Notes de programmation (C) et d'algorithmique

Roberto M. Amadio

Notes de programmation (C) et d'algorithmique

Roberto M. Amadio
Université Paris-Diderot

17 décembre 2018

Table des matières

Préface

On trouve de nombreux livres, notes de cours, vidéos,... qui proposent une introduction adéquate à la programmation et à l'algorithmique. Dans ce sens ces notes de cours sont redondantes ; elles n'ont d'autre ambition que de fournir une *trace synthétique* des sujets traités dans un cours d'introduction à la programmation (en C) et à l'algorithmique qui s'adresse aux étudiants d'un Master en mathématiques de l'Université Paris Diderot. Les premiers chapitres de ces notes ont été utilisés aussi dans le cadre d'un cours d'initiation à la programmation pour une classe préparatoire aux écoles d'ingénieurs.

Les chapitres 1–13 de ces notes prennent la forme d'une visite guidée des structures principales de la programmation (en C) et d'un certain nombre d'exemples d'algorithmes classiques qui illustrent leur utilisation. On s'attend à que le lecteur teste, modifie et améliore les programmes discutés en cours. Ces chapitres ne sont ni un manuel de référence ni une introduction systématique au langage C (voir, par exemple, le manuel rédigé par les concepteurs du langage C [KR14]). Dans le commentaire aux exemples, on esquissera quelques principes de génie logiciel qu'il convient de suivre dans la conception de programmes (de taille modeste). Le lecteur intéressé pourra consulter, par exemple, [KP17] pour une discussion plus étendue.

Les chapitres 14–27 se focalisent sur des notions d'algorithmique plus avancées et s'articulent autour de 3 thématiques.

— La présentation d'un certain nombre de *structures de données* : tas, arbres, listes à enjambements, tables de hachage, graphes,...
— L'introduction de *techniques de conception* d'algorithmes : diviser pour régner, programmation glutonne et dynamique, approche probabiliste, programmation linéaire.
— La description de *techniques d'analyse* : la notion de complexité asymptotique dans le pire cas et en moyenne et la solution de relations de récurrence.

Ces chapitres s'appuient sur un certain nombre de notions mathématiques qui sont normalement couvertes dans un premier cycle scientifique. On suppose notamment des notions élémentaires de théorie des groupes, d'algèbre linéaire, d'arithmétique modulaire et de calcul des probabilités. Pour les aspects algorithmiques, la lecture du livre [CLRS09] est fortement conseillée et le livre [Sho05] permet d'approfondir les notions mathématiques évoquées.

Notation

Ensembles

$\emptyset$	ensemble vide
$\mathbf{2} = \{0, 1\}$	valeurs booléennes
$\mathbf{N}$	nombres naturels
$\mathbf{Z}$	nombres entiers
$\mathbf{Q}$	nombres rationnels
$\mathbf{R}$	nombres réels
$\cup, \cap$	union, intersection de deux ensembles
$\bigcup, \bigcap$	union, intersection d'une famille d'ensembles
X^c	complémentaire de X
Y^X	fonctions de X dans Y
$\mathcal{P}(X)$	sous-ensembles de X
$\mathcal{P}_{fin}(X)$	sous-ensembles finis de X
$\sharp X$	cardinal de X
R^*	clôture réflexive et transitive d'une relation R

Arithmétique

$\mathbf{Z}_n$	entiers modulo n
$\mathbf{Z}_n^*$	groupe multiplicatif des entiers modulo n
$\equiv$	congruence
a/b	quotient division entière
$a \mod b$	reste de la division entière (ou module)

Algèbre linéaire

$A, B, \ldots$	matrices
$adj(A)$	matrice adjointe
$det(A)$	déterminant
A^{-1}	matrice inverse

Probabilité

Ω	Ensemble des expériences
$\mathcal{A}$	Ensemble des événements
$P(A)$	probabilité d'un événement
$P(A \mid B)$	probabilité conditionnelle
X, Y	variables aléatoires discrètes (v.a.d)

Algorithmique

f est $O(g)$	$\exists n_0, k \geq 0 \ \forall n \geq n_0 \ (f(n) \leq k \cdot g(n))$
f polynomiale	$\exists d \geq 0 \ f$ est $O(n^d)$

Chapitre 1

Introduction

On introduit les notions d'algorithme et de programme et on discute la structure et l'interprétation d'un programme C. Il s'agit de deux sujets fondamentaux pour la suite du cours.

