Notes de cours INFO607, L3 Informatique
Algorithmique II

Jacques-Olivier Lachaud
LAMA, Université de Savoie
http://www.lama.univ-savoie.fr/wiki
(suivre INFO607)

9 avril 2024

Introduction

Ce document propose quelques éléments nouveaux d'algorithmique, qui font suite à l'étude des structures de données linéaires et arborescentes, tout en restant assez indépendant des algorithmes sur les graphes. Il donne des techniques pour déterminer les temps d'exécution des algorithmes dans les cas où l'anlyse en pire cas échoue. Pour les travaux pratiques, il présuppose une certaine connaissance du langage C (vous pouvez vous référer aux notes de cours de INFO504 - Programmation C, sur le même site), notamment sa syntaxe, ses mécanismes d'allocation mémoire (pile, tas), son organisation (fichiers sources, fichiers en-tête), son cycle de développement.

Les livres sur l'algorithmique et les structures de données sont pléthores. Un livre très complet sur l'algorithmique est *Introduction à l'algorithmique, Cormen/Leiserson/Rivest/Stein, Dunod, 2004*. Pour le langage C, *The C programming Langage, Kernighan/Ritchie* est très bon, mais n'intègre pas les modifications de la dernière version du C. On peut conseiller aussi le *Méthodologie de la programmation en C, Braquelaire, Ed. Dunod, 2005*, qui intègre la norme C99.

Plan

0. Complexité des algorithmes (Rappels)
1. Analyse des fonctions récursives
2. Analyse amortie
3. Structures de données pour ensembles disjoints
4. Géométrie algorithmique

0. Complexité des algorithmes (Rappels)

Il y a souvent deux buts contradictoires lorsque l'on cherche à mettre au point un algorithme pour résoudre un problème donné :

1. L'algorithme doit être facile à comprendre, coder, maintenir, mais aussi facile à vérifier.
2. L'algorithme doit utiliser efficacement les ressources de l'ordinateur, c'est-à-dire s'exécuter rapidement mais aussi prendre une place raisonnable en mémoire.

Si un algorithme doit être utilisé très souvent, il est alors intéressant de mettre en œuvre une solution complexe mais efficace en temps et/ou en espace mémoire. Il est alors utile de pouvoir comparer objectivement les complexités relatives.

0.1 Mesure du temps d'exécution d'un programme

Le temps d'exécution d'un programme dépend :

1. des données en entrée,
2. de la qualité du code généré par le compilateur,
3. de la nature et de la rapidité des instructions de la machine d'exécution du programme,
4. de l'algorithme utilisé pour résoudre le problème.

D'après le premier point, il est clair que le temps d'exécution n'est pas juste une valeur, mais une fonction des données. Très souvent, la valeur des données n'est pas significative, mais seul compte le nombre de données, mettons n. Le temps d'exécution d'un programme sera donc une fonction $T(n)$, qui est le temps d'exécution de ce programme pour n données en entrée. Par exemple, il est clair qu'un programme de tri sera de plus en plus lent si on augmente le nombre de données à trier.

Maintenant l'unité de temps de $T(n)$ ne peut être précisée du fait des points (2) et (3). L'unité ne sera donc que relative. Un même programme P aura peut-être un temps d'exécution $T_1(n) = c_1 n^2$ sur une machine $M1$ et temps d'exécution $T_2(n) = c_2 n^2$ sur une machine $M2$. Si les constantes c_1 et c_2 peuvent être distinctes (et très variables), il est en revanche peu probable que la partie n^2 du temps d'exécution se transforme d'une machine à une autre. En effet, un processeur peut être cadencé

plus rapidement qu'un autre, mais globalement, s'il doit faire K opérations, cela lui prendra un temps proportionnel à K.

On dira donc souvent que le programme P s'exécute en un temps proportionnel à n^2, et non s'exécute en $c_1 n^2$ sur la machine $M1$, car cela ne présente pas toujours un intérêt majeur.

Parfois, le temps d'exécution d'un programme peut être rapide sur n données mais lent sur n autres données. Un exemple typique est le tri insertion avec des données déjà triées, qui est rapide, mais qui est lent sur la plupart des autres données. Dans ces cas-là, $T(n)$ désignera le temps d'exécution dans le *pire cas*, car c'est celui qui est problématique.

Une autre façon est de définir le temps d'exécution *moyen* $\hat{T}(n)$, qui est la moyenne des temps d'exécution de toutes les données de taille n. Si cette mesure peut paraître plus utile ou objective, il faut néanmoins garder à l'esprit que les ensemble de n données sont rarement équiprobables dans les applications réelles. Dans le cas du tri, on a souvent des données quasi-triées en entrée, du fait des processus de saisie ou d'acquisition. Néanmoins, on montrera dans certains cas comment calculer $\hat{T}(n)$, et sous quelles hypothèses ce temps est valide.

Exemples :

1. L'algorithme de calcul du plus grand élément d'un tableau à n éléments nécessite de regarder toutes les cases du tableau une fois exactement. Le temps d'exécution dans le pire cas est donc proportionnel à n. Comme dans le meilleur cas il est aussi proportionnel à n, il est clair que le temps d'exécution moyen est proportionnel à n lui-aussi.
2. Un algorithme de recherche dichotomique dans un tableau trié est beaucoup plus rapide. On montre que son temps d'exécution est proportionnel à $\log_2 n$, dans le pire cas et dans le cas moyen aussi.
3. L'algorithme de tri insertion a un temps d'exécution dans le pire cas proportionnel à n^2, mais son temps d'exécution moyen est moins clair. Si on suppose que tous les ordres sont équiprobables, on peut montrer que le temps d'exécution moyen est aussi proportionnel à n^2 (avec une constante inférieure).

0.2 Notations O, Θ, Ω

Lorsque l'on veut comparer les vitesses d'accroissement de fonction sans se préoccuper des constantes mises en jeu, il est pratique d'utiliser une notation concise pour exprimer la notion de proportionnalité, où le fait qu'une fonction grandit plus vite ou moins vite qu'une autre "à l'infini". On dispose pour cela de trois notations classiques : O = "grand O", Θ = "Téta", Ω = "grand Oméga".

Dans la suite T et f sont deux fonctions de n.

— $T(n) = O(f(n))$ ssi il existe deux constantes c et n_0 telles que $\forall n \geq n_0, T(n) \leq cf(n)$. Cette notation indique que T croît moins vite que f.

— $T(n) = \Omega(f(n))$ ssi il existe deux constantes c et n_0 telles que $\forall n \geq n_0, T(n) \geq cf(n)$.[1] Cette notation indique que T croît plus vite que f.

— $T(n) = \Theta(f(n))$ ssi il existe trois constantes c_1, c_2 et n_0 telles que $\forall n \geq n_0, c_1 f(n) \leq T(n) \leq c_2 f(n)$. Cette notation indique que T et f croissent aussi vite.

Une notation importante est $O(1)$ qui exprime la croissance de toute fonction constante. Ainsi, on dira qu'un ensemble d'instructions dont le temps d'exécution ne dépend pas de la taille des données en entrée et est borné par une constante a une complexité $O(1)$.

Exemples :

— Il est clair que $n = O(n)$, $n = \Omega(n)$, et $n = \Theta(n)$.
— Plus généralement, $f(n) = O(\alpha f(n))$, $f(n) = \Omega(\alpha f(n))$ et $f(n) = \Theta(\alpha f(n))$.
— On a aussi que $n = O(n^2)$, $n^2 = O(n^3)$ et plus généralement $n^a = O(n^b)$ ssi $0 \leq a \leq b$.
— On a bien sûr $n = O(n \log n)$ et $n \log n = O(n^2)$

Exercice : (Notations O, Θ, Ω)

1. Montrez que $T(n) = \Theta(f(n))$ ssi $T(n) = O(f(n))$ et $T(n) = \Omega(f(n))$.
2. Montrez que $T(n) = O(f(n))$ ssi $f(n) = \Omega(T(n))$.
3. Montrez que si $T(n) = O(f(n))$ et $f(n) = O(g(n))$ alors $T(n) = O(g(n))$.
4. Montrez que si $T(n) = \Theta(f(n))$ et $f(n) = \Theta(g(n))$ alors $T(n) = \Theta(g(n))$.

1. Une définition non symétrique parfois utilisée est de dire qu'il existe une infinité de $n \geq n_0$ pour lesquels $T(n) \geq cf(n)$, mais pas forcément tous.

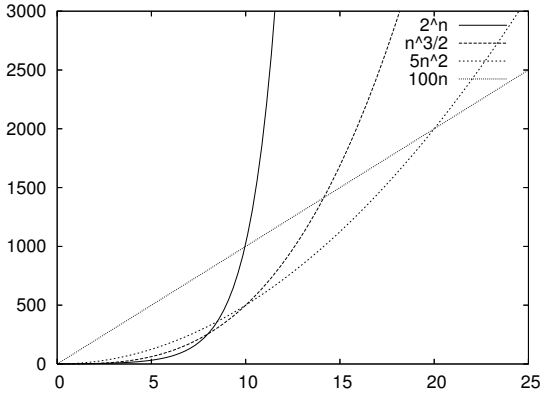


FIGURE 1 – Temps d'exécution de quatre programmes différents, de temps d'exécution respectifs 2^n, $n^3/2$, $5n^2$, $100n$. L'unité de temps est sans importance, mettons des secondes.

On dispose de règles d'addition et de multiplication de ces notations, notamment :

Addition de O. Si $T_1(n) = O(f(n))$ et $T_2(n) = O(g(n))$, alors $T_1(n)+T_2(n) = O(\max(f(n), g(n)))$.

C'est notamment utile lorsque vous avez mesuré la complexité de deux parties successives de votre programme et que vous cherchez à déterminer la complexité du programme tout entier. Il s'agit bien de l'addition de deux temps.

Produit de O. Si $T_1(n) = O(f(n))$ et $T_2(n) = O(g(n))$, alors $T_1(n)T_2(n) = O(f(n)g(n))$.

Cela montre par exemple que $O(n^2/2) = O(n^2)$. La règle des produits est utilisée pour mesurer le temps d'exécution de programme contenant des boucles ou des appels répétitifs à un même sous-programme de complexité connue.

Quelques questions :

1. Comment montrer que $\log n = O(n)$?
2. Montrez que si $f(n) = O(g(n))$ alors $h(n)f(n) = O(h(n)g(n))$.
3. En déduire que $n \log n = O(n^2)$.

0.3 Complexité et temps d'exécution asymptotique

Il n'est donc pas facile de comparer les efficacités respectives d'algorithmes, sachant que leur vitesse d'exécution dépend de beaucoup de paramètres, dont la machine. On va voir néanmoins que l'on dispose d'un moyen pour le faire qui est assez objectif.

Supposons par exemple que l'on dispose de quatre programmes $(P_i)_{i=1..4}$ qui résolvent le même problème. Chaque programme P_i s'exécute sur une machine M_i. On note $T_i(n)$ leurs temps d'exécution respectifs, que l'on peut observer sur la Figure 1.

Lequel est le meilleur ? Cela dépend de la taille des données à traiter et du temps que l'on peut y consacrer. Si on suppose que l'on ne dispose que de 10^3 secondes, ces quatre programmes/machines sont quasiment aussi efficaces les uns que les autres. Si maintenant on dispose de 10^4 secondes, on s'aperçoit que c'est le programme/machine avec le taux d'accroissement le plus faible qui devient vite le plus efficace. Ainsi, pour un algorithme en $O(n)$, le gain réalisé est identique au temps rajouté, ce qui n'est pas le cas pour les autres.

$T(n)$	Taille max pour 10^3 s	Taille max pour 10^4 s	Gain
$100n$	10	100	10
$5n^2$	14	45	3,2
$n^3/2$	12	27	2,3
2^n	10	13	1,3

Un façon complètement symétrique de voir les choses est de supposer que l'on garde les mêmes programmes compilés de la même façon, mais qu'on puisse cadencer les processeurs dix fois plus vite.

Le gain observé pour le même temps sera alors complètement similaire au fait de se donner dix fois plus de temps.

