ≠ nil alors
        return F[n]
    F[n] = calculFib2(n − 1) + calculFib2(n − 2)
    return F[n]
```

Combien de noeuds est-ce que l'arbre de récursion de cet algorithme aura ? On voit qu'une chaîne d'appels récursifs "gauches" seront d'abord faits pour $n, n-1, \ldots, 2, 1, 0$. Puisque chaque appel stocke la valeur de Fibonacci, le deuxième appel de chaque récursion tombera sur le cas où $F[n] \neq nil$. Donc, il y a une chaîne de n appels gauches, puis chaque appel droit est une feuille de l'arbre de récursion. Donc, il y a au plus $2n$ noeuds de récursion, et chaque appel est $O(1)$. Le temps de l'algorithme est $O(n)$, nettement mieux que $O(1.618^n)$.

Il existe aussi une version itérative de cet algorithme. Il suffit de calculer les F en ordre croissant.

```
fonction calculFib2(n)
    F = [0] * n                    // F est un tableau indicé à 0
    F[0] = 1
    F[1] = 1
    pour i = 2..n faire
        F[i] = F[i − 1] + F[i − 2]
    fin
    return F[n]
```

Il est facile de voir que cet algorithme prend un temps $O(n)$.

Bien que simple, cet algorithme illustre les bases de la programmation dynamique. Au lieu d'essayer de calculer $fib(n)$ tout de suite, on comment par calculer $F[2]$ et on mémorise sa valeur. Puis on calcule $F[3]$ en réutilisant les valeurs de F calculées auparavant, puis $F[4]$ en utilisant les valeurs précédentes de F, et ainsi de suite. Notez qu'en principe, on pourrait se contenter de conserver les deux valeurs précédentes au lieu de tout le tableau.

On voit ici l'idée de la programmation dynamique : on construit une table des valeurs de Fibonacci pour des valeurs de plus en plus grandes. Chaque calcul d'une nouvelle valeur se base sur les calculs précédents jusqu'à ce

qu'ont ait réussi à calculer $fib(n)$, la valeur désirée.

8.2 Faire de la monnaie avec un minimum de pièces

Dans le problème de la monnaie, on a un ensemble de valeurs de pièces $P = [p_1, \ldots, p_n]$ et un somme de monnaie à remettre t. On cherche à donner un minimum de pièces utilisant les valeurs de P de façon à sommer à t. Par exemple, dans notre monde, $P = \{200, 100, 25, 10, 5\}$. Si $t = 425$, on peut remettre 2×200, 2×10 et 1×5, pour un total de 5 pièces.

En général, on ne sait même pas comment déterminer s'il est possible d'atteindre un somme de t avec les dénominations de P. Ce problème s'apparente au problème de la sous-somme, à la différence que l'on peut utiliser une valeur p_i autant de fois que l'on veut et que l'on veut minimiser le nombre d'éléments utilisés dans la somme.

On peut résoudre ce problème par programmation dynamique en temps $O(tn)$. L'idée est la suivante : on construit une table S de solutions de taille t. Pour chaque $i \in \{1, \ldots, t\}$, $S[i]$ contient le nombre minimum de pièces pour atteindre une somme de i. Chaque $S[i]$ se calcule avec les $S[j]$ précédents, lorsque possible. La question à se poser est : si on sait comment avoir une somme de j pour chaque $j = 1, \ldots, i-1$, comment avoir une somme de i ? On peut soit atteindre une somme de $i - p_1$ et ajouter une pièce de valeur p_1, ou avoir une somme de $i - p_2$ et ajouter une pièce de valeur p_2, etc.

On peut donc exprimer $S[i]$ par

$$S[i] = \min_{p_j \in P}(S[i - p_j] + 1)$$

Il y a tout de même des cas d'exception à gérer. Si $i < 0$, on définit $S[i] = \infty$ car atteindre $i < 0$ est impossible, et si $i = 0$, on a $S[i] = 0$ car on n'a besoin d'aucune pièce.

On peut implémenter cette récurrence directement avec un algorithme récursif.

```
global S = dict()
fonction minPiecesRec(P = [p_1, ..., p_n], i)
    si i < 0 alors return ∞
    si i = 0 alors return 0
    si S[i] ≠ nil alors return S[i]

    smin = ∞
    pour p_j ∈ P faire
        si S[i − p_j] + 1 < smin alors
            smin = S[i − p_j] + 1
    fin
    S[i] = smin
    return S[i]
```

En ce qui concerne la complexité de cet algorithme, on peut que le temps est proportionnel au nombre d'entrées de S à remplir, fois le temps requis pour remplir une telle entrée. Rappelons que chaque $S[i]$ n'est calculé qu'une fois. Le nombre d'entrée de S à remplir est $O(t)$, et le temps pour en calculer une est $O(n)$, et la complexité est donc $O(nt)$.

Voici l'algorithme itératif qui calcule cette récurrence. Il suffit de calculer les i en ordre croissant.

```
fonction minPieces(P = [p_1, ..., p_n], t)
    S = [∞] ∗ t
    S[0] = 0
    pour i = 1 .. t faire
        pour p_j ∈ P faire
            si i − p_j ≥ 0 et S[i − p_j] + 1 < S[i] alors
                S[i] = S[i − p_j] + 1
        fin
    fin
    return S[t]
```

Cet algorithme s'exécutera en temps $O(nt)$. Il n'est pas très difficile de voir que cet algorithme fonctionne correctement, mais prenons le temps de le démontrer. On utilise la version itérative, mais la preuve pour la version récursive serait identique.

Théorème 12. *Après l'exécution de l'algorithme minPieces, pour tout $i = 1..t$, $S[i]$ contient le nombre minimum de pièces requis pour obtenir une somme de i.*

Démonstration. On peut montrer par induction sur $i = 1..t$ que lorsque $S[i]$ est affecté pour la dernière fois, sa valeur correspond au nombre minimum de pièces pour avoir la somme i (et $S[i] = \infty$ si la somme est impossible à atteindre). On peut prendre $i = 0$ comme cas de base, et il est facile de voir que $S[0] = 0$ est correct.

Pour l'induction, on suppose que $S[j]$ contient la bonne valeur pour tout $j < i$. Supposons d'abord qu'il est possible d'atteindre une somme de i. Considérons une solution optimale R de pièces qui somment à i. Soit p_k une pièce de R, n'importe laquelle. Quand on enlève p_k de R, on obtient un ensemble de pièces R' qui somment à $i - p_k$. Il faut que R' contienne le minimum de pièces dont la somme est $i - p_k$, car sinon on pourrait remplacer R' par une solution R'' ayant moins de pièces et qui somme à $i - p_k$, pour ensuite ajouter p_k à R'' et obtenir une solution meilleure que R. Par hypothèse d'induction, le nombre de pièces dans R' est $S[i - p_k]$, et donc il y a $S[i - p_k] + 1$ pièces dans R. On peut voir que notre algorithme affectera $S[i] = S[i-p_k]+1$ lorsqu'il considérera la pièces p_k dans la boucle imbriquée. On a donc démontré que $S[i]$ contient la bonne valeur.

Supposons maintenant que le montant de i est impossible à atteindre. Donc pour tout $p_j \in P$, il est impossible d'atteindre un montant de $i - p_j$. Par induction, $S[i - p_j] = \infty$ pour tout p_j, et donc l'algorithme prendra le minimum parmi des ∞, et on aura $S[i] = \infty$, ce qui est aussi correct. □

En particulier, la preuve ci-haut montre que $S[t]$ représente bien le nombre de pièces minimum pour sommer à t.