1.1 Algorithmes et programmes

L'informatique (en tant que science) s'intéresse au traitement *automatique* de l'*information*.

En général, une *information* est codifiée par une suite finie de symboles qui varient sur un certain alphabet et, modulo codage de cet alphabet, on peut voir cette suite comme une suite de valeurs binaires (typiquement 0 ou 1). Par exemple, une information pourrait être la suite 'bab' qui est une suite sur l'alphabet français. Il existe un code standard, appelé code ASCII, qui code les symboles du clavier avec des suites de 8 chiffres binaires. En particulier, le code ASCII de 'a' est '01100001' et le code ASCII de 'b' est '01100010'.

L'aspect *automatique* de l'informatique est lié au fait qu'on s'attend à que les *fonctions* qu'on définit sur un ensemble de données (les informations) soient *effectivement calculables* et même qu'elles puissent être mises-en-oeuvre dans les dispositifs électroniques qu'on appelle ordinateurs.

L'ensemble des suites finies de symboles binaires 0 et 1 est dénombrable (il est infini et en correspondance bijective avec l'ensemble des nombres naturels). Plus en général, l'ensemble des suites finies de symboles d'un alphabet fini (ou même dénombrable) est dénombrable.

Considérons maintenant l'ensemble des fonctions partielles de type $f : D \rightharpoonup D'$ où D et D' sont des ensembles dénombrables. Un *algorithme* est une telle fonction pour laquelle *en plus* on peut préciser une *méthode de calcul*.

Exemple 1 *On dénote par* $\{0,1\}^*$ *l'ensemble des suites finies de* 0 *ou* 1 *(y compris la suite vide). Prenons* $D = D' = \{0,1\}^*$ *et associons à toute suite* $w = b_0 \cdots b_n \in D$ *un nombre naturel* $\langle w \rangle$ *défini par :*

$$\langle w \rangle = \Sigma_{k=0,\ldots,n} b_i \cdot 2^i \ .$$

Par exemple :

$$\langle 01010 \rangle = 0 \cdot 2^0 + 1 \cdot 2^1 + 0 \cdot 2^2 + 1 \cdot 2^3 + 0 \cdot 2^4 = 2 + 8 = 10 \ .$$

La suite w *représente donc un nombre naturel en base* 2 *avec le chiffre le plus significatif à droite (SIC). Tout nombre naturel peut être représenté de cette façon mais la représentation*

n'est pas *unique. Par exemple :*

$$\langle 01\rangle = \langle 010\rangle = \langle 0100\rangle = \cdots = \langle 010\cdots 0\rangle = 2 \ .$$

Cependant, on peut obtenir l'unicité en se limitant aux suites de 0 *et* 1 *qui ne terminent pas par* 0*. Si* n *est un nombre naturel, on dénote par* $\lfloor n \rfloor$ *la seule suite* $w \in \{0,1\}^*$ *telle que : (i)* $\langle w\rangle = n$ *et (ii)* w *ne termine pas par* 0*.* [1] *On peut maintenant définir une fonction :*

$$f : \{0,1\}^* \to \{0,1\}^* \ ,$$

telle que $f(w) = w'$ *ssi* $w' = \lfloor (\langle w\rangle)^2 \rfloor$*. Par exemple, si* $w = 010$ *on a* $(\langle w\rangle)^2 = 2^2 = 4$ *et* $w' = 001$*. Modulo codage, on a défini la fonction carré sur les nombres naturels. Pour avoir un* algorithme*, il faut encore préciser une méthode de calcul. Par exemple, une façon de procéder pourrait être de prendre la suite* w *en entrée la voir comme un nombre binaire en base* 2 *et le multiplier par lui même en adaptant à la base* 2 *l'algorithme pour la multiplication appris en primaire. Une autre façon de procéder (et donc un autre algorithme), serait de convertir la suite* w *dans un nombre en base* 10*, de multiplier ce nombre par lui même et enfin de retrouver sa représentation binaire (on verra en détail comment effectuer ces conversions dans la section 4.6). On voit donc dans cet exemple qu'on peut associer plusieurs algorithmes à la même fonction.* [2]

Dans notre exemple, on a utilisé l'intuition des calculs appris en primaire pour spécifier l'algorithme (la méthode de calcul). Le lecteur sait que l'on peut effectuer les opérations arithmétiques sur des nombres de taille arbitraire à condition de disposer de suffisement de papier, de crayons et de temps. Plus en général, on peut imaginer des 'machines' qui savent manipuler des chiffres, stocker des informations et les récupérer. Un *programme* est alors un algorithme qui est formalisé de façon à pouvoir être exécuté par une telle 'machine'. [3]

Exemple 2 *Considérons le problème de calculer le* produit scalaire *de deux vecteurs de taille* n*. Une première description de l'algorithme pourrait être la suivante :*

Entrée $x, y \in \mathbf{R}^n$.