On en conclut que lorsqu'on veut traiter des données de plus en plus grandes, il est intéressant de comparer les temps d'exécution en terme d'accroissement O, c'est-à-dire de manière *asymptotique*, en négligeant les constantes qui ne sont pertinentes que pour des petites données. La *complexité en temps* d'un programme est donc son temps d'exécution mesuré en terme d'accroissement de la taille des données en entrée.

Exemples :

1. Sur l'exemple précédent, la meilleure complexité est celle du programme de temps $100n$, même si ce n'est pas le programme le plus efficace pour de petites valeurs de n.
2. Dans certains cas, la constante est importante. Il existe un problème d'optimisation classique (programmation linéaire) dont l'algorithme classique dit du simplexe est efficace en pratique, mais peut avoir une complexité exponentielle dans certains cas. Il existe un algorithme de complexité polynomiale qui résoud le même problème, mais la constante est très importante et sur toutes les données que l'on peut traiter le rend inutilisable.

0.4 Calcul de la complexité d'un algorithme

On peut maintenant déterminer (à des constantes près) la complexité d'un algorithme donné. Attention, on est souvent obligé de donner une complexité dans le pire cas, notamment lorsque le programme a des morceaux d'instructions qui sont conditionnés.

Les règles sont les suivantes :

— Le temps d'exécution de chaque affectation, lecture, écriture en mémoire est supposé être constant ou en $O(1)$.

— De même, on suppose souvent (mais pas toujours) que le temps d'exécution de l'addition, soustraction, multiplication, division est constant. Cela est faux en général, mais assez vrai lorsqu'on limite la taille des données à des valeurs codées sur moins de 32 ou 64 bits.

— Si $T_1(n)$ et $T_2(n)$ sont les temps d'exécution de deux fragments de programme, le temps d'exécution de leur succession est $T_1(n) + T_2(n)$. Si $T_1(n) = O(f(n))$ et $T_2(n) = O(g(n))$ alors la règle des sommes donne $T(n) = O(\max(f(n), g(n)))$.

En particulier, une succession d'instructions élémentaires prend $O(\max(1, 1, \ldots, 1))$, soit $O(1)$.

— Le temps d'exécution d'un "Si" est le temps d'exécution de la condition (souvent $O(1)$) plus le temps d'exécution le plus large entre la partie "alors" et la partie "sinon". On note que le temps d'exécution devient un temps dans le pire cas.

On peut utiliser la notation Ω pour le meilleur cas.

— Le temps d'exécution d'une boucle est la somme de tous les temps d'exécution du bloc interne plus les temps d'exécution de la condition de terminaison. Si le nombre d'itération maximal $O(f(n))$ est connu et que le temps d'exécution du bloc interne est borné par $O(g(n))$, alors le temps d'exécution de la boucle est $O(f(n)g(n))$ (règle des produits).

— Pour les appels de fonction/procédure, il faut bien sûr sommer leur temps d'exécution. Si l'appel est récursif, il est en général sur une partie plus petite des données. On obtient donc une relation de récurrence sur les temps d'exécution, et il existe des techniques classiques pour trouver la forme close qui correspond à la récurrence.

Nb : exemple de calcul de la factorielle : $T(n) = c+T(n-1)$. On en déduit $T(n) = cn = O(n)$.

0.5 Complexité de quelques algorithmes classiques

Quelques questions :

- Montrez que la complexité d'un algorithme de sommation conditionnelle est $O(n)$. Exemple la moyenne des notes différentes de 0.
- Montrez que le tri à bulle est un $O(n^2)$.
- Montrez que le pire cas de quicksort est un $O(n^2)$.
- Quelle est la complexité de la recherche dichotomique ?
- Quelle est la complexité des calculs récursif/itératif de la factorielle ?
- Quelle est la complexité de l'algorithme du sac-à-dos ?
- Quelle est la complexité de calcul de la version récursive du binomial C_n^p ? (Remarquez que la somme des binomiaux fait 2^n).
- Quelle est la complexité du calcul de l'enveloppe convexe par l'algorithme de Graham ? Par Melkman ?
- Quelle est la complexité des algorithmes Insérer et SupprimerMin des tas ? En déduire la complexité du tri par tas ?

On note que l'on a bien montré la complexité en pire cas d'un algorithme en $O(f(n))$ lorsqu'on peut exhiber un exemple d'exécution où le temps d'exécution atteint bien asymptotiquement ce $f(n)$. C'est la même chose pour le meilleur cas. Ainsi $O(n^3)$ est une borne supérieure de la complexité dans le pire cas du tri à bulle, mais n'est jamais atteinte. De même $\Omega(n)$ est une borne inférieure de la complexité dans le meilleur cas ce même algorithme, mais n'est jamais atteinte non plus.

0.6 Complexité moyenne en temps

Il est souvent plus difficile de calculer la complexité moyenne d'un algorithme. Il faut en effet calculer le temps d'exécution de l'algorithme considéré sur toutes les données possibles, en normalisant la probabilité d'apparition de chaque ensemble de données selon l'application visée. Très souvent, pour des raisons de simplicité, on supposera que toutes les données ont des probabilités identiques d'apparition. Evidemment, lorsque les temps d'exécution en pire cas et meilleur cas coïncident en ordre de grandeur, le temps moyen est du même ordre. Ce n'est que lorsqu'ils diffèrent qu'une analyse en moyenne devient nécessaire.

L'analyse en moyenne peut être très délicate dans certains cas, et nécessiter une connaissance poussée d'outils probabilistes (voir par exemple le calcul de la complexité moyenne des Bogosort et Bozosort, cf Wikipédia). Sur certains algorithmes, elle est plus facile moyennant des connaissances sur les séries.

0.6.1 Complexité moyenne d'une recherche dans un tableau

Il faut distinguer deux cas, selon que la donnée recherchée est dans le tableau ou non.

Si oui, on suppose qu'elle peut être dans n'importe quelle case de manière équiprobable. Dans ce cas, si elle est dans la case d'indice i, le temps de recherche de l'élément est proportionnel à i. On a donc

$$\begin{aligned} \hat{T}(n) &= \frac{1}{n}\sum_{i=0}^{n-1} i+1 \\ &= \frac{n+1}{2} \end{aligned}$$

Si l'élement n'est pas dans le tableau, le temps de recherche est invariablement n, le temps moyen dans ce cas est donc n. Si on se donne maintenant p comme étant la probabilité *a priori* que l'élément appartienne au tableau, le temps moyen d'exécution est donc proportionnel à

$$\hat{T}(n) = p\tfrac{n+1}{2} + (1-p)n = \tfrac{(2-p)n+p}{2} = (1-p/2)O(n)$$

Quelques questions :

1. est-il légitime d'ignorer la constante de proportionnalité devant le temps de recherche ? Mettre à jour si nécessaire ce calcul. Comment calculer de manière effective cette/ces constante(s) pour un exécutable donné ?
2. Qu'en est-il de la recherche dans une liste, triée ou non ?

0.6.2 Complexité moyenne d'une recherche dans un ABR

Nous avons montré dans un autre cours que la profondeur moyenne d'un ABR était inférieure à $1 + 2\log n$, où log désigne le logarithme naturel, et en faisant certaines hypothèses sur la construction de l'ABR et sur les données insérées. Au vu des algorithmes d'insertion, de recherche et de suppression, leur complexité moyenne dépend de cette profondeur moyenne et on en déduit qu'ils sont en $O(\log n)$.

En fait, il est clair qu'une recherche d'un élément existant est en $O(\log n)$. Pour un élément non existant, il faudrait plutôt calculer la longueur moyenne d'un chemin de la racine à une feuille ou à un nœud qui n'a qu'un descendant. Pour l'insertion, c'est plutôt aussi cette quantité qu'il faut examiner.

0.6.3 Complexité moyenne du quicksort

On peut procéder d'une manière similaire au calcul de la profondeur moyenne d'un arbre binaire pour déterminer la complexité moyenne du quicksort. Il faut faire l'hypothèse que tous les ordres sont équiprobables et qu'à chaque étape de partitionnement la position du pivot peut être n'importe laquelle des cases étudiées avec même probabilité.

Le temps d'exécution $T(n)$ d'une étape de quicksort est donc de la forme :

$$\begin{aligned} T(n) = \quad & n \quad \text{(RecherchePivot)} \\ + \quad & T(i) \quad \text{(Quicksort sur les i premiers éléments)} \\ + \quad & T(n-1-i) \quad \text{(Quicksort sur les n-1-i derniers éléments)} \end{aligned}$$

Le temps moyen $\hat{T}(n)$ d'une étape est donc la moyenne des temps possibles d'exécution. Or le pivot peut se retrouver à une position i quelconque de façon équiprobable. Ceci induit, pour $n \geq 2$:

$$\hat{T}(n) = \frac{1}{n}\sum_{i=0}^{n-1} n + \hat{T}(i) + \hat{T}(n-1-i)$$

avec les temps moyens $\hat{T}(1) = 1$ et $\hat{T}(0) = 0$. Le premier terme sort de la somme. Les deux autres termes sont symétriques. Cela donne

$$\hat{T}(n) = n + \frac{2}{n}\sum_{i=0}^{n-1} \hat{T}(i)$$

On calcule maintenant la quantité suivante :

$$\begin{aligned} n\hat{T}(n) - (n-1)\hat{T}(n-1) &= n^2 - (n-1)^2 + 2\hat{T}(n-1) \\ \Leftrightarrow n\hat{T}(n) &= 2n - 1 + (n+1)\hat{T}(n-1) \\ \Leftrightarrow \hat{T}(n) &= 2 - \frac{1}{n} + \frac{(n+1)}{n}\hat{T}(n-1) \end{aligned}$$

En développant le terme de droite

$$\begin{aligned}
\hat{T}(n) &= 2 - \frac{1}{n} + \frac{(n+1)}{n}\left(2 - \frac{1}{n-1} + \frac{n}{(n-1)}\hat{T}(n-2)\right) \\
&= 2\left(\frac{n+1}{n+1} + \frac{n+1}{n} + \cdots\right) - \left(\frac{n+1}{(n+1)n} + \frac{n+1}{n(n-1)} + \cdots\right) + \frac{n+1}{2}\hat{T}(1) \\
&= 2(n+1)\sum_{i=1}^{n+1}\frac{1}{i} - (n+1)\sum_{i=1}^{n}\frac{1}{(i)(i+1)} + \frac{n+1}{2}
\end{aligned}$$

Le premier terme est de l'ordre de $2(n+1)\log(n+1)$, le deuxième terme comme le troisième est un $O(n)$. Cela nous donne la complexité en moyenne du quicksort en $O(n \log n)$.

0.7 Quelques exercices détaillés

0.7.1 Complexité de calcul de la suite de Fibonacci

La version itérative du calcul de cette suite, définie par $u_{n+2} = u_{n+1} + u_n$, $u_0 = 0$, $u_1 = 1$, est clairement en $\Theta(n)$. Le temps d'exécution $T(n)$ de sa version récursive donne :

$$\begin{aligned}
T(n) &= 1 + T(n-1) + T(n-2) \\
&= 1 + 1 + 2T(n-2) + T(n-3) \\
&= 1 + 1 + 2 + 3T(n-3) + 2T(n-4) \\
&= 1 + 1 + 2 + 3 + 5T(n-4) + 3T(n-5) \\
&= 1 + \ldots + u_i + u_{i+1}T(n-i) + u_iT(n-i-1)
\end{aligned}$$

On montre facilement que $1 + \ldots + u_i = u_{i+2} - 1$. D'où

$$\begin{aligned}
T(n) &= u_{i+2} - 1 + u_{i+1}T(n-i) + u_iT(n-i-1) \\
&= u_{n+1} - 1 + u_nT(1) + u_{n-1}T(0) \\
&= u_{n+2} - 1
\end{aligned}$$

Sachant que $u_n \approx \frac{1}{\sqrt{5}}\left(\frac{1+\sqrt{5}}{2}\right)^n$, on en déduit que $T(n) = \Theta(\left(\frac{1+\sqrt{5}}{2}\right)^{n+2})$, ce qui est quand même assez coûteux !