Comment retrouver une solution concrète

L'algorithme ci-haut ne donne que le nombre optimal de pièces, et non pas un ensemble de pièces à donner. On peut retrouver une solution à partir des valeurs de S. Ceci est typique des solutions de programmation dynamique : on commence par calculer une table de valeurs optimales, puis à partir de la valeur qui nous intéresse, on retrace le chemin qui a mené à notre solution. En anglais, cette procédure s'appelle le *backtracking*. Dans notre cas, si on reçoit P, t ainsi que S, le backtracking peut se faire en retrouvant quelle entrée de S a mené à $S[t]$, puis quelle entrée a mené à cette entrée, et ainsi de suite.

```
fonction minPiecesBacktracking(P = [p_1, ..., p_n], t, S)
    R = ∅                                          // Valeur de retour
    cur = t                                // Position courante dans S
    tant que cur > 0 faire
        pour p ∈ P faire
            si S[cur − p] = S[cur] − 1 alors
                R.append(p)
                cur = cur − p
                break                              // Sortir du tant que
        fin
    fin
    return R
```

Notez aussi que dans la procédure *minPieces*, on aurait aussi pu stocker, dans un tableau alternatif Q, pour chaque i la somme $i - p$ qui a mené à l'optimal $S[i]$. De cette façon, le backtracking serait accéléré d'un facteur linéaire.

Complexité de l'algorithme : est-ce polynomial ?

Le problème du minimum de pièces est NP-complet, c'est-à-dire qu'il n'y a aucun algorithme connu pour résoudre le problème qui prend un temps polynomial par rapport à la taille de l'entrée. Mais on vient de développer un algorithme en temps $O(nt)$. C'est polynomial, non ?

Ceci dépend de la valeur de t. Par exemple, si $t = 2^n$, alors cet algorithme prendra un temps exponentiel (par rapport à n). En fait jusqu'à maintenant, nous n'avons jamais vraiment spécifié ce que voulait dire "la taille de l'entrée". On peut évaluer la complexité par rapport à n, mais formellement, la taille de l'entrée du problème ci-haut dépend du nombre de bits nécessaires pour représenter $p_1, \ldots, p_n$ et t. Rappelons qu'on peut représenter un entier n avec $\lceil \log n \rceil$ bits. Donc si on suppose que $t \geq p_i$ pour $i \in \{1, \ldots, n\}$, la taille de l'entrée à notre problème ne dépassera jamais $(n+1)\lceil \log t \rceil$. Donc encore une fois, si $t = 2^n$, la taille de l'entrée est $O(n^2)$ et la complexité $O(nt) = O(n2^n)$ est exponentielle par rapport à la taille de l'entrée.

Ces détails importent en théorie des langages lorsque l'on veut classer les problèmes dans des classes de complexités. Dans le cadre de ce cours, nous allons étudier la complexité par rapport à des paramètres bien définis plutôt qu'aux tailles d'entrée, mais il est sain de garder en tête ces éléments.

8.3 Principe d'optimalité de Bellman

On associe souvent la programmation dynamique à ce qu'on appelle le principe d'optimalité de Bellman. Ce principe s'applique seulement à certains problèmes et dit, de façon informelle : une solution optimale d'un problème s'obtient en combinant des solutions optimales à des sous-problèmes.

Si un problème satisfait ce principe, l'idée est donc de construire des solutions optimales pour des sous-instances, et ces solutions peuvent ensuite être combinées à des instances plus grandes. Souvent, on peut écrire que l'optimal à une instance X peut être calculé en considérant certaines sous-solutions que l'on modifie pour obtenir une solution à X. Beaucoup de ces problèmes peuvent s'écrire comme

$$OPT(X) = \min_{Y}(OPT(Y) + \text{ “quelque chose”})$$

où le minimum est pris sur un ensemble de sous-instances choisies adéquatement (i.e. les Y). Nous avons vu un exemple de ce phénomène avec le problème de la monnaie, où on avait

$$S[i] = \min_{p \in P}(S[i-p] + 1)$$

(en omettant les cas limite $i \in P$ et $i - p < 1$).

Bien sûr, lorsque l'on a un problème à résoudre, on doit d'abord vérifier qu'il satisfait ce principe. Lorsque c'est possible, on veut aussi minimiser le nombre de sous-solutions Y à considérer. Il faut parfois être créatif dans notre formulation de l'optimalité pour expliciter le principe.

8.4 Ensemble indépendant dans un arbre

Nous allons considérer un exemple de problème dans lequel il faut tabuler plusieurs solutions possibles pour une même sous-instance.

Soit $G = (V, E)$ un graphe. Un sous-ensemble $X \subseteq V$ est appelé *indépendant* si, pour toute paire de sommets $x, y \in X$, $xy \notin E$. Donc aucune paire de sommets dans X ne partagent une arête. Nous sommes souvent intéressés à trouver un ensemble indépendant de *poids* maximum (contrairement au *nombre* comme dans le chapitre précédent). Les ensembles indépendants servent à modéliser des ensembles d'éléments qui ont l'absence de conflits, par exemple si on veut trouver un maximum d'activités qui ne se chevauchent pas.

Entrée :un *arbre enraciné* $T = (V, E, w)$ où $w : V \to \mathbb{R}^{\geq 0}$ attribue un poids à chaque sommet, et la racine r de l'arbre ;

Sortie : un ensemble indépendant $X \subseteq V$ qui maximise $\sum_{x \in X} w(x)$.

En général, trouver l'ensemble indépendant de taille maximum est NP-complet, mais pas sur les arbres. Si on a un arbre *enraciné* $T = (V, E)$, on peut utiliser la programmation dynamique sur T pour avoir une solution optimale à chaque noeud.

C'est-à-dire, pour $v \in V$, soit $T[v]$ le sous-arbre enraciné en v (i.e. l'arbre qu'on obtient en conservant seulement v et tous ses descendants). En particulier, si v est une feuille, $T[v]$ n'a qu'un seul sommet. On dénote par $M_1[v]$ la taille de l'ensemble indépendant maximum dans $T[v]$ avec la contrainte que l'ensemble contienne v. Similairement, $M_0[v]$ dénote la taille de l'ensemble indépendant maximum dans $T[v]$ avec la contrainte que v n'est *pas* dans l'ensemble. Si r dénote la racine, la valeur que l'on voudrait est $\max(M_0[r], M_1[r])$.

On applique le principe d'optimalité comme suit : soit v la racine du sous-arbre d'intérêt. Il y a deux cas possibles pour un ensemble indépendant X .

1. Si $v \notin X$, chaque enfant v_i de v peut être dans X ou non. Dans ce cas, on prend la solution optimale pour chaque enfant v_i, sans contrainte.
2. Si $v \in X$, aucun enfant de v ne peut être dans X. Dans ce cas, pour chaque enfant v_i on prend la solution optimale pour $T[v_i]$ avec la contrainte que v_i n'est pas dans l'ensemble.