Calcul $s = 0$*. Pour* $i = 1, \ldots, n$ *on calcule* $s = s + x_i y_i$.

Sortie s.

Pour aller vers un programme*, il faut préciser une représentation des nombres entiers et des nombres réels. Les langages de programmation disposent de types prédéfinis. En particulier, en* C *on peut utiliser le type* int *pour représenter des entiers sur* 32 *bits et le type* double *pour représenter les réels sur* 64 *bits. Dans les deux cas, on ne peut représenter qu'un sous-ensemble fini des nombres entiers et réels. Il faut donc savoir que les opérations arithmétiques dans le contexte de la programmation peuvent provoquer des* débordements*, et dans les cas des réels des* approximations *aussi. Par ailleurs, dans les langages de programmation on peut représenter les vecteurs par des* tableaux *(qu'on étudiera dans le chapitre 7). Ainsi un programme* C *qui raffine l'algorithme ci-dessus pourrait être le suivant.*

1. Notez qu'avec cette convention $\lfloor 0 \rfloor$ est la suite vide.

2. En général, on peut montrer que pour tout algorithme il y a un nombre dénombrable d'algorithmes qui sont équivalents dans le sens qu'ils calculent la même fonction.

3. Un exemple particulièrement simple d'une telle machine est la *machine de Turing* qui a été formalisée autour de 1930 par Alan Turing.

```
double produit_scalaire(double x[], double y[], int n){
  double s=0;
  int i;
  for(i=0;i<n;i++){s=s+x[i]*y[i];}
  return s;}
```

En résumant, un *algorithme* est une *fonction* partielle avec domaine et codomaine dénombrable et avec une méthode de calcul qui précise pour chaque entrée comment obtenir une sortie. A ce stade, la méthode de calcul est typiquement décrite dans le langage semi-formel des mathématiques. Un *programme* est un algorithme qui est codifié dans le *langage de programmation* d'une machine. C'est une bonne pratique de passer de la fonction à l'algorithme et ensuite de l'algorithme au programme. Avec une *fonction* on spécifie le problème, avec un *algorithme* on développe une méthode de calcul (pour la fonction) en négligeant un certain nombre de détails et enfin avec le *programme* on peut vraiment exécuter la méthode de calcul sur une machine.

Remarque 1 *Les notions d'algorithme, de modèle de calcul et de programme ont été développées autour de 1930 dans un cadre mathématique fortement inspiré par la logique mathématique qu'on appelle* théorie de la calculabilité. *Deux conclusions fondamentales de cette théorie sont :*

1. *Les modèles de calcul et les langages de programmation associés (du moins ceux considérés en pratique) sont équivalents dans les sens qu'ils définissent la même classe d'algorithmes (c'est la thèse de Church-Turing). Par exemple, pour tout algorithme codifié dans un programme* Java *on a un algorithme équivalent codifié dans un programme* C *(et réciproquement).*
2. *Il n'y a qu'un nombre* dénombrable *de programmes et donc une très grande majorité des fonctions qu'on peut définir sur des ensembles dénombrables n'ont pas de méthode de calcul associée. Par exemple, il n'y pas de programme qui prend une assertion dans le langage de l'arithmétique et qui décide si cette assertion est vraie ou fausse. Et il est aussi impossible d'écrire un programme qui prend en entrée un programme* C *et décide si le programme termine ou pas.*