0.8 Complexité en espace

A faire.

0.9 Théorie de la complexité des algorithmes

A faire.

1 Analyse des fonctions récursives

On suppose connu les points suivants en analyse d'algorithme :
— notations asymptotiques $\mathcal{O}$, Θ, Ω ;
— complexité en temps en pire cas pour des programmes non-récursifs

On abordera ici essentiellement la complexité en pire cas des algorithmes récursifs, dans les cas relativement simples (récurrences linéaires). Ensuite, on s'intéressera à la complexité amortie. Enfin on parlera un peu d'analyse en moyenne, même si on ne pourra pas rentrer dans les détails de la génération uniforme.

1.1 Complexité en pire cas des fonctions récursives

La complexité en pire cas d'une fonction récursive $f(\chi)$ s'étudie en général en associant (au moins) un paramètre n au temps d'exécution T. Les *paramètres* ou *contexte* χ de f sont réduits à ce paramètre n de manière à simplifier l'étude du temps d'exécution de f. Ce paramètre n est construit de manière à être décroissant à chaque appel de f. De plus, lorsque $n \leq cst$, la récursion doit se terminer. On pourra alors exprimer le temps d'exécution en pire cas de f comme une suite T, où $T(0), \dots T(cst)$ ont des valeurs données, et $T(n)$ s'exprime en fonction de valeurs de la suite T plus petites.

Quelques exemples :

```
// Calcul de la factorielle d'un entier m
Fonction FACT( E m : entier ) : entier ;                    // Ici, on choisit simplement n = m
début
    si m = 0 alors
        Retourner 1 ;                                       // On a facilement : T(0) = Θ(1)
    sinon
        Retourner m*FACT(m − 1) ;                           // et T(n) = Θ(1) + T(n − 1)
```

```
// Calcul du m-ième terme de la suite de Fibonacci
Fonction FIB( E m : entier ) : entier ;                     // Ici, on choisit simplement n = m
début
    si m = 0 alors
        Retourner 0 ;                                       // On a facilement : T(0) = Θ(1)
    sinon si m = 1 alors
        Retourner 1 ;                                       // On a facilement : T(1) = Θ(1)
    sinon
        Retourner FIB(m − 1)+FIB(m − 2) ;                   // et T(n) = Θ(1) + T(n − 1) + T(n − 2)
```

```
// Calcul du maximum d'un tableau T entre deux bornes a ≤ b
Fonction MAX( E T : tableau de Elem, E a,b :entier ) : Elem ;
 // Ici, on choisit n = b − a
début
    si a = b alors
        Retourner T[a] ;                                    // On a facilement : T(0) = Θ(1)
    sinon
        Retourner Max2(T[a], MAX(T, a + 1, b) ;             // et T(n) = Θ(1) + T(n − 1)
```

```
// Recherche dichotomique de x dans un tableau T entre deux bornes a ≤ b
Fonction DICHO( E T : tableau de Elem, E a,b :entier, x : Elem ) : booléen ;
 // Ici, on choisit n = b − a
début
    si a = b alors
        Retourner T[a] = x ;                                // On a facilement : T(0) = Θ(1)
    sinon
        m ← ⌊(a+b)/2⌋ ;
        si x ≤ T[m] alors
            Retourner DICHO(T, a, m, x) ;                   // et T(n) = Θ(1) + T(⌊n/2⌋)
        sinon
            Retourner DICHO(T, m + 1, b, x) ;               // et T(n) = Θ(1) + T(⌊n/2⌋)
```

1.2 Récurrences linéaires simples

Dans beaucoup de cas, on peut se ramener à une formule du genre :

$$\begin{cases} T(0) = \mathcal{O}(1), T(1) = \mathcal{O}(1), \text{etc}, T(n_0) = \mathcal{O}(1) \\ \forall n \geq n_0, T(n+n_0+1) = a_{n_0} T(n+n_0) + a_{n_0-1} T(n+n_0-1) + \ldots + a_0 T(n) + f(n) \end{cases} \quad (1)$$

Exemples :

Fonction	n_0	$T(0), \ldots, T(n_0)$	récurrence
Fact	0	$T(0) = \mathcal{O}(1)$	$T(n+1) = T(n) + \mathcal{O}(1)$
Fib	1	$T(0) = T(1) = \mathcal{O}(1)$	$T(n+2) = T(n+1) + T(n) + \mathcal{O}(1)$
Max	0	$T(0) = \mathcal{O}(1)$	$T(n+1) = T(n) + \mathcal{O}(1)$
Dicho	0	$T(0) = \mathcal{O}(1)$	$T(n+1) = T(\lfloor \frac{n+1}{2} \rfloor) + \mathcal{O}(1)$

On voit que Dicho a un comportement différent des précédentes.

1.2.1 La forme $T(n+1) = aT(n) + 1$

Ici, $n_0 = 0$. On la découpe en deux en regardant seulement $T'(n+1) = aT'(n)$ dans un premier temps.

- Il vient facilement :
  $T(n+1) = aT(n)+1 = a^2T(n-1)+a+1 = a^3T(n-2)+a^2+a+1 = \ldots = a^nT(0)+a^{n-1}+\ldots+a+1$
- Si $a = 1$, on a immediatement $\forall n \geq 1, T(n) = 1 + \ldots + 1 = \mathcal{O}(n)$.
- Sinon, il vient facilement (avec T(0)=1) : $T(n+1) = (a^{n+1}-1)/(a-1)$. C'est donc une série géométrique de raison a.

Lorsque $a > 1$, l'algorithme prend un temps $\mathcal{O}(a^n)$. C'est par exemple le cas des algorithmes récursifs :

- calcul des coefficients binomiaux par relation de récurrence
- de l'algorithme du sac-à-dos.
- résolution des tours de Hanoi

1.2.2 La forme $T(n+1) = aT(n) + f(n)$

Cela dépend de $f(n)$ et doit être traité au cas par cas.

1.2.3 La forme $T(n+2) = aT(n+1) + bT(n) + 1$

Ici, $n_0 = 1$. On la découpe en deux en regardant seulement $T'(n+2) = aT'(n+1) + bT'(n)$ dans un premier temps. On va chercher les racines r_1 et r_2 du polynôme $X^2 - aX - b$.

- Si $r_1 \neq r_2$ et réels, il est facile de vérifier que $\lambda r_1^n + \mu r_2^n$ satisfont la relation de récurrence. On détermine alors λ et μ avec les valeurs $T(0)$ et $T(1)$.
- les autres cas ne se produisent pas *a priori* du fait que ce sont des temps de calcul (a et b sont positifs).

On peut par exemple vérifier la formule sur Fib. Le discriminant vaut $\sqrt{5}$ d'où les deux racines $r_1 = \frac{1+\sqrt{5}}{2}$ et $r_2 = \frac{1-\sqrt{5}}{2}$. On reconnait r_1 comme étant le nombre d'or. En prenant $T(0) = T(1) = 1$, on trouve $\lambda = \frac{\sqrt{5}+1}{2\sqrt{5}}$ et $\mu = \frac{\sqrt{5}-1}{2\sqrt{5}}$. D'où :

$$T'(n) = \frac{\sqrt{5}+1}{2\sqrt{5}} \left(\frac{1+\sqrt{5}}{2} \right)^n + \frac{\sqrt{5}-1}{2\sqrt{5}} \left(\frac{1-\sqrt{5}}{2} \right)^n$$

On voit alors que le deuxième terme tend vers 0. On déduit que $T'(n) = \mathcal{O}\left(\frac{1+\sqrt{5}}{2}\right)^n$.

Le deuxième terme "+1" correspond en fait à la même somme ! On déduit alors que $T(n)$ suit la même loi que $T'(n)$.

NB : Le nombre d'or $\Phi \approx 1,618$ est plus petit que 2. On aurait pu montrer très facilement que $T(n) = \mathcal{O}(2^n)$ avec la section précédente. Là, on montre que $T(n) = \Theta(\Phi^n)$ ce qui est bien plus précis.

1.3 Autres formes classiques

Il est très fréquent d'avoir des algorithmes de la forme "diviser pour régner". Ceci est par exemple le cas du tri fusion (voir Algorithme 1). On observe une récurrence de la forme $T(n) = T(\lceil \frac{n}{2} \rceil) + T(\lfloor \frac{n}{2} \rfloor) + \Theta(n)$. En réalité les bornes inférieures ou supérieures ne changent pas la complexité finale.

Algorithme 1 : Tri fusion.

```
// Divise en O(n)
Action DIVISE( E T : tableau de Elem, E n : entier, S T1,T2 : tableau de Elem, S n1,n2 ) ;
Var : i : entier ;
début
    i ← 0, n1 ← 0, n2 ← 0 ;
    tant que i < n faire
        T1[n1] ← T[i] ;
        n1 ← n1 + 1, i ← i + 1 ;
        si i < n alors
            T2[n2] ← T[i] ;
            n2 ← n2 + 1, i ← i + 1 ;
// Fusionne en O(n1) + O(n2) = O(n)
Action FUSIONNE( E T1,T2 : tableau de Elem, E n1,n2, S T : tableau de Elem, E n : entier)
// Tri fusion
Action TRIFUSION( E T : tableau de Elem, E m : entier ) ;        // Ici, on choisit n = m
Var : T1,T2 : Tableau de Elem ;
n1,n2 : entier ;
début
    si m > 1 alors
        DIVISE( T,m,T1,T2,n1,n2 ) ;                    // On a facilement : T(0) = Θ(1)
        TRIFUSION( T1,n1 ) ;                           // et T(n) = Θ(n) + T(⌈n/2⌉)
        TRIFUSION( T2,n2 ) ;                           //       +T(⌊n/2⌋) + Θ(n)
        FUSIONNE( T1,T2,n1,n2,T,m ) ;
```

On dispose du théorème général suivant :

Théorème 1 *Soient $a \geq 1$ et $b > 1$ deux constantes, soit $f(n)$ une fonction et soit $T(n)$ définie pour les entiers non négatifs par la récurrence*

$$T(n) = aT(n/b) + f(n),$$

où n/b peut être interprétée comme $\lceil \frac{n}{b} \rceil$ ou $\lfloor \frac{n}{b} \rfloor$. De plus $T(0) = O(1)$. Alors $T(n)$ peut être bornée asymptotiquement ainsi :

1. *si $f(n) = \mathcal{O}(n^{\log_b a - \epsilon})$ pour une constante $\epsilon > 0$, alors $T(n) = \Theta(n^{\log_b a})$,*
2. *si $f(n) = \Theta(n^{\log_b a})$, alors $T(n) = \Theta(n^{\log_b a} \log n)$,*
3. *si $f(n) = \Omega(n^{\log_b a + \epsilon})$ pour une constante $\epsilon > 0$, et si $af(n/b) \geq cf(n)$ pour une certaine constante $c < 1$ et pour tout n assez grand, alors $T(n) = \Theta(f(n))$.*

Dit autrement, dans le cas 1, c'est le terme de gauche et l'initialisation qui dominent. Dans le cas 2, les deux ont une influence commune et cela induit un terme logarithmique. Dans le cas 3, c'est la fonction f qui domine largement. On peut voir une preuve de ce théorème dans (Corben *et al.*, 2004).

Qu'est-ce que cela donne sur le TRIFUSION ? Prenez $a = 2, b = 2, f(n) = \Theta(n) = \Theta(n^{\log_2 2})$, c'est donc le cas 2, et on obtient la complexité connue en $T(n) = \Theta(n \log n)$.

Quelques questions :

— Peut-on analyser QUICKSORT ainsi ?

— Dans TRIFUSION, comptez précisément le nombre de fois où une valeur est copiée. Cela vous pourrait-il plus ou moins qu'un QUICKSORT ? Même question pour le nombre de comparaisons.

— Utiliser ce théorème pour montrer les bornes asymptotiques dans les cas suivant : $T_1(n) = 4T_1(n/2) + n, T_2(n) = 4T_2(n/2) + n^2, T_3(n) = 4T_3(n/2) + n^3$.
— Peut-on appliquer le théorème précédent pour une récurrence de la forme $T(n) = 2T(n/2) + n \log n$?