Si on dénote par $enf(v)$ l'ensemble des enfants de v, on peut exprimer les cas ci-haut mathématiquement comme suit :

$$M_0[v] = \sum_{v_i \in enf(v)} \max(M_0[v_i], M_1[v_i])$$
$$M_1[v] = w(v) + \sum_{v_i \in enf(v)} M_0[v_i]$$

Le "$+w(v)$" de $M_1[v]$ vient du fait qu'on doit compter v lui-même comme faisant partie d'une solution. Si v est une feuille, alors $M_0[v] = 0$ et $M_1[v] = w(v)$.

Il n'est pas difficile d'imaginer une procédure récursive pour implémenter cette récurrence. Concrètement, cet algorithme va descendre l'arbre récursivement

et calculer M_0 et M_1 aux feuilles, puis aller graduellement vers la racine. Notez que si on vous donne un arbre non-enraciné, vous pouvez choisir n'importe quel sommet et l'appeler la racine.

fonction *maxIndep*$(T = (V, E), v)$
  //On suppose que M_0 et M_1 sont des variables globales
  $M_0[v] = 0$
  $M_1[v] = 1$
  pour $v_i \in enf(v)$ **faire**
    $maxIndep(T, v_i)$ `// Remplira` $M_0[v_i]$ `et` $M_1[v_i]$
    $M_0[v] += \max(M_0[v_i], M_1[v_i])$
    $M_1[v] += M_0[v_i]$
  fin
  return $\max(M_0[v], M_1[v])$

Nous laissons en exercice la démonstration que cet algorithme fonctionne. L'idée est que puisque T est un arbre, tout sous-arbre $T[v]$ est tel que ses sous-arbres enfants ne partagent pas d'arêtes. On peut donc prendre les solutions de chaque enfant séparément et les combiner. Ceci ne fonctionne pas sur un graphe en général. Un autre exercice est de développer une procédure de backtracking pour reconstruire un ensemble indépendant concret.

Pour ce qui est de la complexité, exprimer cet algorithme par une récurrence n'est pas facile, car le nombre de sous-appels dépend du degré de la racine courante. On peut toutefois voir que chaque appel récursif à *maxIndep* se fait avec un v différent. Le temps passé dans chaque appel sur racine v prend un temps $O(deg(v))$. On peut donc se convaincre que le temps de cet algorithme est $O(n + m) = O(n)$ (parce que dans un arbre, $m = n - 1$, vous vous rappelez ?).

Reconstruire une solution à partir de M_0 et M_1

Le backtracking pour ce problème est plus subtil que le précédent. L'idée est la suivante. On démarre de la racine r. Si $M_0[r] > M_1[r]$, alors r n'est *pas* dans une solution optimale, et sinon, r est dans une solution optimale.

On démarre une procédure récursive de construction de solution à partir de r, en lui disant si r doit être présent ou non (avec le booléen $useM_0$). Selon le cas, on décide si les enfants doivent être présent ou non, et on continue résursivement.

```
fonction getSol(T = (V, E), M_0, M_1, v, useM_0)
    //Appel initial avec getSol(T, M_0, M_1, r, (M_0[r] > M_1[r]))
    //I est une variable globale à retourner
    si useM_0 alors
        pour w ∈ enf(v) faire
            getSol(T, M_0, M_1, w, (M_0[w] > M_1[w]))
        fin
    sinon
        I.append(v)
        pour w ∈ enf(v) faire
            //v est inclut, donc impossible d'inclure w
            getSol(T, M_0, M_1, true)
        fin
    fin
```

8.5 Algorithme de Dijkstra

On se rappelle que la fouille en largeur dans un graphe nous permet de trouver les distances à partir d'un sommet u jusqu'à n'importe quel autre sommet. Cette approche ne fonctionne que sur des graphes non-valués. Dans le cas de graphes valués (i.e. avec des poids sur les arêtes), on a besoin de travailler un peu plus. Rappelons que sur un graphe valué, le poids d'un chemin est la somme des valeurs des arêtes sur le chemin, et $dist(u, v)$ est le poids minimum d'un chemin entre u et v. Notons que dans un graphe non-orienté, $dist(u, v) = dist(v, u)$.

L'algorithme de Dijkstra permet, pour un sommet quelconque u d'un graphe $G = (V, E, f)$, de trouver $dist(u, v)$ pour tout $v \in V$ (en particulier, $dist(u, u) = 0$). On doit supposer que G n'a pas d'arête avec un poids négatif, et on suppose que $f(v, w) = \infty$ si $vw \notin E$. Dans ce cas, on peut appliquer le principe d'optimalité : si w se trouve sur le chemin optimal entre u et v, alors on doit prendre le chemin optimal entre u et w, puis le chemin optimal entre w et v. Un cas spécifique de ce principe implique que

$$dist(u, v) = \min_{w \in V \setminus \{v\}} (dist(u, w) + f(w, v))$$

Le principe derrière Dijkstra est de calculer les distances de u vers de sommets de $X \subseteq V$ pour des sous-ensembles X de plus en plus grands. On commence avec $X = \{u\}$ et on se rappelle que $dist(u, u) = 0$. On note aussi

dans une table D, pour chaque $v \in V$, la distance minimum entre u et v dans le sous-graphe induit $G[X \cup \{v\}]$. Ensuite, on ajoute à X le sommet $v \in V \setminus X$ tel que $D[v]$ est minimum. On met à jour D, et on continue. Pour fins de backtracking (que l'on ne fera pas ici), chaque fois qu'on ajoute un sommet w dans X, on se rappelle dans la table P du sommet dans X qui précède w dans un chemin optimal entre u et w.

fonction *dijkstra*$(G = (V, E, f), u)$
  //On suppose que $V = \{1, \ldots, n\}$
  Ajouter u à X
  $R = V \setminus \{u\}$ `// Sommets restants`
  $D = [\infty] * n$
  $D[u] = 0$
  $P = [0] * n$ `// Tableau des précédents`
  pour chaque *voisin* v *de* u **faire**
    $D[v] = f(u, v)$
  fin
  tant que R *est non-vide* **faire**
    $v =$ le sommet de R qui minimise $D[v]$
    Ajouter v à X
    Enlever v de R
    pour chaque *voisin* w *de* v **faire**
      si $w \notin X$ **alors**
        $d = D[v] + f(v, w)$
        si $d < D[w]$ **alors**
          $D[w] = d$
          $P[w] = v$
    fin
  fin

Nous vous invitons à exécuter cet algorithme sur des exemples. Il peut sembler superflu de maintenir à la fois R et X. Une des raisons est que l'on doit faire deux types d'opérations : trouver le sommet de R qui minimise $D[v]$, et vérifier si $w \in X$. Tel que sera discuté ci-bas, il est plus efficace d'implémenter R comme un file de priorité (par exemple avec un monceau) et X comme un ensemble (un simple tableau de booléens).

Exactitude de l'algorithme. Comment savoir que cet algorithme fonctionne correctement ? On va prouver qu'en tout temps, pour tout sommet x dans X, $D[x] = dist(u, x)$. Puisque $X = V$ à la fin de l'algorithme, ceci

voudra dire que l'on a bien calculé toutes les distances les plus courtes à partir de u. On le démontre par induction sur X. Comme cas de base, on prend $X = \{u\}$ et il est trivial de voir que D contient la distance minimum de u à chaque sommet de X.