Avec le développement des ordinateurs, on a cherché à cerner l'ensemble des problèmes qui peuvent être résolus de façon efficace. *Comme on le verra dans la suite du cours (chapitre 6), la complexité d'un problème est une fonction de la taille des données qui décrivent son entrée. Par exemple, si on considère le problème de la multiplication de deux nombres naturels, on peut montrer que l'algorithme du primaire permet de multiplier deux nombres de n chiffres avec un nombre d'opérations élémentaires qui est de l'ordre de n^2. On dit que la complexité de l'algorithme est* quadratique. *Plus en général, un algorithme polynomial est une méthode de calcul tel qu'il existe un polynôme $p(n)$ avec la propriété que la méthode sur une entrée de taille n effectue un nombre d'opérations élémentaires borné par $p(n)$. On appelle* théorie de la complexité *la branche de l'informatique théorique qui cherche à classifier la complexité des problèmes. Dans ce contexte, dans les années 1970 on a formulé le problème ouvert qui est probablement le plus important et certainement le plus célèbre de l'informatique. D'une certaine façon, la question est de savoir si* trouver *une solution d'un problème est beaucoup plus difficile que de* vérifier *sa correction. L'intuition suggère une réponse positive mais dans un certain cadre on est incapable de prouver le bien fondé de cette intuition. Le cadre est le suivant : existe-t-il un algorithme qui prend en entrée une formule A du calcul propositionnel et qui décide dans un temps polynomial dans la taille de A si A est satisfiable ? Par exemple,*

si $A = (\overline{x} \vee y) \wedge (x \vee \overline{y})$ *alors on peut satisfaire la formule avec l'affectation* $v(x) = 0$ *et* $v(y) = 0$. *Par contre, le lecteur peut vérifier que la formule* $B = (\overline{x} \vee y) \wedge (x \vee \overline{y}) \wedge (\overline{x} \vee \overline{y}) \wedge (\overline{x} \vee y)$ *n'est pas satisfiable. Pour toute affectation* v, *il est facile de* vérifier *si une formule* A *est vraie par rapport à l'affectation. Par ailleurs, pour savoir si une formule est satisfiable on peut générer toutes les affectations et vérifier s'il y en a une qui satisfait la formule. Malheureusement, cette méthode n'est pas efficace car pour une formule avec* n *variables il faut considérer* 2^n *affectations (la fonction exponentielle* 2^n *croit beaucoup plus vite que n'importe quel polynôme). La question ouverte est donc de trouver un algorithme polynomial qui nous permet de décider si une formule est satisfiable ou de montrer qu'un tel algorithme n'existe pas.* [4]

1.2 Structure et interprétation d'un programme C

Le langage C a été conçu autour de 1970 dans le but d'écrire un système d'exploitation (qui deviendra le système Unix) en utilisant C plutôt qu'un langage assembleur, ce qui est bénéfique pour la portabilité du système. Il s'agit d'un *langage impératif* dans le style des langages ALGOL et PASCAL. Le calcul est donc organisé autour de l'exécution de *commandes* qui modifient la *mémoire*. Il se distingue de ses prédécesseurs par la possibilité d'effectuer des *opérations de bas niveau sur la mémoire* (arithmétique de pointeurs) ; ce qui est une source potentielle d'*efficacité* et d'*erreurs*. Par rapport à ses successeurs (C++, JAVA,...), on notera l'absence d'un mécanisme pour combiner types de données et opérations et d'un système automatique de récupération de mémoire (on dit aussi ramasse miettes ou *garbage collector*, en anglais).

En général, dans un langage la *syntaxe* est un ensemble de règles qui permettent de produire des phrases admissibles du langage et la *sémantique* est une façon d'attacher une signification aux phrases admissibles du langage.

Dans le cas des langages de programmation, on a besoin de règles pour écrire des programmes qui seront acceptés par la machine et aussi d'une méthode pour déterminer la sémantique à savoir la fonction calculée par le programme. On aura l'occasion de revenir sur les détails de la syntaxe dans la suite du cours. Pour l'instant, on souhaite esquisser une méthode pour calculer le comportement d'un programme (sa sémantique).

En première approximation, la sémantique d'un programme C (et plus en général d'un langage impératif) s'articule autour de 6 concepts : mémoire, environnement, variable, fonction, bloc d'activation (*frame* en anglais) et contrôle.

Mémoire Une fonction (partielle) qui associe des *valeurs* aux *adresses de mémoire*. Il est possible de :
- *lire* une adresse de mémoire,
- *modifier* son contenu,
- *allouer* une valeur à une nouvelle adresse,
- *récupérer* une adresse de mémoire pour la réutiliser.

Environnement Dans un langage de programmation de 'haut niveau' on donne des *noms symboliques* aux entités qu'on manipule (une constante, une variable, une fonction,...)

4. On dit aussi que le problème est de savoir si la classe NP est identique à la classe P des problèmes qui admettent un algorithme polynomial. Intuitivement, la classe NP est la classe des problèmes dont la solution peut être vérifiée en temps polynomial. A priori NP contient P et le problème est de savoir si l'inclusion est stricte.