1.4 Analyse asymptotique expérimentale

Il est dans certains cas difficile d'obtenir des formules explicites pour la complexité d'un algorithme. Dans d'autres cas, il peut être utile de deviner quelle est la complexité théorique pour pouvoir mieux la prouver. Une approche possible est l'expérimentation. On mesure alors le temps d'exécution d'un programme pour des valeurs n de plus en plus grandes, ce qui donnent m couples $(n_i, T(n_i))$ où T est la fonction mesurant le temps d'exécution, que l'on suppose triés dans l'ordre croissant des n_i.

Sauf peut-être dans le cas où T est linéaire, un simple tracé de ces points n'est souvent pas suffisant pour deviner la complexité asymptotique de T. Nous disposons néanmoins d'un moyen pour déterminer la complexité de T dans le cas où T a la forme $T(n) = an^b$. En effet, en prenant le logarithme de T, nous obtenons :

$$\begin{aligned} T(n) &= an^b \\ \Leftrightarrow \log(T(n)) &= \log a + b \log n \\ \Leftrightarrow y &= cst + bx, \end{aligned}$$

en choisissant $x = \log n$ et $y = \log(T(n))$. Prendre le logarithme des valeurs sur chaque axe, c'est se placer en échelle logarithmique. Dans cette échelle, le graphe de T est donc une droite de pente b où b est la puissance de la complexité. Il ne reste plus qu'à estimer cette pente sur le graphe pour déterminer une borne possible de la complexité de l'algorithme.

NB : il est important de prendre des n suffisamment grands pour que ce soit bien le comportement asymptotique qui domine dans le temps mesuré. Evidemment, cette méthode ne *prouve* rien. Elle donne des éléments de réflexion pour approfondir l'analyse.

En résumé : Analyse des fonctions récursives
— Pour analyser la complexité d'une fonction récursive, on se ramène à une suite récurrente paramétrée par un seul entier n, qui est généralement lié à la taille du problème.
— On dispose ensuite d'outils standards pour expliciter les complexités asymptotiques de la plupart des suites, comme les suites récurrentes linéaires ou les suites "diviser pour régner".

2 Analyse amortie des algorithmes

Dans une *analyse amortie*, le temps requis pour effectuer une suite d'opérations sur une structure de données est une moyenne sur l'ensemble des opérations effectuées. Chaque opération peut être coûteuse, mais l'analyse amortie permet de montrer que le coût moyen de chaque opération est faible (bien sûr si tel est vraiment le cas). Une analyse amortie est différente de l'analyse en moyenne car il n'est ici nul question de chance ou de probabilités. L'analyse amortie garantit les performances en moyenne de chaque opération dans le *cas le plus défavorable.* [2]

2.1 Méthode de l'agrégat

Dans la *méthode de l'agrégat* on montre que, pour tout n, une suite de n opérations prend le temps total $T(n)$ dans le pire cas. Dans ce pire cas, le coût moyen ou **coût amorti** par opération est $T(n)/n$. Ce coût amorti est donc le même quelque soit l'opération effectué dans la séquence (car la séquence peut contenir plusieurs types d'opérations).

2. Une partie de ce chapitre est un résumé de Corben *et al.*

2.1.1 Exemple sur des opérations de Pile.

On se donne les opérations suivantes sur les Piles : EMPILER(S, x), DÉPILER(S), PILEVIDE(S) et MULTIDÉPILER(S, k) qui appelle k fois DÉPILE(S) ou s'arrête si la pile est vide. Pour simplifier, on supposera donc que les trois premières méthodes ont chacune un coût de 1, la dernière a en revanche un coût en $\mathcal{O}(k)$.

Il est clair qu'une suite de n opérations EMPILER et DÉPILER dans un ordre quelconque a un coût n (avec un temps d'exécution en temps réel en $\Theta(n)$). Le coût d'un appel à MULTIDÉPILER est en revanche en $\min(s, k)$ où s est le nombre d'éléments dans la pile.

Prenons une pile initialement vide. Effectuons une suite de n opérations arbitraires EMPILER, DÉPILER, et MULTIDÉPILER. La taille de la pile étant en pire cas de n, et une opération MULTIDÉPILER pouvant avoir aussi un coût $\Theta(n)$, il vient vite que les n opérations ont un coût en $\mathcal{O}(n^2)$.

Cette borne est largement surestimée, tout simplement car on a regardé le pire cas d'une opération mais pas de toutes les opérations. Si on regarde les n opérations, on sait qu'il ne peut pas y avoir plus d'appels à DÉPILER qu'à EMPILER. La somme de tous les appels DÉPILER faits directement ou par MULTIDÉPILER ne peut excéder le nombre d'éléments empilés, qui est $\leq n \in \Theta(n)$. On obtient alors la complexité en pire cas de n opérations dans $\Theta(n)$.

Le **coût amorti** de MULTIDÉPILER est donc bien de $\Theta(n)/n = \Theta(1)$.

2.1.2 Exemple sur un compteur binaire.

On se donne maintenant un compteur binaire implementé comme un tableau $A[0..k-1]$ de bits, avec $long(A) = k$. Un tel tableau correspond au codage binaire d'un nombre $x = \sum_{i=0}^{k-1} A[i] \cdot 2^i$. Au départ $x = 0$. Pour ajouter 1 modulo 2^k, on utilise l'action suivante :

Algorithme 2 : Fonction INCRÉMENTER pour compteur binaire.

```
[!htp] Action INCRÉMENTER(ES A );
début
    i ← 0;
    tant que i < long(A) et A[i] = 1 faire
        A[i] ← 0;
        i ← i + 1;
    si i < long(A) alors A[i] ← 1;
```

Le coût de INCRÉMENTER est donc proportionnel au nombre de bits basculés. Une séquence de n appels à INCRÉMENTER prend donc $\Theta(nk)$ opérations dans le cas le plus défavorable. Mais ceci surestime la borne. En réalité, on peut borner le nombre de fois où chaque bit bascule.

On obtient assez facilement que le nombre *total* de bits basculés est de l'ordre de $2n$. Le coût amorti de INCRÉMENTER est donc de $\Theta(1)$.

2.2 Méthode comptable

Dans la **méthode comptable** on affecte des coûts différents à chaque opération, certaines recevant un coût supérieur et d'autres un coût inférieur à leur coût réel. Chacun de ces coût définit le *coût amorti* de l'opération. Lorsque pour une opération donnée, le coût amorti excède le coût réel, la différence est affectée à un objet de la structure comme crédit pour une opération future. On voit donc que dans cette méthode les coût amortis des opérations sont potentiellement différents, contrairement à la méthode de l'agrégat où le coût amorti est réparti équitablement.

La difficulté est donc de bien choisir les coûts amortis. Si on note c_i le coût réel de la i-ème opération et $\hat{c}_i$ son coût amorti, il faut faire en sorte que pour toutes les séquence de n opérations, on ait

$$\sum_{i=1}^{n} \hat{c}_i \geq \sum_{i=1}^{n} c_i. \tag{2}$$

D'après (2), il faut faire attention à ce que le crédit total stocké ne devienne jamais négatif.

2.2.1 Exemple sur des opérations de Pile.

Pour illustrer la méthode, voilà les coûts réel et coûts amortis choisis pour les opérations sur la pile :

Opération	coût réel	coût amorti
EMPILER(S, x)	1	2
DÉPILER(S)	1	0
MULTIDÉPILER(S, k)	$\min(s, k)$	0

Pour simplifier, on considère que PILEVIDE ne coûte rien. Il est clair qu'à chaque fois qu'on empile une valeur, on dispose d'un crédit de 1 associé à cette valeur empilée. Donc, lorsqu'on dépile avec DÉPILER, la valeur qu'on dépile dispose toujours d'un crédit de 1. Ce crédit est donc un acompte pour payer le coût de son dépilement. Lorsqu'on dépile, le crédit permet de payer exactement la différence coût réel moins coût amorti.

De même, lors d'un appel à MULTIDÉPILER, il y aura toujours un crédit de 1 par valeur dépilée, et donc le coût réel $\min(s, k)$ sera toujours compensé par ces crédits. Donc, pour une séquence quelconque de n opérations EMPILER, DÉPILER et MULTIDÉPILER, le coût amorti total est un majorant du coût total d'après (2). Or ce coût est un $\mathcal{O}(n)$.

Exercices

1. On affecte les coûts réels suivants aux fonctions de pile.

Opération	coût réel	coût amorti
EMPILER(S, x)	e	?
DÉPILER(S)	d	?
MULTIDÉPILER(S, k)	$m + d\min(s, k)$	?

   Que faut-il choisir comme coûts amortis pour que le raisonnement précédent continue à fonctionner ?

2. On rajoute une fonctionnalité à la pile qui est que toutes les k opérations, on fait une copie de sauvegarde de toute la pile. On suppose de plus que la pile ne dépasse jamais k valeurs. Montrer alors, en choisissant bien les coûts amortis, que le coût amorti total de n opérations (sauvegardes incluses) est bien $\mathcal{O}(n)$.

2.2.2 Exemple sur un compteur binaire.

Pour le compteur binaire, le principe est similaire. A chaque fois que l'on bascule un bit à 1, on met un coût amorti de 2 alors que le coût réel est 1. A chaque fois que l'on rebasculera ce bit à 0, on aura donc un crédit de 1, qui correspond à son coût réel. Comme le nombre de bits à 1 n'est jamais négatif, le crédit n'est jamais négatif. Le coût total d'un séquence de n INCRÉMENTER est donc de $\mathcal{O}(n)$.

Exercices

1. Imaginons qu'on rajoute une méthode RÉINITIALISER qui remet le compteur à 0. Comment faire évoluer la structure de compteur pour qu'une séquence quelconque de n opérations INCRÉMENTER ou RÉINITIALISER prenne un temps $\mathcal{O}(n)$. On supposera le compteur initialement à 0.

2. Montrer que l'on peut implémenter une file avec deux piles ordinaires, de telle manière que le coût amorti de chaque opération ENFILER et DÉFILER soit $O(1)$.

3. Imaginer une structure de données qui permettent les opérations suivantes pour un ensemble S d'entiers :
   — INSÉRER(S, x) insère x dans l'ensemble S.
   — SUPPRIMEMOITIÉSUP(S) supprime les $\lceil \frac{\text{Card}(S)}{2} \rceil$ plus grands éléments de S.
   Comment implémenter cette structure pour que toute séquence de m opérations soit exécutée en temps $\mathcal{O}(m)$.

2.3 Méthode du potentiel

La méthode du potentiel considère un crédit global (ou potentiel) attaché à toute la structure plutôt que de répartir le crédit sur chacun des éléments de la structure. Ce potentiel peut donc servir à payer des opérations futures. On construira ce potentiel de manière à ce qu'il soit toujours positif ou nul.

Soit D_0 notre structure de données initiale sur laquelle les n opérations sont effectuées. Pour chaque i-ème opération sur la structure D_{i-1}, soit c_i son coût réel et D_i la structure résultante. La fonction *potentiel* Φ associe un nombre réel à chaque structure D_i. On définit le **coût amorti** $\hat{c}_i$ de la i-ème opération par :

$$\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}). \tag{3}$$

Il est facile de voir que le **coût amorti total** est :

$$\sum_{i=1}^{n} \hat{c}_i = \sum_{i=1}^{n} c_i + \Phi(D_n) - \Phi(D_0). \tag{4}$$

Si Φ est construite de manière à ce que $\Phi(D_i) \geq \Phi(D_0)$ alors le coût amorti total est un majorant du coût réel total. Le résultat dépend donc de la fonction Φ choisie. Il faudra donc "sentir" quelle est la bonne fonction Φ pour obtenir la complexité voulue.

On calculera ainsi le coût amorti de chaque opération possible, on sommera ces coûts amortis, ce qui nous donnera un majorant du coût réel total en pire cas.

2.3.1 Exemple sur des opérations de Pile.

On définit simplement le potentiel Φ d'une pile comme étant le nombre d'éléments de la pile. On a donc évidemment $\Phi(D_0) = 0$ si la pile est vide au début et $\forall i, \Phi(D_i) \geq 0$. On est donc bien dans les conditions où le coût amorti est à tout moment un majorant du coût réel.