Pour l'induction, on suppose que X contient plus d'un élément et que $D[x] = dist(u, x)$ pour chaque $x \in X$. Considérons le moment où l'algorithme ajoute un sommet v à X, et soit $w = P[v]$. Puisqu'on ajoute v dans X à ce moment, on veut montrer que $dist(u, v) = D[v]$. La valeur de $D[v]$ correspond au poids du chemin minimum de u à w suivi du poids de l'arête $f(w, v)$. Si la valeur de $D[v]$ est erronée, c'est que $dist(u, v) < D[v] = D[w] + f(w, v)$. Soit C un chemin minimum de u à v et supposons que le poids de ses arêtes est plus petit que $D[w]+f(w, v)$. Soit z le premier sommet de C en dehors de X et y le sommet qui précède z sur C (donc $y \in X$). La somme des arêtes sur C est au moins $dist(u, y) + f(y, z)$. Par induction, ceci est égal à $D[y]+f(y, z)$, et on a supposé que $D[y]+f(y, z) < D[w]+f(w, v)$. Si $z \neq v$, on a une contradiction, car l'algorithme aurait alors choisi d'inclure z au lieu de v, par la minimalité du choix. Si $z = v$, on a une autre contradiction, car l'algorithme aurait mis à jour $D[v]$ au moment où y a été ajouté (en mettant $P[v] = y$). On déduit donc qu'il n'existe pas de chemin meilleur de u à v que celui de passer par w et aller à v, et donc la valeur de $D[v] = D[w] + f(w, v)$ est correcte.

Complexité. En supposant que $V = \{1, \ldots, n\}$, on peut implémenter X comme un tableau de n booléens pour savoir quel sommet s'y trouve ou non. L'ajout à X et la vérification $w \in X$ peuvent se faire en temps constant. Puisque chaque v est ajouté une seule fois dans X, le nombre total de fois que l'on passera dans la boucle imbriquée sera la somme des degrés, donc $O(m)$ (c'est l'argument habituel).

La seule instruction qui peut prendre un temps plus difficile à analyser est de trouver le sommet de R qui minimise $D[v]$. Si on l'implémente de façon naïve, on trouvera ce commet en temps $O(n)$ et la complexité totale deviendra $O(n^2)$. Une autre option est de gérer R comme une *file de priorité*. Peut-être vous rappellerez-vous qu'un monceau permet d'insérer/supprimer des éléments en temps $O(\log n)$, de modifier des valeurs de priorité en temps $O(\log n)$ et de trouver l'élément le plus prioritaire en temps $O(1)$. On peut affecter à chaque élément v de R la priorité $D[v]$ et la mettre à jour au besoin. L'effet sera que les instructions impliquant R prendront un temps $O(\log n)$. La complexité devient donc $O((m + n) \log n)$.

À noter qu'avec un *monceau Fibonacci*, la mise à jour d'une priorité peut

se faire en temps $O(1)$. L'analyse de l'algorithme devient donc $O(m+n \log n)$.

Version utilisant une file de priorité. Voici une version de l'algorithme qui utilise une file de priorité tel que discuté ci-haut.

```
fonction dijkstra2(G = (V, E, f), u)
    Ajouter u à X
    R = new Monceau()
    D = [∞] * n
    D[u] = 0
    P = [0] * n                          // Tableau des précédents
    pour chaque v ∈ V \ {u} faire
        si v est un voisin de u alors
            D[v] = f(u, v)
            R.ajouter(v, D[v])
        sinon
            R.ajouter(v, ∞)
    fin
    tant que R est non-vide faire
        v = R.getMin()
        R.supprimerMin()
        Ajouter v à X
        pour chaque voisin w de v faire
            si w ∉ X alors
                d = D[v] + f(v, w)
                si d < D[w] alors
                    D[w] = d
                    P[w] = v
                    R.updatePriorite(w, D[w])
        fin
    fin
```

8.6 Algorithme de Floyd-Warshall

Alors que l'algorithme de Dijkstra trouve les distances à partir d'une seule source, l'algorithme de Floyd-Warshall trouve les distances les plus courtes entre *chaque* paire de sommet. On suppose donc qu'on reçoit un graphe valué $G = (V, E, f)$ avec $V = \{1, 2, \ldots, n\}$ et on veut calculer une

table $D[u, v] = dist(i, j)$ pour tout $i, j \in V$ (on utilise i, j, k dans cette section pour mettre l'emphase sur le fait que les sommets sont représentés par des entiers).

L'idée est la suivante : soit $k \in \{0, 1, 2, \ldots, n\}$. On définit par $dist(i, j, k)$ la longueur du chemin minimum entre i et j qui passe seulement par des sommets dans $\{i, j\} \cup \{1, 2, \ldots, k\}$. Donc on ne se donne droit qu'aux k premiers sommets comme intermédiaires. Le but est de calculer $d(i, j, n)$ pour tout $i, j \in V$. En particulier, $dist(i, j, 0)$ ne permet aucun intermédiaire, et on aura $dist(i, j, 0) = f(i, j)$.

Pour $k > 0$, il suffit de constater qu'il n'y a que deux possibilités pour un chemin optimal C entre i et j qui passe par $\{1, 2, \ldots, k\}$: (1) soit le sommet k fait partie du chemin C ; ou (2) soit il n'en fait pas partie.

Dans le cas (1), il suffit de prendre le chemin optimal de i à k qui n'a droit qu'à des sommets dans $\{1, 2, \ldots, k-1\}$, suivi du chemin optimal de k à j qui n'a droit qu'à des sommets dans $\{1, 2, \ldots, k-1\}$. Dans le cas (2), ceci veut dire que C n'utilise que les sommets $\{1, 2, \ldots, k-1\}$ comme intermédiaires entre i et j. Ceci peut s'exprimer de la façon suivante :

$$dist(i, j, k) = \min\left(dist(i, k, k-1) + dist(k, j, k-1), dist(i, j, k-1)\right)$$

Notez que dans le cas (1), il n'est pas interdit que les chemins de i à k et de k à j intersectent. Toutefois, si c'est le cas, ceci veut dire qu'il y a un chemin plus court parmi $\{1, 2, \ldots, k-1\}$ et celui-ci sera trouvé par notre fonction de minimisation. Cette fonction nous donne un algorithme de façon presque directe : on n'a qu'à calculer $dist(i, j, k)$ pour des valeurs de k de plus en plus grandes.

```
fonction floydWarshall(G = (V, E, f))
    //On suppose V = {1, 2, ..., n}
    //On suppose f(u, u) = 0 et f(u, v) = ∞ si uv ∉ E` ;
     dist = tableau de dimension n × n × n
    pour chaque i ∈ V faire
        pour chaque j ∈ V faire
            dist[i, j, 0] = f(i, j)
        fin
    fin
    pour chaque k = 1..n faire
        pour chaque i = 1..n faire
            pour chaque j = 1..n faire
                dist[i, j, k] =
                  min(dist[i, k, k − 1] + dist[k, j, k − 1], dist[i, j, k − 1])
            fin
        fin
    fin
    return dist
```

Il y a trois boucles imbriquées, chacune s'exécutant $O(n)$ fois, et la complexité est $O(n^3)$.

À noter que la troisième dimension de *dist* ci-haut n'est pas requise. On pourrait avoir un *dist* comme un tableau $n \times n$, initialiser $dist[i, j] = f(i, j)$ dans la première boucle. Dans la boucle principale, on aurait $dist[i, j] = \min(dist[i, k] + dist[k, j], dist[i, j])$ et le tout fonctionnerait quand même. L'avantage serait un gain d'un facteur n en espace. La troisième dimension est donc présente à des fins conceptuelles.