Un *environnement* associe à chaque nom du programme une entité (une valeur, une adresse mémoire, un segment de code,...) Environnement et mémoire sont liés. Par exemple, dans :

$$\mathsf{int\ x = 10;}$$

on associe au nom x une nouvelle adresse de mémoire l (modification de l'environnement) et à l'adresse de mémoire l la valeur 10 (modification de la mémoire). Pour décrire ces associations, on écrit :

$$\mathsf{x{-}{>}l,\ l{-}{>}10}\ .$$

Variable Un *nom* qui est associé à une *adresse de mémoire* (on dit aussi location ou référence) qui contient éventuellement une *valeur*. Dans un langage *impératif* comme C la valeur peut être modifiée plusieurs fois pendant l'exécution. Il ne faut pas confondre les variables au sens mathématique avec les variables au sens informatique.

Fonction Un *segment de code* qu'on peut exécuter simplement en invoquant son nom. Souvent une fonction prend des *arguments* et rend un *résultat*. Dans un langage *impératif* comme C, le résultat rendu dépend à la fois des arguments et du contenu de la mémoire. Comme pour les variables, il convient de ne pas confondre les fonctions mathématiques avec les fonctions informatiques.

Bloc d'activation Un *vecteur* qui contient :
- un nom de fonction,
- ses paramètres (arguments, variables locales),
- le compteur ordinal (adresse de la prochaine instruction de la fonction à exécuter).

Contrôle Une pile de blocs d'activation. L'ordre correspond à l'ordre d'appel. Le bloc le plus profond dans la pile est le plus ancien.

Exemple 3 *On illustre l'utilisation des* 6 *concepts dans l'exemple suivant d'un programme* C *qui calcule le plus grand commun diviseur (pgcd) d'après l'algorithme d'Euclide. On rappelle que si* a, b *sont des entiers avec* $b > 0$ *alors ils existent uniques* q *et* r *tels que* $0 \leq r < b$ *et*

$$a = b \cdot q + r\ .$$

On appelle q *le quotient ou la division entière de* a *par* b *et* r *le reste qu'on dénote aussi par* $a \mod b$. *En supposant* a, b *entiers avec* $b > 0$ *on a la propriété suivante :*

$$pgcd(a,b) = \begin{cases} b & \text{si } a \mod b = 0 \\ pgcd(b, a \mod b) & \text{autrement.} \end{cases}$$

En C, *l'opération de quotient est dénotée par* / *et celle de reste par* %. *Voici un programme pour le* pgcd.

```
#include <stdio.h>
void lire(int *p){
  printf("Entrez un entier positif:");
  scanf("%d",p);}
int pgcd(int a,int b){
  int mod=a%b;
  if (mod==0){return b;}
  else {return pgcd(b,mod);}}
```

```
PILE FRAMES                    MEMOIRE
main()
a->l1, b->l2
   lire(l1)
   p->l3                       l3->l1, l1->6
   fin_lire
   lire(l2)
   p->l4                       l4->l1, l2->4
   fin_lire
resultat->l5
   pgcd(6,4)
   a->l6, b->l7                l6->6, l7->4
   mod->l8,                    l8->2
       pgcd(4,2)
       a->l9, b->l10           l9->4, l10->2
       mod->l11                l11->0
       fin_pgcd 2
   fin_pgcd 1
                               l5->2
fin_main
```

TABLE 1.1 – Trace de l'exécution du programme avec entrées 6 et 4

```
void main() {
  int a, b;
  lire(&a);
  lire(&b);
  int resultat;
  resultat = pgcd(a,b);
  printf("le pgcd est %d\n",resultat);}
```

Ce programme commence avec une directive au compilateur pour inclure les fonctions de bibliothèque contenues dans stdio.h. *Parmi ces fonctions, on trouve les fonctions* printf *et* scanf *qu'on utilisera dans le cours pour imprimer et lire des valeurs.*

Le programme comporte 3 *fonctions :* lire, pgcd *et* main. *L'*interface *(ou* en tête*) de chaque fonction précise le type du résultat et les noms et types des arguments de la fonction. Par exemple, la fonction* pgcd *rend un résultat de type* int *et attend deux arguments de type* int *dont les noms sont* a *et* b. *La table 1.1 décrit l'exécution du programme en supposant que l'utilisateur rentre les valeurs* 6 *et* 4. *Chaque instant du calcul est décrit par la pile des blocs d'activation et le contenu de la mémoire.*