On vérifie les coûts amortis :

— Empiler : $\hat{c}_i = 1 + 1$ (car la pile augmente de 1)

— Dépiler : $\hat{c}_i = 1 - 1$ (car la pile diminue de 1)

— MultiDépiler : $\hat{c}_i = k' - k'$ (si $k' = \min(k, s)$, car la pile diminue de k')

Comme le coût amorti de chaque opération est un $O(1)$, il est linéaire pour n'importe quelle séquence d'opération.

2.3.2 Exemple sur un compteur binaire.

On prend cette fois-ci pour $\Phi(D_i)$ le nombre de bits à 1 dans la structure D_i.

Si l'appel de Incrémenter a réinitialisé t_i bits, le coût réel de l'opération est au plus de $1 + t_i$ (car il met t_i bits à 0 et 1 bit à 1). Si $b_i = 0$ alors $b_{i-1} = t_i = k$. Si $b_i > 0$ alors $b_i = b_{i-1} - t_i + 1$. On vérifie que $b_i \leq b_{i-1} - t_i + 1$ dans tous les cas. La différence de potentiel est :

$$\begin{aligned} \Phi(D_i) - \Phi(D_{i-1}) &= b_i - b_{i-1} \\ &\leq b_{i-1} - t_i + 1 - b_{i-1} \\ &= 1 - t_i. \end{aligned}$$

Le coût amorti vaut alors :

$$\begin{aligned} \hat{c}_i &= c_i + \Phi(D_i) - \Phi(D_{i-1}) \\ &\leq (t_i + 1) + (1 - t_i) \\ &= 2. \end{aligned}$$

On peut même analyser le coût amorti lorsqu'on part d'un compteur à une valeur arbitraire. On trouve que le coût global :

$$\sum_{i=1}^{n} c_i \leq 2n - b_n + b_0.$$

Ce qui permet de conclure quand même sur la linéarité de n incrémentations si $k = O(n)$.

En résumé : Analyse amortie des algorithmes

— L'analyse amortie permet de déterminer de façon plus fine la complexité en pire cas d'une séquence d'opérations, notamment dans le cas où les opérations n'ont pas un coût forcément constant.

— On dispose de trois méthodes standards pour l'analyse amortie : (i) la méthode de l'agrégat détermine un majorant pour le temps d'exécution de toute la séquence en pire cas, (ii) la méthode comptable affecte des coûts différents à chaque opération considérée et permet de mettre de côté du temps pour des opérations ultérieures plus coûteuses, (iii) la méthode du potentiel attribue un crédit global à toute la structure, ce crédit pouvant être dépensé ultérieurement dans d'autres opérations.

3 Structures de données pour ensembles disjoints (Union-Find)

Une structure de données d'ensembles disjoints gère une collection $S = \{S_1, \ldots, S_k\}$ d'ensembles dynamiques disjoints (les ensembles sont touts deux à deux disjoints). Chaque ensemble S_i est identifié par un *représentant* qui appartient à l'ensemble. Savoir si deux éléments appartiennent au même ensemble revient donc à déterminer si ils ont le même représentant.

Ces structures de données permettent de représenter les classes d'équivalence dans les relations. Une application courante est la détermination des composantes connexes d'un graphe. Une autre est de déterminer si l'ajout d'une arête à un arbre couvrant induit un cycle.

Exemples :

— On examine un ensemble de n personnes et on veut le décomposer en groupes de personnes appartenant à la même ville (une relation d'équivalence). On veut ensuite les regrouper par intercommunalité, puis par département, et enfin région. Les structures pour ensembles disjoints sont efficaces pour ces requêtes.

— L'algorithme de Kruskal de calcul de l'arbre couvrant de poids minimal utilise aussi cette structure. Elle permet de détecter si l'ajout d'une arête crée un cycle dans un graphe, simplement en regardant si ses deux extrémités sont déjà dans le même ensemble.

3.1 Opérations requises sur ces structures

Les ensembles sont composées d'éléments appelés objets. Chaque ensemble est représenté par l'un de ses objets, qui le caractérise puisque tous les ensembles sont disjoints deux à deux. On note que l'on ne manipulera qu'une seule structure par ensemble disjoint à la fois, elle sera donc implicite dans les fonctions ci-dessous. On veut disposer des opérations suivantes :

— **Action** Créer-Ensemble*(x)* crée un nouvel ensemble dont le seul membre et représentant est x. Attention, il faut que x ne soit pas déjà dans un autre ensemble.

— **Action** Union*(x, y)* réunit les ensembles dynamiques qui contiennent x et y, mettons S_x et S_y, dans un nouvel ensemble qui est l'union des deux. Le représentant de ce nouvel ensemble est un élément quelconque de $S_x \cup S_y$ (c'est souvent l'un des représentants de S_x ou S_y). Les ensembles S_x et S_y n'existent plus dans la collection d'ensembles S.

— **Fonction** Trouver-Ensemble*(x)* retourne un pointeur vers le représentant de l'unique ensemble contenant l'objet x.

On pourrait vouloir diviser des ensembles, mais cela n'induit pas de difficultés algorithmiques spécifiques. En gros, la division d'un ensemble en un certain nombre de sous-ensembles est proportionnel à son nombre d'éléments (car il faut les désigner). On ne se préoccupe pas de ce type d'opérations ici.

Dans toute la suite, l'analyse des algorithmes se fera en fonction du nombre n d'appels à Créer-Ensemble et de m le nombre total d'opérations Créer-Ensemble, Union et Trouver-Ensemble.

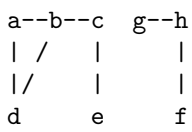
3.2 Le calcul des composantes connexes

Il s'agit de l'application la plus connue des structures pour ensembles disjoints. Etant donné un graphe, on veut savoir le nombre de composantes connexes, ou effectuer des réquêtes pour savoir si deux sommets sont dans la même composante connexe. Une approche sera de faire un parcours en largeur à partir d'un sommet, mais cette approche devient très vite coûteuse si on a plusieurs couples de sommets à tester. Il est alors plus intéressant de construire les ensembles disjoints qui correspondent aux composantes connexes du graphe. De plus, si on rajoute des arêtes au graphe, ces structures sont mises à jour grâce à des appels à UNION. L'algorithme basé sur les ensembles disjoints s'écrit ainsi. On fait d'abord le précalcul avec l'appel de COMPOSANTES-CONNEXES, puis on appelle autant de fois que l'on souhaite MÊME-COMPOSANTE.

```
// Calcule les ensembles disjoints correspondant aux composantes connexes d'un
   graphe
Action COMPOSANTES-CONNEXES( E G : Graphe );
Var : u,v : Sommet
début
    Pour chaque sommet v de G Faire
        CRÉER-ENSEMBLE(v);
    Pour chaque arête (u, v) de G Faire
        si TROUVER-ENSEMBLE(u) ≠ TROUVER-ENSEMBLE(v) alors
            UNION(u, v);
```

```
// Retourne vrai si u et v sont dans la même composante
Fonction MÊME-COMPOSANTE( E u, v : Sommet ) : booléen ;
début
    Retourner TROUVER-ENSEMBLE(u) = TROUVER-ENSEMBLE(v) ;
```

L'exécution de COMPOSANTES-CONNEXES sur le graphe ci-dessous est faite pas à pas dans la Table 1.



Dans une implémentation réelle, il faudrait que l'objet représentant un sommet ait un pointeur vers l'objet correspondant dans la structure pour ensembles disjoints et vice-versa. Nous ignorons ces détails ici.

Exercices

1. Adaptez l'action COMPOSANTES-CONNEXES pour que ce soit une fonction qui retourne aussi le nombre de composantes connexes du graphe G.

3.3 Ensembles disjoints par listes chaînées

Chaque ensemble de S est une liste chaînée de ses objets. Le premier élément de la liste est le représentant de l'ensemble. Chaque élément a un pointeur vers son successeur et un pointeur vers le représentant de la liste (donc le premier élément de la liste). De plus chaque liste a un pointeur vers le premier élément plus un pointeur vers le dernier élément (pour concaténer les listes lors d'un UNION).

Arêtes testées	Ensembles disjoints							
au début	$\{a\}$	$\{b\}$	$\{c\}$	$\{d\}$	$\{e\}$	$\{f\}$	$\{g\}$	$\{h\}$
(b,a)	$\{a,b\}$		$\{c\}$	$\{d\}$	$\{e\}$	$\{f\}$	$\{g\}$	$\{h\}$
(d,a)	$\{a,b,d\}$		$\{c\}$		$\{e\}$	$\{f\}$	$\{g\}$	$\{h\}$
(a,b)	$\{a,b,d\}$		$\{c\}$		$\{e\}$	$\{f\}$	$\{g\}$	$\{h\}$
(c,b)	$\{a,b,c,d\}$				$\{e\}$	$\{f\}$	$\{g\}$	$\{h\}$
(d,b)	$\{a,b,c,d\}$				$\{e\}$	$\{f\}$	$\{g\}$	$\{h\}$
(b,c)	$\{a,b,c,d\}$				$\{e\}$	$\{f\}$	$\{g\}$	$\{h\}$
(e,c)	$\{a,b,c,d,e\}$					$\{f\}$	$\{g\}$	$\{h\}$
(a,d)	$\{a,b,c,d,e\}$					$\{f\}$	$\{g\}$	$\{h\}$
(b,d)	$\{a,b,c,d,e\}$					$\{f\}$	$\{g\}$	$\{h\}$
(c,e)	$\{a,b,c,d,e\}$					$\{f\}$	$\{g\}$	$\{h\}$
(h,f)	$\{a,b,c,d,e\}$					$\{f,h\}$	$\{g\}$	
(h,g)	$\{a,b,c,d,e\}$					$\{f,g,h\}$		
(f,h)	$\{a,b,c,d,e\}$					$\{f,g,h\}$		
(g,h)	$\{a,b,c,d,e\}$					$\{f,g,h\}$		

TABLE 1 – Exécution de COMPOSANTES-CONNEXES sur le graphe non orienté $G = \{\{a,b\},\{a,d\},\{b,c\},\{c,e\},\{g,h\},\{h,f\}\}$.

On note immédiatement que :
- Tout appel à CRÉER-ENSEMBLE coûte $\Theta(1)$ car il suffit de créer une liste à un élément.
- Tout appel à TROUVER-ENSEMBLE coûte $\Theta(1)$ car chaque élément a un pointeur vers son représentant. Il suffit donc de le consulter.
- Si UNION(x,y) est implémentée en déplaçant la liste de x à la fin de la liste de y, il faut mettre à jour les représentants de la liste de x. Cela prend donc un temps $\Theta(k)$ où k est la taille de la liste de x.

Il est facile de voir que n CRÉER-ENSEMBLE suivis de $n-1$ UNION donc un coût total en $\Theta(n^2)$, donc un coût amorti en $\Theta(n)$.

On peut baisser cette borne en utilisant l'*heuristique de l'union pondérée*. Le principe est le suivant. Chaque liste stocke aussi son nombre d'éléments. Lors d'un appel UNION(x,y) c'est toujours la liste la plus courte qui est concaténée à la plus longue. Grâce à ce petit test tout simple, on montre que :

Théorème 2 *Si les ensembles disjoints sont représentés par listes chaînées et qu'on utilise l'heuristique de l'union pondérée, une séquence arbitraire de m opérations* CRÉER-ENSEMBLE, TROUVER-ENSEMBLE *et* UNION, *dont n* CRÉER-ENSEMBLE *prend un temps $\mathcal{O}(m + n\log n)$.*

Preuve. Il suffit de calculer un majorant du nombre de fois où le pointeur vers le représentant d'un objet x est changé. Or chaque fois qu'il est mis à jour, l'objet x était dans le plus petit des ensembles. Donc la première fois, au plus petit, il était tout seul et a rejoint un ensemble de taille au moins 2. La deuxième fois, il rejoint un ensemble de taille au moins 4, etc, jusqu'à au maximum $\lceil \log_2 n \rceil$ fois (car l'ensemble le plus grand a une taille n maximum). □

3.4 Ensembles disjoints par forêts

Chaque ensemble sera maintenant représenté par un arbre, dont les nœuds sont les objets appartenant à l'ensemble et la racine est le représentant de l'ensemble. Chaque nœud d'un arbre a donc un pointeur vers son parent. La racine pointe vers elle-même. Les fonctions sur les ensembles disjoints sont mis en œuvre ainsi :
- Un appel de CRÉER-ENSEMBLE crée simplement un arbre a un seul nœud (coûte $\Theta(1)$).
- La fonction TROUVER-ENSEMBLE remonte d'un nœud jusqu'à sa racine, ce qui coûte $\Theta(k)$ où k est la profondeur du nœud.
- La fonction UNION(x,y) relie la racine de x à la racine de y. Pour ce faire, il suffit donc de remonter à la racine de chaque arbre et de modifier un pointeur.