8.7 Résumé

Ce que vous devez savoir :

— Sachez prouver qu'un algorithme utilisant les principes de Bellman fonctionne.
— Sachez concevoir des algorithmes utilisant la programmation dynamique.
— Sachez concevoir un algorithme semblable à maxIndep.
— Comprenez Dijkstra, Floyd-Warshall et leurs différences.
— Faites les exercices.

Chapitre 9

Algorithmes probabilistes

Les algorithmes considérés jusqu'à maintenant sont appelés des algorithmes *déterministes* : si on fait deux exécutions sur la même entrée, la suite d'instructions qui sera exécutée sera toujours la même, et elle est donc pré-déterminée.

Il est intéressant de se demander ce qu'on peut accomplir si on permet à l'algorithme d'effectuer des choix *aléatoires* (par exemple, en prenant une décision de brancher ou non dans un "if" selon le résultat d'un lancer de pièce ou de dés). Dans ce cas, deux exécutions sur une même entrée peuvent mener à des suites d'instructions différentes. Ceci donne lieu à ce qu'on appelle des algorithmes *probabilistes*. De façon intuitive, ces algorithmes devraient être plus "puissants" que les algorithmes déterministes, puisque ces derniers sont des cas spéciaux d'algorithmes probabilistes qui ne font aucun choix aléatoire.

On considère deux situations qui peuvent se produire avec un algorithme probabiliste sur une même entrée. Il est possible qu'une exécution donne la bonne sortie et qu'une autre donne la mauvaise sortie. Ou bien, il est possible que toutes les exécutions donnent la bonne sortie, mais que les temps d'exécution diffèrent selon les choix aléatoires effectués.

Nous allons d'abord nous attarder au deuxième cas, dans lequel nous nous intéressons à calculer le *temps espéré*, qui représente la moyenne des temps selon les choix aléatoires faits.

9.1 Espérance en temps

Prenons un exemple artificiel d'algorithme aléatoire. On suppose qu'on a accès à une fonction $lancerPiece$, qui retourne *pile* avec probabilité 1/2

et $face$ avec probabilité 1/2.

fonction *algoBidon(n)*
  $r = lancerPiece()$
  si $r = face$ **alors**
    $t = n^2$
  sinon
    $t = n$
  $i = 1$
  tant que $i < t$ **faire**
    $i = i + 1$
  fin

Cet algorithme est très bidon. Mais combien de temps peut-on s'attendre à ce qu'il prenne ? On voit qu'il y a deux chemins d'exécution possible : l'un prend un temps n^2 et l'autre un temps n. Le premier cas arrive arrive probabilité 1/2 et le second aussi. On dénote par $t(algoBidon, n)$ le temps requis par l'algorithme sur entrée n. Bien sûr, $t(algoBidon, n)$ n'a pas de valeur unique : elle dépend des choix faits par l'algorithme. Le temps est donc une variable aléatoire et on peut y associer une distribution de probabilité. Le mieux qu'on peut faire dans ce cas est d'analyser la moyenne.

Donc en moyenne, on peut s'attendre à un temps d'exécution

$$\frac{1}{2}n^2 + \frac{1}{2}n$$

ce qui est égal à $\frac{n^2+n}{2}$. Ce qu'on vient de calculer s'appelle le *temps espéré* de l'algorithme, et ici on dira que l'espérance est $O(n^2)$.

Plus généralement, un algorithme A peut avoir beaucoup de chemins d'exécutions possibles. Par exemple, si l'algorithme lance k pièces, alors il peut y avoir jusqu'à 2^k chemins d'exécution possibles. On écrit $t(A, n)$ pour le temps pris par l'algorithme sur entrée de taille n, qui est une variable aléatoire.

Dénotons par X l'ensemble de toutes les séquences d'instructions qui peuvent être exécutées par un algorithme aléatoire. Donc $t(A, n)$ dépend du choix dans X qui est fait. On écrit $t(A, n, x)$ pour le temps pris par l'algorithme lorsque la séquence x est exécutée. Notez que $t(A, n, x)$ a une valeur bien définie et n'est pas aléatoire. Pour $x \in X$, on dénote par $Pr(x)$ la probabilité que l'exécution x doit effectuée par l'algorithme. Alors l'espérance en temps de l'algorithme sur entrée de taille n s'écrit

$$\mathop{\mathbb{E}}_{X}[t(A,n)] = \sum_{x \in X} Pr(x) \cdot t(A,n,x)$$

L'espérance est donc la somme des temps de chaque exécution $x \in X$, chaque temps étant pondéré par la probabilité que x survienne. Dans la notation $\mathbb{E}_X$, on indique que la variable aléatoire considérée dépend de l'ensemble X. Ce sous-indice est souvent omis dans la littérature et dépend du contexte.

L'espérance est largement utilisée en probabilité et statistiques, et cette définition peut être généralisée à n'importe quelle distribution de probabilités. Une de ses propriétés les plus puissantes est la *linéarité de l'espérance*.

Dit de façon intuitive, cette propriété dit que si on se retrouve à calculer la moyenne d'une somme de variables aléatoires, alors on peut plutôt faire la somme des moyennes des variables. Plus formellement :

Théorème 13. *Soient $r_1, r_2, \ldots, r_n$ des variables aléatoires, possiblement dépendantes. Alors*

$$\mathbb{E}\left[\sum_{i=1}^{n} r_i\right] = \sum_{i=1}^{n} \mathbb{E}[r_i]$$

Exemple : lancer deux dés

Par exemple, prenons un lancer de dé, chaque valeur entre 1 et 6 ayant une probabilité 1/6. Appelons d_1 la valeur du dé. La valeur espérée de d_1 est

$$\begin{aligned}
\mathbb{E}[d_1] &= \frac{1}{6} \cdot 1 + \frac{1}{6} \cdot 2 + \frac{1}{6} \cdot 3 + \frac{1}{6} \cdot 4 + \frac{1}{6} \cdot 5 + \frac{1}{6} \cdot 6 \\
&= \frac{1}{6}(1 + 2 + 3 + 4 + 5 + 6) \\
&= \frac{21}{6} \\
&= 3.5
\end{aligned}$$

Maintenant, on voudrait connaître la valeur espérée de $d_1 + d_2$, i.e. la somme des résultats de deux dés différents d_1 et d_2. Ceci est

$$\mathbb{E}[d_1 + d_2] = \mathbb{E}[d_1] + \mathbb{E}[d_2] = 7$$

et voilà pourquoi le 7 est aussi populaire ! Plus généralement, le résultat de n dés est

$$\mathbb{E}\left[\sum_{i=1}^{n} d_i\right] = \sum_{i=1}^{n} \mathbb{E}[d_i] = \sum_{i=1}^{n} 3.5 = 3.5n$$

Bien sûr, ce principe s'applique aussi si vous avez des variables aléatoires avec des moyennes différentes.

9.2 Algorithmes Monte Carlo et Las Vegas

On distingue deux classes d'algorithmes probabilistes : ceux qui peuvent se tromper, ou pas.

Un algorithme **Las Vegas** est un algorithme probabiliste dont la sortie est toujours correcte pour l'instance donnée. Le temps d'exécution varie selon les choix aléatoires. Tous les algorithmes vus jusqu'à maintenant sont Las Vegas : il donnent toujours la sortie attendue.