Remarque 2 *Variables et fonctions sont des entités qu'on peut associer à certaines portions du texte (la* syntaxe*) du programme et qui ne changent pas pendant l'exécution. Par opposition, mémoire, environnement, bloc d'activation et contrôle sont des entités qui nous permettent d'expliquer et prévoir l'exécution du programme (la* sémantique*) et qui changent pendant l'exécution.* [5]

Pendant l'exécution peuvent coexister plusieurs instances du même objet syntaxique. *Par exemple, on peut avoir plusieurs instances de la fonction* pgcd *et des variables* a,b,mod. *En particulier, dans le premier appel de* pgcd, a *est associée à l'adresse* l4 *et dans le deuxième à l'adresse* l7.

5. Il faut *raffiner* ce modèle pour arriver à couvrir *tout* C.

*Aussi, on peut avoir des situations d'*homonymie. *Par exemple,* `a` *est une variable de* `main` *et un paramètre (un argument) de* `pgcd`. *On élimine toute ambiguïté en supposant qu'on s'adresse toujours au* `a` *qui est le plus 'proche'.*

Exercice 1 *Le programme suivant est constitué de* 5 *fonctions, dont les fonctions* `f` *et* `g` *qui s'appellent mutuellement. Comme le compilateur* C *insiste pour que tout appel de fonction soit précédé par sa déclaration, on introduit dans* (1) *le* prototype *de la fonction* `g` *et on précise dans* (2) *le corps de la fonction. Que fait ce programme ?*

```
void lire(int *p){
  printf("Entrer nombre >=1\n");
  scanf("%d",p);}
void imprimer(int x){
  printf("La sortie est %d\n",x);}
int g(int);                                    \\(1)
int f(int x){
  if (x>1){return g(2*x);} else {return 0;}}
int g(int x){                                  \\(2)
  if (x>1){return f(x/3);} else {return 1;}}
void main (){
   int x; lire(&x);
   int r=f(x); imprimer(r);}
```

Chapitre 2

Préliminaires

On introduit des notions de nature assez technique qui permettent de commencer à programmer : la compilation et l'exécution d'un programme, la gestion des entrées-sorties et les types primitifs.

2.1 Compilation et exécution

Un programme est d'abord *compilé* (= traduit dans le langage de la machine) et ensuite *exécuté* (par le processeur de la machine).

Exemple 4 *Considérons un programme* C *qui imprime à l'écran le mot* Bonjour. *A partir de maintenant, on omet les directives nécessaire à l'utilisation des fonctions de bibliothèque. Le lecteur peut trouver ces directives dans tout manuel de programmation.*

```
int main(){             //(1)
  printf("Bonjour\n");  //(2)
  return 0;}            //(3)
```

Tout programme C *contient au moins une fonction dont le nom est* main *(ligne* (1)*). Le calcul commence avec un appel à cette fonction et termine quand cette fonction termine son exécution. Par défaut, la fonction* main *ne prend pas d'arguments et rend un entier* 0 *comme résultat (ligne* (3)*). Par convention, l'entier* 0 *indique une terminaison normale. Certains compilateurs permettent aussi un résultat de type* void *(le type vide) et dans ce cas on marque la fin de la fonction avec une commande* return. *La commande qui imprime* Bonjour *est à la ligne* (2). *La fonction* printf *est une fonction de la bibliothèque* stdio *et pour l'utiliser il faut ajouter au programme ci-dessus une directive* ♯include⟨stdio.h⟩. *Le texte à imprimer est compris entre guillemets. Dans l'exemple, après le mot* Bonjour *on imprime aussi un retour de ligne qui est dénoté par le caractère* \n. *On notera que chaque commande dans le corps de la fonction est suivie par un point virgule.*

Tout ce qui suit le symbole // et se trouve dans la même ligne est un commentaire. Un commentaire aide à comprendre le comportement d'un programme mais n'affecte en rien son exécution. En particulier, on utilise cette notation pour numéroter les lignes. Si l'on veut écrire un texte de commentaire sur plusieurs lignes on utilisera la notation :

```
/* texte
   de commentaire ici */
```

L'utilisateur commence par écrire à l'aide d'un éditeur de texte (par exemple emacs*) le programme dans un fichier dont le nom termine par* .c*. Par exemple :* Bonjour.c*. Pour compiler avec le compilateur* gcc *on écrira une commande :*

cc Bonjour.c

Par défaut, le code exécutable généré est mémorisé dans le fichier a.out*. Pour l'exécuter, on lance la commande :*