Tel quel, une telle implémentation n'est pas plus rapide que la précédente. On applique deux heuristiques à cette structure de façon à la rendre la plus efficace possible.

L'union par rang. On stocke dans chaque nœud un majorant de la hauteur de son sous-arbre appelé *rang* (qui vaut 0 lorsque l'arbre est réduit à un élément). Lorsqu'on réalise l'union, c'est la racine de moindre rang qui pointe vers la racine de rang supérieur. C'est seulement lorsque les deux racines ont même rang qu'on augmente le rang. Il est clair que cette opération ne rajoute pas de surcoût en temps.

La compression de chemin. Dès que l'on utilise TROUVER-ENSEMBLE, tous les nœuds traversés pour trouver la racine, sont modifiés de façon à ce que leur parent soit directement la racine. Il est clair que cette opération ne rajoute pas de surcoût en temps.

On stocke le rang et le père d'un objet x dans des champs de x. Cela donne les pseudo-codes suivants :

Action CRÉER-ENSEMBLE*(**E** x : Objet) ;*
début
  $x.p \leftarrow x$;
  $x.rang \leftarrow 0$;
Action UNION*(**E** x, y : Objet) ;*
début
  LIER(TROUVER-ENSEMBLE(x),TROUVER-ENSEMBLE(y));
Action LIER*(**E** x, y : Objet) ;* `// x et y sont des racines.`
début
  si $x.rang > y.rang$ **alors** $y.p \leftarrow x$;
  else
    $x.p \leftarrow y$;
    si $x.rang = y.rang$ **alors** $y.rang \leftarrow y.rang + 1$;
Fonction TROUVER-ENSEMBLE*(**E** x : Objet) : Objet ;*
début
  si $x \neq x.p$ **alors**
    $x.p \leftarrow$ TROUVER-ENSEMBLE($x.p$)
  Retourner $x.p$

Ces heuristiques ont un effet très important sur la complexité amortie des opérations sur les ensembles disjoints. On a [Tarjan 1975] :

Théorème 3 *Si on utilise les forêts d'ensembles disjoints avec les heuristiques d'union par rang et de compression de chemin, alors une séquence arbitraire de m opérations* CRÉER-ENSEMBLE, TROUVER-ENSEMBLE *et* UNION, *dont n* CRÉER-ENSEMBLE *prend un temps $\mathcal{O}(m\alpha(n))$, où $\alpha(n)$ est une fonction qui croît extrêmement lentement ($n \leq 4$ dans tous les cas concevables).*

Preuve. Nous ne démontrerons pas cette borne. La preuve se fait en utilisant la méthode du potentiel. □

Qu'est-ce que $\alpha(n)$ et que vaut-il ? On définit d'abord la fonction à croissance très rapide ci-dessous :

$$\forall k \geq 0, j \geq 1, \quad A_k(j) = \begin{cases} j+1 & \text{si } k = 0, \\ A_{k-1}^{(j+1)}(j) & \text{si } k \geq 1. \end{cases} \tag{5}$$

On note qu'on utilise la notation exponentielle (j) pour exprimer qu'on affectue j la composée de la fonction. Le paramètre k est appelé *niveau* de la fonction A.

On montre facilement que :
— $\forall j \geq 1, A_1(j) = 2j + 1$.
— $\forall j \geq 2, A_2(j) = 2^{j+1}(j+1) - 1$.
— $A_3(1) = A_2^{(2)}(1) = A_2(A_2(1)) = A_2(7) = 2^8 \cdot 8 - 1 = 2047$
— $A_4(1) = A_3^{(2)}(1) = A_3(A_3(1)) = A_3(2047) = A_2^{(2048)}(2047) \gg A_2(2047) = 2^{2048} \cdot 2048 - 1$.

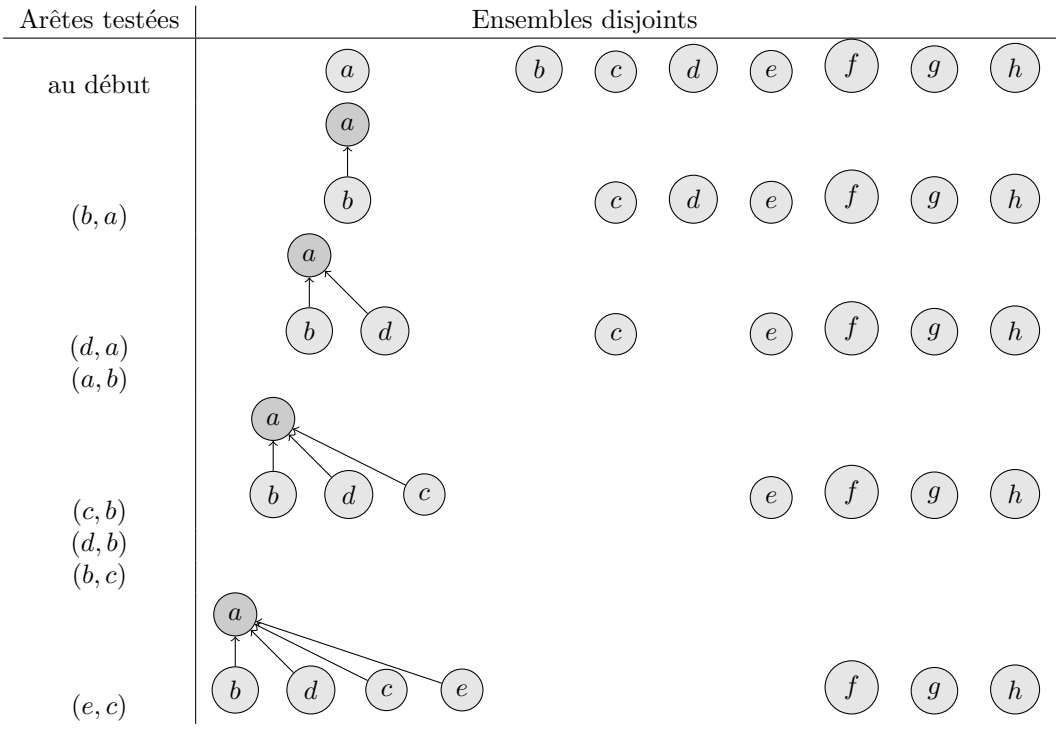


TABLE 2 – Une partie de l'exécution de COMPOSANTES-CONNEXES sur le graphe non orienté $G = \{\{a,b\},\{a,d\},\{b,c\},\{c,e\},\{g,h\},\{h,f\}\}$, avec les forêts d'ensembles disjoints. Le rang est donné par le niveau de gris.

j	1	2	3	4	5
$A_0(j)$	2	3	4	5	6
$A_1(j)$	3	5	7	9	11
$A_2(j)$	7	23	63	159	383
$A_3(j)$	2047	$\gg 2^{2^{27}}$			
$A_4(j)$	$\gg 2^{1000000}$				

La fonction $\alpha(n)$ est définie comme $\min\{k, A_k(1) \geq n\}$. Autrement dit :

$$\alpha(n) = \begin{cases} 0 & \text{pour } 0 \leq n \leq 2, \\ 1 & \text{pour } n = 3, \\ 2 & \text{pour } 4 \leq n \leq 7, \\ 3 & \text{pour } 8 \leq n \leq 2047, \\ 4 & \text{pour } 2048 \leq n \leq A_4(1). \end{cases}$$

La Table 2 donne une partie de l'exécution de l'algorithme COMPOSANTES-CONNEXES avec des forêts d'ensembles disjoints. On voit que la profondeur des arbres n'augmente vraiment pas.

Exercices

1. Essayez de déterminer un ensemble à 8 éléments et une séquence d'union qui maximise le rang d'un élément. Quel est le rang maximum que l'on peut atteindre ?
2. Peut-on dire que la hauteur de l'arbre est toujours limité par $\alpha(n)$? Si non, donnez un majorant.
3. Peut-on dire que le rang d'un sommet de l'arbre est toujours limité par $\alpha(n)$? Si non, donnez un majorant.
4. Soit l'ensemble $S = \{a, b, c, d, e, f, g, h\}$. Dessinez la forêt d'ensembles disjoints après :
   (a) UNION(b, a), UNION(d, c), UNION(c, a),
   (b) UNION(f, e), UNION(h, g), UNION(e, g),
   (c) UNION(g, a),

(d) TROUVER-ENSEMBLE(h)

5. Peut-on dire que tout appel à TROUVER-ENSEMBLE prend en pire cas $\alpha(n)$? Combien faut-il avoir d'éléments pour trouver une suite d'UNION qui construise un nœud avec une profondeur k ? En déduire une borne inférieure sur le nombre de nœuds d'un arbre dans la racine a rang k.
6. Si jamais un appel à TROUVER-ENSEMBLE est fait sur un sommet de profondeur k, combien de sommets voient leur profondeur réduit à 1 ? De façon plus générale, montrez que la somme des profondeurs de l'arbre diminue d'au moins $2^{k-1} - 1$.

---

4 Géométrie algorithmique

Ce domaine étudie les algorithmes pour résoudre des problèmes géométriques, souvent formulés dans l'espace Euclidien. On trouve beaucoup d'applications de ces algorithmes en conception assistée par ordinateur, infographie, analyse d'image, ingénierie (calcul numérique sur des structures), robotique, jeux vidéo. Nous ferons ici plutôt de la géométrie dans le plan, même si beaucoup de notions s'étendent à l'espace. Attention, les algorithmes deviennent souvent difficiles dans l'espace.

On supposera que nous disposons d'un type assez précis pour représenter les nombres réels, même si une grosse difficulté des algorithmes géométriques est souvent liées à des erreurs numériques. Notamment, on sent bien que la notion "un point est-il sur une droite" est très sensible à la moindre erreur numérique. Les implémentations bien faites d'algorithmes géométriques font très attention aux imprécisions et utilisent les stratégies suivantes :

— favoriser les opérations qui n'augmentent que peu la taille mémoire nécessaire pour représenter les nombres : + et − notamment, × si nécessaire, / à éviter.
— définir des zones de résultats où on sait qu'on n'a pas besoin d'être précis.
— utiliser des nombres à précision arbitraire
— ne pas utiliser des valeurs mais juste déterminer le bon signe

La bibliothèque CGAL est une bibliothèque C++ qui contient beaucoup d'algorithmes géométriques sûrs.

On peut aussi se placer dans le plan discret des entiers et ne manipuler que des entiers. Ce domaine de la géométrie est appelé géométrie discrète (*digital geometry* en anglais).

4.1 Notions élémentaires

On notera $\mathbb{R}^2$ le plan Euclidien. Un *point* $p = (x, y)$ du plan est caractérisé par ses coordonnées x et y, deux nombres réels. Un *segment* $[p_1, p_2]$ est un sous-ensemble du plan caractérisé par ses extrémités p_1 et p_2. Il est constitué de tous les points "entre" p_1 et p_2. Plus formellement, si $0 \leq \alpha \leq 1$, $p = \alpha p_1 + (1 - \alpha)p_2$ est entre p_1 et p_2. On dit que p est une *combinaison linéaire convexe* de p_1 et p_2. Le segment est donc l'ensemble des combinaisons linéaires convexes de p_1 et p_2. C'est d'ailleurs l'enveloppe convexe de p_1 et p_2.