Un algorithme **Monte Carlo** est un algorithme probabiliste qui peut produire une sortie inccorrecte avec une certaine probabilité. Par exemple, un algorithme Monte Carlo peut donner une solution optimale 95% du temps, et une solution sous-optimale 5% du temps. Notez qu'un algorithme Monte Carlo peut aussi prendre des temps différents selon ses choix aléatoires.

Dans les deux types d'algorithmes, on exige que l'espérance en temps ne soit pas infinie.

Techniquement, un algorithme Las Vegas est un algorithme Monte Carlo avec probabilité 0 d'erreur. Toutefois, nous utiliserons le terme Monte Carlo pour indiquer implicitement qu'un algorithme peut se tromper.

Voici un exemple algorithme Las Vegas : le *bogosort*, une des pires façons de trier.

fonction *bogoSort(T) //T est de taille n*
  tant que *T n'est pas trié* **faire**
    T = permutation aléatoire des éléments de T, chacune choisie avec probabilité $1/n!$
  fin
  return T

Cet algorithme retourne toujours un tableau T trié, même si ceci peut prendre énormément de temps. C'est donc un algorithme Las Vegas. Il est possible de montrer que son espérance en temps est $O(n \cdot n!)$.

Voici un algorithme (bidon) Monte Carlo.

fonction *plusGrandQue(x, y) //retourne true si* $x > y$
  $r = lancerPiece()$
  si $r = pile$ **alors**
    retourner true
  sinon
    retourner false

Cet algorithme prend toujours le même temps, mais rappelons que ce n,est pas une obligation. Il produit la mauvaise sortie avec probabilité 0.5.

Un autre exemple :

fonction *badSort(T) //*T *est de taille* n
  T = permutation aléatoire des éléments de T, chacune choisie
   avec probabilité $1/n!$
  return T

Cet algorithme retourne une solution incorrecte avec probabilité $1 - \frac{1}{n!}$.

L'analyse de probabilité d'erreur est rarement triviale. Voici un algorithme qui retourne une couverture par sommets d'un graphe (i.e. un ensemble de sommets X tel que chaque arête a au moins un bout dans X). Quelle est la probabilité qu'il retourne un ensemble couvrant de taille minimum ? (réponse : je ne sais pas)

fonction *randomCover(*$G = (V, E)$*)*
  $X = \emptyset$
  tant que $\exists uv \in E$ *tel que* $u \notin X, v \notin X$ **faire**
    $w = u$ avec probabilité 0.5, et $w = v$ avec probabilité 0.5
    Ajotuer w à X
  fin
  return X

Le temps espéré de cet algorithme est d'ailleurs aussi difficile à analyser.

9.3 Quicksort probabiliste

Dans la suite, nous nous atterdons à deux algorithmes probabilistes classiques. Le quicksort, qui est un algorithme Las Vegas, et l'algorithme de Karger pour la coupe minimum, qui est un algorithme Monte Carlo.

Considérons d'abord le *quickSort* probabiliste. On se rappelle que le choix du pivot dans le quickSort influence la complexité de l'algorithme : si le pivot sépare toujours le tableau en deux parties égales, on atteint un temps $O(n \log n)$, mais si un des deux sous tableaux est toujours de taille $n-1$, on atteint un temps $O(n^2)$.

Pour simplifier notre analyse, on suppose que tous les éléments de T sont différents. La version probabiliste choisit le pivot de façon aléatoire.

fonction *quickSortProb(T) //T est de taille n*
  si $n == 1$ **alors**
    return T
  Choisir un entier p entre 1 et n aléatoirement, chacun avec probabilité $1/n$
  $A = [\,]$
  $B = [\,]$
  pour $i = 1..n$, $i \neq p$ **faire**
    si $T[i] \leq T[p]$ **alors**
      $A.append(T[i])$
    sinon
      $B.append(T[i])$
  fin
  $A' = quickSort(A)$
  $B' = quickSort(B)$
  return $A' + [T[p]] + B'$ //où "+" dénote la concaténation

Il est facile de voir que cet algorithme retourne toujours un tableau trié.

Le pire cas reste $O(n^2)$, mais on peut maintenant se demander quel est le temps espéré parmi toutes les exécutions possibles sur une même entrée.

On peut se convaincre rapidement que le nombre de comparaisons effectuées au total donne une borne O sur le temps requis par l'algorithme. On peut donc se contenter de compter combien de fois l'instruction $T[i] \leq T[p]$ est exécutée, en moyenne. Ceci demande de calculer l'espérance de la somme du nombre de comparaisons faites pour chaque paire d'éléments $T[i], T[j]$. On note que si $T[i]$ et $T[j]$ sont comparés à un certain moment, c'est que l'un de $T[i]$ et $T[j]$ est devenu un pivot. À ce moment, un des éléments sera envoyé dans un appel récursif et le pivot ne sera pas envoyé, ce qui veut dire que $T[i]$ et $T[j]$ peuvent être comparés au maximum une fois.

À l'appel initial, on choisit le pivot et on le compare toujours avec les $n-1$ valeurs. S'il y a k éléments plus petits que le pivot, alors le sous-tableau A aura k éléments et le sous-tableau B en aura $n-k-1$. On note

que $k \in \{0, 1, \ldots, n-1\}$.

Pour un k donné, quelle est la probabilité qu'on choisisse un pivot ayant k éléments plus petits ? Il y a n éléments dans le tableau, et seul le k-ième plus grand élément a k éléments plus petits. Il n'y a donc qu'une seule façon d'avoir un tel pivot. Puisque le choix parmi les n pivots possibles se fait de façon uniforme (i.e. équiprobable), pour chaque k, on a une probabilité $\frac{1}{n}$ de choisir un pivot qui a k éléments plus petits.

Avec ces informations, on peut exprimer le nombre de comparaisons espéré de l'algorithme par une récurrence. Soit E_n le nombre de comparaisons espéré pour trier un tableau de taille n avec l'algorithme QuickSort probabiliste. On a

$$\begin{aligned} E_n &= n - 1 + \frac{1}{n}(E_0 + E_{n-1}) + \frac{1}{n}(E_1 + E_{n-2}) + \ldots + \frac{1}{n}(E_{n-1} + E_1) \\ &= n - 1 + \frac{1}{n}\sum_{k=0}^{n-1}(E_k + E_{n-k-1}) \end{aligned}$$

où le terme $n-1$ vient des $n-1$ comparaisons avec le pivot à l'appel initial. Observez que $E_0 = E_1 = 0$, car il n'y a rien à faire pour trier 0 ou 1 élément. Ce sont nos cas terminaux. Ce sont nos cas de base — nous allons tout de même conserver le terme E_1 dans la sommation. Aussi, chaque terme E_k apparaît deux fois dans la somme ci-haut. On peut donc écrire

$$E_n = n - 1 + \frac{2}{n}\sum_{k=1}^{n-1} E_k$$

On voudrait bien montrer que ceci est $O(n \log n)$. À partir d'ici, nous n'avons plus besoin de notion de probabilité - nous allons appliquer le même genre d'analyse que précédemment.

On va démontrer par induction que $E_n \leq 2n \log n$. Avant d'entreprendre cette preuve, on peut déjà voir que l'induction va nous demander de faire la somme $\sum_{k=2}^{n-1} 2k \log k$. On va d'abord obtenir une borne sur cette expression.