./a.out

On peut modifier le nom du fichier qui contient l'exécutable en utilisant l'option -o*. Par exemple, avec la commande :*

cc −o Bonjour Bonjour.c

on mémorise l'exécutable dans le fichier Bonjour*. Chaque compilateur propose nombreuses options. En* gcc*, avec l'option* -O *on peut générer un code optimisé, avec l'option* -Wall *on sollicite un certain nombre d'avertissements, avec l'option* -lm *on lie les fonctions de bibliothèque à l'exécutable, avec l'option* -save-temps *on visualise les codes intermédiaires et assembleurs produits par le compilateur.*

2.2 Erreurs

En programmation, on est confronté à des erreurs qu'on peut classifier en deux catégories.

- Les erreurs générées au moment de la *compilation* : parenthèse oubliée, variable non déclarée, type du résultat incompatible avec le type de la fonction,...
- Les erreurs observées au moment de l'*exécution* : division par zéro, indice d'un tableau hors des bornes, manque de mémoire,...

En général, le compilateur fournit assez d'indications pour éliminer les erreurs du premier type. Pour les erreurs de deuxième type, il est souvent nécessaire d'analyser le programme en détail et d'en tester le comportement.

Exemple 5 *Considérons le petit programme suivant :*

```
int main(){            //(1)
  printf("%d\n",3/1);  //(2)
  int x=1;             //(3)
  int y=x-x;           //(4)
  printf("%d\n",3/y);  //(5)
  return 0;}           //(6)
```

Si l'on remplace le ; *en ligne* (3) *par* : *on obtient une erreur au moment de la compilation. Autrement, ce programme compile sans problème mais au moment de l'exécution il génère un message d'erreur car on cherche à diviser* 3 *par* 0*. La raison de ce message tardif est que le compilateur* gcc *ne peut pas* prévoir *que la variable* y *prend la valeur* 0 *à la ligne* (5)*.*

2.3 Entrées-sorties

On utilisera en priorité printf pour écrire une valeur à l'écran et scanf pour lire une valeur de l'écran. On verra plus tard (section 11.6) que des variantes de ces commandes permettent aussi de lire/écrire des fichiers. Voici deux exemples d'utilisation des commandes printf et scanf.

```
printf("x=%d",4);      //imprime : x=4

scanf("%d",&x);        //lit un entier et le sauve dans la variable x de type int
```

On remarquera la présence des symboles %d. Dans le cas de la commande printf, il faut interpréter ces symboles comme un entier dont la valeur doit être déterminée en évaluant l'expression suivante qu'on passe en argument à printf. L'expression 4 ayant comme valeur l'entier 4, l'impression de x=%d produit en effet l'impression des caractères x=4. Dans le cas de la commande scanf, on interprète les symboles %d comme un entier rentré par l'utilisateur qui doit être mémorisé à une adresse spécifiée dans l'expression qu'on passe en argument à scanf. Dans l'exemple, il s'agit de l'adresse de la variable x. On notera l'introduction d'un nouveau opérateur de *déréférencement* & qui sert à déterminer l'adresse associée à une variable. Il s'agit d'un cas particulier de *pointeur* dont on examinera l'utilisation dans le chapitre 11.

Exemple 6 *Voici un programme qui lit un entier n et imprime $n+1$.*

```
int main(){
int x;
printf("Entrez un nombre : \n");
scanf("%d",&x);
printf("%d\n",x+1);
return 0; }
```

Les symboles %d servent à lire/écrire des valeurs de type entier. Pour manipuler des caractères on utilise les symboles %c et pour des flottants les symboles %f ou %lf. Par ailleurs, une commande printf (ou scanf) peut contenir plusieurs occurrences de ces symboles et dans ce cas pour chaque occurrence il faut prévoir une expression (ou une adresse de mémoire) d'un type compatible. Par exemple, la commande :

```
printf("\%d : %f", 3, 455.45);
```

va imprimer un entier et un flottant de la façon suivante :

```
3 : 455.45
```

2.4 Types primitifs et opérations

En première approximation, on peut voir un type comme un ensemble de valeurs (les entiers, les flottants,...). En C on associe à chaque variable un type. Cette information est importante car d'une part elle permet de prévoir la quantité de mémoire qu'il faut associer à la variable et d'autre part elle permet de documenter les intentions du programmeur et d'éviter un certain nombre de situations erronées où l'on cherche à combiner des valeurs incompatibles.