La *droite* (p_1p_2) est définie comme le segment $[p_1, p_2]$, sauf que α peut prendre n'importe quelle valeur. Le *rayon* $[p_1p_2)$ est obtenu pour des valeurs $1 - \alpha \geq 0$, tandis que le rayon $(p_1p_2]$ est obtenu pour des valeurs $\alpha \geq 0$. Si l'ordre de p_1 et p_2 est important, on parlera du segment orienté $[\overrightarrow{p_1p_2}]$.

La longueur du segment $[p_1, p_2]$ est la distance euclidienne entre les deux extrémités p_1 et p_2, ou de manière équivalente la norme-2 du vecteur $p_1\vec{p}_2$. On utilise le théorème de Pythagore pour déterminer que

$$L([p_1, p_2]) = d(p_1, p_2) = \|p_2 - p_1\| = \sqrt{(p_2.x - p_1.x)^2 + (p_2.y - p_1.y)^2}.$$

Dans la suite on voudra déterminer si un point est à gauche ou à droite d'un segment orienté, si deux segments s'intersectent, etc. On voudra aussi déterminer quels sont les points les plus proches d'un autre, calculer des enveloppes convexes, etc.

4.2 Orientation et produit en croix

On veut déterminer si deux segments orientés $[\overrightarrow{p_1p_2}]$ et $[\overrightarrow{p_1p_3}]$ sont dans l'ordre trigonométrique, ou, dit autrement, si le point p_3 est "à gauche" du segment orienté $[\overrightarrow{p_1p_2}]$.

Une solution est de calculer l'équation de droite (p_1p_2) puis de vérifier la hauteur de p_3 par rapport à celle de la droite. C'est correct mathématiquement mais assez instable numériquement.

Une meilleure idée est le produit en croix. Soit deux vecteurs $\vec{v_1} = (x_1, y_1)$ et $\vec{v_2} = (x_2, y_2)$. Leur produit en croix (en fait leur déterminant) est :

$$\vec{v_1} \times \vec{v_2} = x_1y_2 - y_1x_2 = \det \begin{pmatrix} x_1 & x_2 \\ y_1 & y_2 \end{pmatrix}. \tag{6}$$

On note que $\vec{v_1} \times \vec{v_2} = -\vec{v_2} \times \vec{v_1}$. En fait, cette quantité représente l'aire *signée* du parallélogramme de côtés $\vec{v_1}$ et $\vec{v_2}$.

Pour revenir à l'orientation, il suffit donc de faire le produit en croix $p_1\vec{p}_2 \times p_1\vec{p}_3$. Si le signe est positif, p_3 est à gauche. Si le signe est négatif p_3 est à droite. Si le signe est 0, p_3 est sur la droite (p_1p_2).

Exercices

1. Soit la structure `Point` composée de deux champs `x` et `y`. Ecrire la fonction ORIENTATION(p_1, p_2, p_3) qui détermine l'orientation des points en calculant le produit en croix.
2. Soient deux segments consécutifs $[p_1, p_2]$ et $[p_2, p_3]$. Comment déterminer si on fait un virage à gauche ou à droite ou si on va tout droit ? Est-on obligé de déterminer l'angle ?

4.3 Comment déterminer si deux segments sont sécants ?

Un segment traverse une droite ssi ses deux extrémités sont de chaque côté. Deux segments se coupent ssi : 1) soit chaque segment traverse la droite contenant l'autre, ou 2) une extrémité d'un segment appartient à l'autre segment.

Cela nous donne donc le pseudo-code de l'Algorithme 3.

4.4 Polygones

Un *polygone* est une courbe du plan, refermée sur elle-même, composée d'une suite de segments de droites consécutifs appelés *côtés* du polygone. Les points reliant deux segments consécutifs sont les *sommets* du polygone. Un polygone est dit *simple* lorsque seuls les segments consécutifs s'intersectent. Il est clair que l'on peut caractériser un polygone par la séquence de ses sommets. Un polygone sera donc souvent représenté sous forme d'un tableau de points ou d'une liste de points.

Si le polygone est simple, par le théorème de Jordan, il coupe le plan en deux domaines dont l'un est fini, appelé *interieur* du polygone, l'autre est infini, appelé *estérieur*. Les segments eux-même forment le *contour* du polygone. Voilà quelques problèmes classiques que les outils précédents peuvent résoudre assez simplement.

Exercices

1. Si $P = (p_1, \ldots, p_n)$ est un polygone simple, montrer comment calculer son aire en un temps linéaire en n. Aide : que vaut ORIENTATION(p_1, p_i, p_{i+1}) ?
2. Donner un algorithme quadratique pour déterminer si un polygone est simple. Notez que le meilleur algorithme possible est dans $\Theta(n \log n)$.
3. Tout polygone simple peut être décomposé en union de triangles, i.e. *triangulé*. Il est possible d'exhiber une triangulation en temps linéaire (Tarjan 1991), mais l'algorithme est difficile. Un algorithme quadratique facile à écrire est basé sur le "découpage d'oreille". Tout polygone simple contient au moins deux oreilles, c'est-à-dire deux segments consécutifs qui tournent à gauche (en supposant que le contour du polygone va dans le sens trigonométrique). On met un

Algorithme 3 : Algorithme décidant si deux segments s'intersectent (temps constant).

```
// Sachant que r est sur la droite (pq), détermine si r appartient au segment [pq].
Fonction SUR-SEGMENT( E p, q, r : Point ) : booléen ;
début
    Retourner min(p.x, q.x) ≤ r.x ≤ max(p.x, q.x) et min(p.y, q.y) ≤ r.y ≤ max(p.y, q.y) ;
// Détermine si les segments [p1, p2] et [p3, p4] s'intersectent.
Fonction INTERSECTION-SEGMENTS( E p1, p2, p3, p4 : Point ) : booléen;
début
    d1 ← ORIENTATION(p3, p4, p1);
    d2 ← ORIENTATION(p3, p4, p2);
    d3 ← ORIENTATION(p1, p2, p3);
    d4 ← ORIENTATION(p1, p2, p4);
    si (d1 < 0 et d2 > 0) ou (d1 > 0 et d2 < 0) ou (d3 < 0 et d4 > 0) ou (d3 > 0 et d4 < 0)
     alors Retourner Vrai ;
    sinon si d1 = 0 et SUR-SEGMENT(p3, p4, p1) alors
        Retourner Vrai ;
    sinon si d2 = 0 et SUR-SEGMENT(p3, p4, p2) alors
        Retourner Vrai ;
    sinon si d3 = 0 et SUR-SEGMENT(p1, p2, p3) alors
        Retourner Vrai ;
    sinon si d4 = 0 et SUR-SEGMENT(p1, p2, p4) alors
        Retourner Vrai ;
    sinon Retourner Faux ;
```

triangle à chacune des oreilles, on enlève les deux sommets au fond de l'oreille et on relance l'algorithme. Ecrivez donc cet algorithme.

4. Un ensemble C est dit *convexe* ssi $\forall p, q \in C, [pq] \subset C$. On voit alors qu'un polygone simple et son intérieur forme un ensemble *convexe* ssi si les segments font toujours un virage à gauche.

   Si on a une liste de points $(p_0, \ldots, p_{n-1})$, est-il suffisant de tester ORIENTATION$(p_{i-1}, p_i, p_{i+1}) \geq 0$ pour tout sommet p_i ? Comment garantir que cela fonctionne ?

5. Ecrire une fonction HRAYON-INTERSECTE-SEGMENT qui retourne vrai si le rayon horizontal $[px)$ coupe le segment $[q, r]$. Adaptez INTERSECTION-SEGMENTS.

6. Un point p est à l'intérieur d'un polygone simple P ssi un rayon partant de lui intersecte un nombre impair de fois le contour. Utiliser la fonction précédente pour trouver un algorithme qui détermine si p est dans l'intérieur de P en temps linéaire.

4.5 Intersection dans une soupe de segments

Etant donné un ensemble de segments, on veut savoir s'il existe (au moins) une intersection. Le meilleur algorithme est en $\Theta(n \log n)$. C'est un algorithme utilisant une technique de balayage, assez classique en géométrie algorithmique.

Le principe est de trier les extrémités des segments suivant une direction, mettons x. Ensuite, on maintient une structure ordonnée T des segments actifs (intersectés par la droite de balayage courante), dont l'ordre est l'ordre des points d'intersection sur la droite de balayage. En général, on choisit un arbre rouge-noir ou un ABR pour cette structure.

Ensuite, deux segments s'intersectent si et seulement si à un moment dans le balayage ils sont côte à côte dans la structure et qu'ils s'intersectent. On a donc seulement à regarder à chaque insertion de segment (lorsqu'une extrémité gauche croise le balayage) et à chaque suppression de segment

(lorsqu'une extrémité droite croise le balayage) si autour de ce segment il y a une intersection. Il est inutile de regarder ailleurs.

Le code final requiert $O(n \log n)$ pour le tri initial, puis $\log n$ par extrémité traité à cause du temps pour réaliser l'insertion ou la suppression dans l'arbre T.

Exercices

1. Exhibez un ensemble de n segments qui induit le plus d'intersections possible entre ces segments.
2. En déduire la complexité en pire cas minimale d'un algorithme pour déterminer toutes les intersections d'un ensemble de segments.
3. Donnez un algorithme qui a cette complexité.

4.6 Convexité et enveloppe convexe

Dans n'importe quel espace euclidien (et donc notamment $\mathbb{R}^2$ et $\mathbb{R}^3$), un ensemble C de cet espace est dit *convexe* ssi $\forall p, q \in C, [pq] \subset C$. Le carré, le losange, le rectangle, le triangle, le disque sont des exemples d'ensembles convexes. Ces ensembles ont des propriétés remarquables et beaucoup d'algorithmes sont facilités lorsque l'on sait que la donnée est convexe en entrée. Par exemple, l'algorithme GJK – algorithme très populaire de détection dynamique de collisions utilisé dans les jeux 3D — détermine les collisions entre ensembles convexes de manière très rapide.

Parfois, on approche une forme géométrique complexe par un ensemble convexe. Ainsi, c'est ce que l'on fait lorsqu'on utilise des boîtes englobantes. Par exemple, avant d'aller tester si réellement deux formes complexes s'intersectent, on teste d'abord si leurs deux boîtes englobantes s'intersectent.

Un ensemble convexe englobant une forme S mais plus précis que la boîte englobante est l'*enveloppe convexe de* S, notée $\mathrm{Conv}(S)$. Il est défini comme l'intersection de tous les ensembles convexes qui contiennent S. On peut aussi le définir comme l'intersection de tous les demi-plans qui contiennent S. Au sens où il est contenu dans tous les autres convexes contenant S, c'est le plus petit convexe qui contient S.

Comme beaucoup d'algorithmes utilisent la convexité, il est important de savoir calculer efficacement l'enveloppe convexe d'un ensemble. Nous regardons ici des algorithmes qui calculent l'enveloppe convexe d'un ensemble de points.

L'enveloppe convexe d'un ensemble de points $\{p_0, \ldots, p_{n-1}\}$ a certaines propriétés. Il s'agit d'un polygone simple dont les sommets doivent être trouvés parmi les p_i. On peut le voir (en 2D) comme un élastique que l'on lâche autour des points et qui se reserre au plus court.

Chacun de ses côtés vérifie la propriété que tous les autres points sont sur la gauche de ce côté. Utilisez cette propriété pour trouver un algorithme qui affiche la liste des côtés de l'enveloppe convexe sous forme de paires de points (pas forcément dans l'ordre). L'algorithme naïf ainsi obtenu est en $\Theta(n^3)$.

Il y a beaucoup d'algorithmes pour calculer l'enveloppe convexe de points dans le plan : méthodes incrémentielles en triant les points de gauche à droite, la méthode diviser pour régner, la méthode par greffes successives. Nous en présentons deux : le balayage de Graham et le parcours de Jarvis.