Lemme 6. *Soit $f(0) = f(1) = 0$ et $f(n) = \sum_{k=1}^{n-1} k \log k$ pour $n \geq 2$. Alors pour tout entier $n \geq 1$, $f(n) \leq \frac{n^2}{2} \log n - \frac{n^2}{8}$.*

Démonstration. On procède par induction sur n. Comme cas de base, on a besoin de considérer $n = 1, 2, 3$, la raison étant que dans l'induction, on aura besoin de supposer que $n \geq 4$. Pour $n = 1$, $f(1) = 0$ et c'est trivial.

Pour $n = 2$, $f(2) = 1 \log 1 = 0$ l'énoncé est aussi correct. Pour $n = 3$, $f(3) = 2 \log 2 = 2$ et $\frac{n^2}{2} \log n - \frac{n^2}{8} = \frac{9}{2} \log 3 - \frac{9}{8} \simeq 3.8$. L'énoncé est correct car $2 \leq 3.8$.

Pour l'induction, on suppose que $f(m) \leq \frac{m^2}{2} \log m - \frac{m^2}{8}$ pour tout $m < n$. En utilisant cette hypothèse, on a

$$
\begin{aligned}
f(n) &= \sum_{k=0}^{n-1} k \log k \\
&= (n-1) \log(n-1) + \sum_{k=1}^{n-2} k \log k \\
&= (n-1) \log(n-1) + f(n-1) \\
&\leq (n-1) \log(n-1) + \frac{(n-1)^2}{2} \log(n-1) - \frac{(n-1)^2}{8} \quad \text{par hypothèse} \\
&= (n-1) \log(n-1) \left[1 + \frac{n-1}{2} \right] - \frac{n^2}{8} - \frac{2n}{8} + 1 \\
&= (n-1) \log(n-1) \left[1 + \frac{n-1}{2} \right] - \frac{n^2}{8} \qquad \text{pour } n \geq 4 \\
&= \frac{1}{2}(n-1)(n+1) \log(n-1) - \frac{n^2}{8} \\
&= \frac{1}{2}(n^2-1) \log(n-1) - \frac{n^2}{8} \\
&\leq \frac{n^2}{2} \log(n-1) - \frac{n^2}{8}
\end{aligned}
$$

ce qu'il fallait démontrer. □

On peut maintenant montrer que l'espérance en temps de QuickSort est bornée par $2n \log n$.

Théorème 14. *Le nombre de comparaisons espéré E_n de QuickSort est au plus $4n \log n$.*

Démonstration. On démontre l'énoncé par induction sur n. On prend $n = 1$ comme cas de base, et puisque $E_1 = 0$ ce cas est trivial.

Pour l'induction, on suppose que $E_m \leq 4m \log m$ pour tout $m < n$. On a déjà argumenté que $E_n = n - 1 + \frac{2}{n} \sum_{k=1}^{n-1} E_k$. En utilisant l'induction, puis le lemme 6, on obtient

$$
\begin{aligned}
E_n &= n - 1 + \frac{2}{n}\sum_{k=1}^{n-1} E_k \\
&\leq n - 1 + \frac{2}{n}\sum_{k=1}^{n-1} 4k\log k && \text{par hypothèse d'induction} \\
&= n - 1 + \frac{8}{n}\sum_{k=1}^{n-1} k\log k \\
&\leq n - 1 + \frac{8}{n}\left[\frac{n^2}{2}\log n - \frac{n^2}{8}\right] && \text{Lemme 6} \\
&= n - 1 + 4n\log n - n \\
&\leq 4n\log n
\end{aligned}
$$

□

9.4 Algorithme de Karger pour la coupe minimum

Nous allons maintenant voir le désormais célèbre algorithme de Karger pour le problème de la *coupe minimum*.

Soit $G = (V, E)$ un graphe. Une *bipartition* de G est une paire d'ensembles (X, Y) telle que $X \cup Y = V, X \cap Y = \emptyset$ et $X \neq \emptyset, Y \neq \emptyset$. Donc (X, Y) sépare les sommets en deux parties non-vides.

Pour une bipartition (X, Y) de G, on dénote $E(X, Y) = \{xy : x \in X, y \in Y\}$, i.e. les arêtes qui traversent x et y. On appelle aussi parfois (X, Y) une *coupe*, car elle coupe les arêtes qui traversent d'un côté et de l'autre. Dans le problème de la coupe minimum, on cherche la bipartition qui minimise le nombre d'arêtes traversantes.

Problème de la coupe minimum
Entrée : un graphe $G = (V, E)$.
Sortie : une bipartition (X, Y) qui minimise $|E(X, Y)|$ parmi tous les choix possibles.

La recherche d'une coupe minimum peut servir à détecter à quel endroit un réseau internet peut être coupé pour perdre la connexion, ou encore pour séparer un réseau social en deux communautés qui partagent peu de liens. On peut résoudre le problème de la coupe minimum en temps polynomial à l'aide de formulations de *flots* dans des graphes (sujet que nous ne couvrons pas

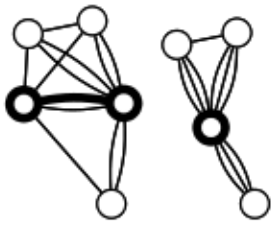


FIGURE 9.1: Un exemple de contraction d'arête sur un multi-graphe. Source : `https://en.wikipedia.org/wiki/Karger's_algorithm`

ici). Les implémentations typiques déterministes prennent un temps $O(n^5)$. Nous présentons un algorithme Monte Carlo pour le problème (donc, qui peut retourner une mauvaise solution).

L'algorithme travaille sur des *multi-graphes*. Dans un multigraphe $G = (V, E)$, V est toujours l'ensemble des sommets mais E est un multiensemble : il peut contenir plusieurs fois une même arête. Par exemple, on permet $V = \{u, v\}$ et $E = \{uv, uv, uv, uv\}$, et il y a quatre arêtes entre u et v.

L'idée est de contracter des sommets au hasard jusqu'à ce qu'il n'en reste qu'un. La contraction se définit comme suit. Soit $G = (V, E)$ un multigraphe et $uv \in E$. La *contraction* de uv consiste à remplacer u et$v$ par un seul zommet z, et donner à z tous les voisins que u et v avaient. Plus formellement, pour contracter uv, on doit

1. Ajouter un nouveau sommet z à G
2. Pour chaque arête $uw \in E$, ajouter une arête wz à G. Si l'arête uw est répétée k fois, alors on ajoute k arêtes uz.
3. Pour chaque arête $vw \in E$, ajouter une arête wz à G. Si l'arête vw est répétée k fois, alors on ajoute k arêtes wz.
4. Retirer u et v de G ainsi que toutes leurs arêtes incidentes.

Pour une arête uv de G, on écrit G/uv le graphe obtenu en contractant l'arête uv. Si on stocke le nombre d'occurrences de chaque arête au lieu de stocker chaque répétition d'une arête, on peut calculer la contraction G/uv en temps $O(n)$. Il suffit d'ajouter le sommet z et, pour chaque sommet w de $V \setminus \{u, v\}$, d'ajouter l'arête zw le nombre de fois qu'on a les arêtes uw et vw.

Notez aussi que lorsqu'on contracte uv, on peut mémoriser le fait que z représente $\{u, v\}$. Plus généralement, si on contracte u et v tels que u et v représentent des ensembles de sommets originaux X_u et X_v, alors on peut

se rappeler que z représente $X_u \cup X_v$. Cette union peut se faire en temps $O(1)$ si X_u et X_v sont des listes chaînées, par exemple.