Le balayage de Graham (Algorithme 4) construit un ensemble étoilé, puis, en partant d'un sommet qu'on sait être sur l'enveloppe convexe, ne conserve que les points sur les branches de l'étoile. La preuve de son exactitude se fait en montrant qu'à chaque début d'itération, les points sur la pile forment l'enveloppe convexe des points $p_1, \ldots, p_{i-1}$, stockés dans l'ordre contraire. La complexité de cet algorithme est dominée par la complexité du tri, et donc en $\Theta(n \log n)$.

Le parcours de Jarvis construit l'enveloppe convexe en deux parties, la partie droite puis la partie gauche. L'idée est de partir du sommet le plus bas, puis de chercher le sommet au-dessus faisant le plus petit angle polaire. On continue jusqu'à arriver tout en haut. On fait pareil de l'autre côté. La complexité est en $O(nh)$ où h est le nombre de sommets de l'enveloppe convexe. On parle d'algorithme *output-sensitive* ou *de complexité dépendante de la sortie*. Dans le cas où $h = o(\log n)$, il vaut mieux utiliser Jarvis que Graham.

Algorithme 4 : Balayage de Graham pour calculer l'enveloppe convexe.

```
Action BALAYAGE-GRAHAM( E P : tableau[1,n] de Point, S S : pile de Point) ;
Var : i : entier ;
début
    Cherchez P[i] le point de P d'ordonnée minimale, et si égalité, d'abscisse minimale ;
    ECHANGER( P[i], P[1] );
    TRIER( P, 2, n ) ;        // tri selon l'angle polaire mesuré par rapport à P[1]
    CRÉERPILE(S);
    EMPILER(S, P[1]);
    EMPILER(S, P[2]);
    EMPILER(S, P[3]);
    pour i de 4 à n faire
        tant que ORIENTATION(SOUS-SOMMET(S), SOMMET(S), P[i]) ≤ 0 faire
            DÉPILER(S);
        EMPILER(S, P[i]);
```

Algorithme 5 : Algorithme de Jarvis pour calculer l'enveloppe convexe (côté droit).

```
Action JARVIS-DROIT( E P : tableau[1,n] de Point, S S : pile de Point) ;
début
    Cherchez P[i] le point de P d'ordonnée minimale, et si égalité, d'abscisse minimale ;
    ECHANGER( P[i], P[1] );
    EMPILER( P[1] );
    trouve ← vrai;
    tant que trouve faire
        j ← 2;
        pour i de 3 à n faire
            si P[i] ≠ SOMMET(S) et ORIENTATION(SOMMET(S), P[j], P[i]) ≤ 0 et
              AU-DESSUS(P[i], SOMMET(S)) alors
                j ← i;
        si AU-DESSUS(P[j], SOMMET(S)) alors EMPILER(S, P[j]) ;;
        else trouve ← faux;
        ;
// Retourne ''vrai'' si et seulement p est au-dessus de q.
Fonction AU-DESSUS( E p, q : Point ) : booléen ;
début
    Retourner p.y > q.y ou (p.y = q.y et p.x > q.x );
```

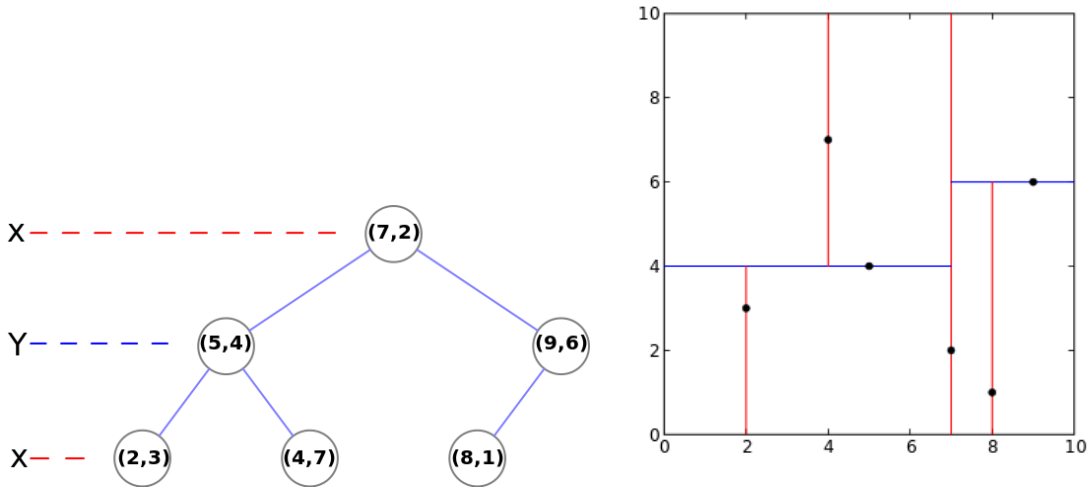


FIGURE 2 – Décomposition en arbre kD des points (2,3), (5,4), (9,6), (4,7), (8,1), (7,2). A gauche, la vision arbre binaire. A droite, le découpage du plan correspondant.

A noter, le meilleur algorithme de calcul d'enveloppe convexe de points est en $\Theta(n \log h)$. Si la séquence de points $P[i]$ forme une ligne polygonale sans auto-intersection, alors son enveloppe convexe se calcule en temps $\Theta(n)$ par l'algorithme de Melkman.

Exercices

1. Est-il nécessaire de calculer les angles polaires pour faire le tri dans le balayage de Graham ?
2. Même question pour Jarvis.

4.7 Structures pour découper le plan ou l'espace

Ces structures permettent de représenter des données géométriques du plan ou de l'espace de façon hiérarchisée, afin d'accélérer les requêtes géométriques. Elles permettent par exemple de savoir quels sont les objets géométriques les plus proches d'un point de l'espace.

Les structures stockant des points les plus simples (et sans doute aussi les plus utilisées) sont :

— les arbres 2^k-aires (arbres binaires de recherche en 1D, arbres quaternaires ou *quadtrees* en 2D, *octrees* en 3D) qui sont des extensions des arbres binaires de recherche en toute dimension. Ici chaque point situé à un nœud de l'arbre coupe l'espace en autant de régions qu'il y a d'orthants.

— les arbres kD qui sont des arbres binaires. Chaque point situé à un nœud de l'arbre coupe l'espace en deux régions selon un plan qui dépend de la profondeur dans l'arbre. Ainsi les nœuds de profondeur 0 coupe l'espace selon leur première coordonnée x, les nœuds de profondeur 1 coupe l'espace selon leur deuxième coordonnée y, les nœuds de profondeur 2 coupe l'espace selon leur troisième coordonnée z, etc, en recommançant à la première coordonnée après avoir coupé selon la dernière.

Les seconds ont l'avantage d'être plus facilement équilibrable que les premiers. Il suffit en effet à chaque fois de choisir comme point de découpe le point situé à la médiane des points restants selon la direction de découpe. Une illustration est donnée sur la Figure 2.

L'algorithme de construction d'un kD-tree bien équilibré est donné par Algorithme 6. En fait, un tel arbre défini un ordre total entre les points (mais qui n'est pas calculable en temps constant).

Ce type de structure accélère en général les requêtes de proximité géométrique en changeant un facteur linéaire n par un facteur logarithmique h, où h est la hauteur de l'arbre ($h = O(\log n)$ si on utilise la consruction de l'Algorithme 6). On peut citer par exemple :

Algorithme 6 : Créer un arbre kD à partir des points d'un tableau de points T entre les indices i et j. L'entier a est l'axe initial utilisé entre 0 et $k-1$, où k est une constante égale à la dimension de l'espace. Attention, le tableau T voit l'ordre de ses points modifiés dans la construction.

```
Fonction CRÉER-ARBRE-KD( ES T : Tableau[0..MAX] de Point, E i, j, a : entier ) : Arbre ;
Var : A : Arbre ; p : Point ; m : entier ;
début
    si i > j alors
        Retourner Null ;
    si i = j alors
        CRÉER-ARBRE(A, T[i]);
        Retourner A ;
    p ← MÉDIANE-SELON-AXE(T, i, j, a) ;
    PARTITION(T, i, j, p);
    m ← ⌊(i+j)/2⌋ ;
    CRÉER-ARBRE(A, T[m]);
    MODIFIER-GAUCHE(A, RACINE(A), CRÉER-ARBRE-KD(T, i, m − 1, (a + 1)%k)) ;
    MODIFIER-DROITE(A, RACINE(A), CRÉER-ARBRE-KD(T, m + 1, j, (a + 1)%k)) ;
    Retourner A;
```

— Déterminer si un point p appartient à la structure prend un temps $O(h)$.
— Déterminer s'il existe un point dans la structure à distance ϵ d'un point p prend un temps $O(h)$, si ϵ est de l'ordre de la plus petite distance entre deux points (voir Algorithme 7).
— Déterminer la liste des m points dans une zone (pour une zone de géométrie simple) prend un temps de l'ordre de $O(mh)$.
— Déterminer les deux points les plus proches prend un temps $O(nh)$.

Algorithme 7 : Sort dans la file F les points de A qui sont dans la boule de centre p et de rayon r. L'entier a désigne l'axe courant dans le kD-tree.

```
Action POINTS-DANS-BOULE( ES F : File de Point, E A : kD-tree, E N : Noeud, E p :
  Point, E r : réel, E a : entier );
Var : q : Point;
début
    si N ≠ Null alors
        q ← VALEUR(A, N);
        si DISTANCE(p, q) ≤ r alors ENFILER(F, q);
        si p[a] ≤ q[a] + r alors
            POINTS-DANS-BOULE(F, A, GAUCHE(A, N), p, r, (a + 1)%k);
        si p[a] ≥ q[a] − r alors
            POINTS-DANS-BOULE(F, A, DROITE(A, N), p, r, (a + 1)%k);
```

Exercices

1. Comme un kD-tree est un arbre binaire ordonné A, on peut définir comme pour les arbres binaires de recherche un premier élément et un suivant à chaque élément. On définit donc les mêmes fonctions :
   — **Fonction** PREMIER(**E** A : kD-tree) : Noeud qui retourne le Noeud de l'arbre dont la valeur est la plus faible.
   — **Fonction** SUIVANT(**E** A : kD-tree, **E** N : Noeud) : Noeud qui retourne le Noeud de l'arbre dont la valeur est juste supérieure à la valeur du noeud N ou Null sinon.

(a) Soit le programme suivant. Que fait-il ? Quelle est la complexité en pire cas de l'opération *Suivant* ? En déduire une borne supérieure sur la complexité en pire cas du programme *Parcours*.

Action Parcours(**E** A : kD) ;
Var : N : Noeud ;
début
  $N \leftarrow$ *Premier*(A);
  Tant Que $N \neq$ *Null* **Faire**
    Affiche(*Valeur*(A, N)) ;
    $N \leftarrow$ *Suivant*(A, N);

(b) Quelle est maintenant le coût amorti de chaque appel à *Suivant* ? Utilisez la méthode des agrégats. Est-il dépendant de la hauteur h du kD-tree ?

2. Comment tester la proximité avec des segments ? Même si cette façon n'est pas optimale, on peut aussi utiliser des kD-tree. L'idée est de placer sur chaque segment suffisament de points. Ensuite, on ne fait qu'un test approximatif pour savoir si l'on est à côté d'un des points du segment. La petite difficulté est de ne pas mettre trop de points. En général, si on veut être sûr de détecter que l'on est à distance ϵ d'un segment, il s'agit de bien choisir le rayon *delta* des boules placés sur le segment de manière à : (1) ne pas avoir trop de points, (2) garantir que si un des points est à distance inférieur à δ alors la distance au segment est inférieure à ϵ.

   Expliquez pourquoi un écart entre les points inférieur à 2ϵ et $\delta = \sqrt{2}\epsilon$ est un bon choix.

   On note que l'aire de l'ensemble des points "distance à 1 segment de longueur l inférieure à ϵ" est de $\pi\epsilon^2 + 2l\epsilon$. Or l'aire de l'ensemble des points "à distance $= \sqrt{2}\epsilon$ des points répartis comme indiqué sur le segment de longueur l" est de $3\pi\epsilon^2 + l(1 + \frac{\pi}{2})\epsilon$. Le ratio d'aire est de $1,28$ pour des longs segments. On va donc surtout tester les bons segments.