fonction *kargerMinCut*($G = (V, E)$) *//V est de taille n*
  tant que $|V| > 2$ **faire**
    Soit $uv \in E$, chaque arête choisie avec probabilité $\frac{1}{|E|}$
    $G = G/uv$
  fin
  Soient u et v les deux sommets restants de G
  Soit X_u l'ensemble de sommets représentés par u
  Soit X_v l'ensemble de sommets représentés par v
  return (X_u, X_v)

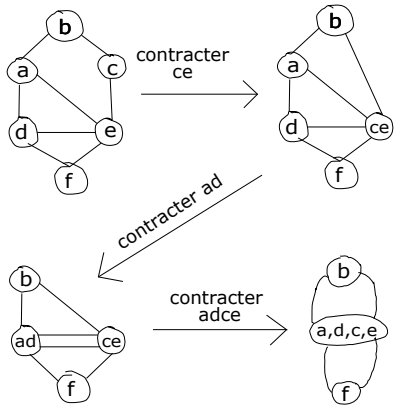


FIGURE 9.2: Un exemple des trois premières étapes de l'execution de Karger.

Il est évident que cet algorithme peut se tromper. C'est donc un algorithme Monte Carlo, mais pas Las Vegas. Puisqu'on fait $n-2$ contractions et que chacune prend un temps $O(n)$, le temps d'exécution est toujours $O(n^2)$.

Si m n'est pas trop grand, il existe une version $O((n+m)\log n)$. L'idée est d'attribuer un poids différent à chaque arête de façon aléatoire, de reconstruire un arbre couvrant minimum avec l'algorithme de Kruskal en temps $O((n+m)\log n)$ (voir série d'exercices sur les algorithmes gloutons), puis de retourner les sommets des deux composantes connexes obtenus en retirant l'arête de poids maximum dans cet arbre. Il est possible de prouver que cet algorithme est équivalent à celui de Karger.

Il existe aussi une version optimisée qui nécessite un temps $O(m)$ (en

supposant $m > n$). Nous ne discuterons pas de cette optimisation ici.

Probabilité de succès

Intéressons-nous maintenant à la probabilité que cet algorithme retourne effectivement une coupe minimum. L'intuition est que la probabilité de contracter une arête de la coupe minimum ne sera pas "trop" élevée parce qu'à chaque contraction, il y a beaucoup d'arêtes à choisir qui ne sont pas dans cette coupe (parce que justement, elle est minimum).

Soit (X, Y) une coupe minimum, et soit $k = |E(X, Y)|$. On observe que $\min_{v \in V} deg(v) \geq k$, parce que s'il y avait un sommet v avec $deg(v) < k$, alors $(\{v\}, V \setminus \{v\})$ serait une meilleure coupe. Puisque le degré minimum est au moins k, on doit avoir $|E| \geq \frac{kn}{2}$ (si vous ne voyez pas pourquoi, arrêtez-vous).

Quelle est la probabilité qu'à la première contraction, on contracte une arête de $E(X, Y)$? On a k chances sur $|E|$ de tomber sur une telle arête, et la probabilité d'échec à la première itération est donc $\frac{k}{|E|}$. On se rappelle que $|E| \geq \frac{kn}{2}$. Donc la probabilité de contracter une arête de $E(X, Y)$ est

$$\frac{k}{|E|} \leq \frac{k}{kn/2} = \frac{2}{n}$$

Donc la première contraction sera "souvent" correcte.

La probabilité de ne *pas* contracter d'arête de $E(X, Y)$ est donc $1 - \frac{2}{n}$. On peut ensuite observer que si on contracte $uv \notin E(X, Y)$, alors en choisissant une arête au hasard dans G/uv, la probabilité de contracter une arête de $E(X, Y)$ est $\frac{k}{|E(G/uv)|} \leq \frac{2}{n-1}$ (encore là, il faut s'arrêter et se convaincre que c'est vrai : ce n'est pas direct car ici G/uv est un multigraphe). La probabilité d'éviter de contracter une arête de $E(X, Y)$ dans la première et la deuxième itération est $(1 - \frac{2}{n})(1 - \frac{2}{n-1})$.

Plus généralement, si p_n est la probabilité d'éviter la contraction d'une arête de $E(X, Y)$ lorsque G a une taille de n, alors on peut écrire

$$p_n = \left(1 - \frac{2}{n}\right) p_{n-1}$$

avec $p_2 = 1$. Si on fait l'expansion de p_n, on obtient

$$\begin{aligned} p_n &= \left(1-\frac{2}{n}\right)\left(1-\frac{2}{n-1}\right)\left(1-\frac{2}{n-2}\right)\cdots\left(1-\frac{2}{4}\right)\left(1-\frac{2}{3}\right) \\ &= \left(\frac{n-2}{n}\right)\left(\frac{n-3}{n-1}\right)\left(\frac{n-4}{n-2}\right)\cdots\left(\frac{2}{4}\right)\left(\frac{1}{3}\right) \\ &= \frac{2}{n(n-1)} \\ &= \frac{1}{\binom{n}{2}} \end{aligned}$$

l'avant-dernière égalité étant obtenue en éliminant les facteurs présents au numérateur et dénominateur.

On vient donc d'apprendre que notre probabilité de succès est au moins $1/\binom{n}{2}$ (et donc une probabilité d'échec de $1 - 1/\binom{n}{2}$). Ce n'est pas remarquable. Si $n = 100$, on a environ 49,999 chances sur 50,000 que l'algorithme échoue. On peut remédier à ce problème en réexécutant l'algorithme plusieurs fois.

Si on exécute l'algorithme T fois, la probabilité que toutes les exécutions échouent est de

$$\left(1-\frac{1}{\binom{n}{2}}\right)^T$$

Voici un fait souvent utile en probabilité. Soit x un réel tel que $0 < x < 1$. Alors $1 - x \leq e^{-x}$ (où e est la constante de Euler (du moins l'une d'entre elles)). Ceci implique que $(1-x)^T \leq e^{-x \cdot T}$. Dans notre cas, $x = 1/\binom{n}{2}$. On obtient

$$\left(1-\frac{1}{\binom{n}{2}}\right)^T \leq e^{-\frac{1}{\binom{n}{2}} \cdot T}$$

Si on choisit $T = \binom{n}{2} \ln n$, ceci devient au plus $1/e^{\ln n} = 1/n$.

Donc en répétant l'algorithme $\binom{n}{2} \ln n$ fois, on a une chance sur n que l'on n'ait jamais obtenu de coupe minimum. Lorsque n est grand, cette probabilité d'échec est négligeable.

Avec ce nombre de répétitions, la complexité de l'algorithme devient $O(n^4 \log n)$. En utilisant la version optimisée $O(m)$ de l'algorithme, ceci devient $O(n^2 m \log n)$.

Pour finir, cet algorithme a été étendu et a donné lie à ce qu'on appelle l'algorithme Karger-Stein, qui atteint une complexité $O(n^2 \log^3 n)$.

9.5 Résumé

Ce que vous devez savoir :
— Sachez appliquer l'algorithme de Karger pour la coupe minimum
— Sachez trouver le temps espéré d'un algorithme probabiliste.
— Comprenez les algorithmes de type Las Vegas, Monte Carlo et leurs différences.
— Faites les exercices.