Le langage C contient un certain nombre de *types primitifs* (ou prédéfinis). [1]

1. On verra dans la suite du cours que l'utilisateur peut aussi définir des types.

char pour les caractères ASCII (8 bits). Par exemple, 'a', 'b', '@' sont des valeurs de type char.

short, int, long pour les entiers (typiquement 16, 32 et 64 bits).

float, double pour les réels en virgule flottante (typiquement 32 et 64 bits).

Par ailleurs, le type des *booléens* est (souvent) codé avec les entiers. La convention est que 0 correspond à *faux* et un nombre différent de 0 (typiquement 1) à *vrai*.

Pour chaque type on dispose de certaines *opérations*. Par exemple, les symboles / et % dénotent respectivement l'opération de division entière et de reste sur les valeurs de type int. Le symbole / dénote aussi la division (réelle) sur les flottants et le symbole % est utilisé dans les directives de lecture/écriture. Un même symbole peut donc être interprété de façon complètement différente selon le contexte dans lequel il est utilisé.

Exemple 7 *Voici un petit programme qui illustre l'introduction de variables des différents types primitifs, la forme des valeurs de ces types et les directives utilisées pour les imprimer avec la commande* printf.

```
int main() {
char x1='a';       printf("%c\n",x1);
short x2=2754;     printf("%d\n",x2);
int x3=333333;     printf("%d\n",x3);
long x4=333333333; printf("%ld\n",x4);
float x5=0.45f;    printf("%f\n",x5);
double x6=455.54;  printf("%lf\n",x6);
return 0;}
```

2.5 Instabilité numérique

Il convient de rappeler qu'à cause des approximations introduites par les opérations arithmétiques sur le flottants la façon de calculer une fonction peut avoir un impact non négligeable sur le résultat.

Exemple 8 *Considérons deux définitions de la même fonction sur les réels :*

$$f(x) = x \cdot (\sqrt{x+1} - \sqrt{x}) = \frac{x}{\sqrt{x+1} + \sqrt{x}} \ .$$

Dans la première formulation, le numérateur tend à 0 *et provoque des erreurs de calcul significatifs pour* $x \geq 10^{10}$*. Le lecteur peut tester le programme suivant (en compilant avec l'option* -lm*).*

```
int main(){
   float x;
   printf("Input?");
   scanf("%f", &x);
   printf("fonction 1\n");
   float y=x * (sqrt(x+1) - sqrt(x));
   printf("%f\n",y);
   printf("fonction 2\n");
   y=x/(sqrt(x+1) + sqrt(x));
   printf("%f\n",y);
   return 0;}
```

2.6 Conversions implicites et explicites

Dans la pratique mathématique, on a l'habitude de voir un entier comme un réel et un réel comme un complexe. Les langages de programmation supportent ce type de pratique en introduisant des *conversions implicites*. Notamment, en C on effectue automatiquement les conversions suivantes :

$$\mathsf{char} \leq \mathsf{short} \leq \mathsf{int} \leq \mathsf{long} \leq \mathsf{float} \leq \mathsf{double}\ .$$

Parfois, il est nécessaire de procéder dans l'autre sens. Par exemple, on veut voir un int comme un char. Dans ce cas, le programmeur doit effectuer une *conversion explicite*. En C, on parle aussi d'opération de *cast* (ou coercition). Par exemple, on peut écrire :

```
int x=34444;
char y=(char)(x);
```

Le lecteur remarquera qu'il y a beaucoup plus d'entiers de type int (32 bits) que de caractères de type char (8 bits). L'opération de cast a donc un caractère arbitraire et il faut bien comprendre son effet. En général, les opérations de cast entre types produisent souvent des erreurs et il faut les utiliser avec parcimonie.

Exemple 9 *Voici un petit programme qui illustre l'utilisation de conversions implicites et explicites (*cast*).*

```
int main(){
    char x1 ='3';
    printf("%c\n",x1);                                          //on imprime 3
    printf("%d\n",x1);                                          //on imprime le code ASCII de '3'
    short x2 =2700;
    short x4 = x1 + x2;  printf("%d\n",x4);                     //conversion implicite de char-short
    int   x5 = 10000;
    long x6=333333333; x6=x6+x5;   printf("%ld\n",x6);     // conversion implicite int-long
    float x7=0.45f;x7=x7+x6;       printf("%f\n",x7);      // conversion implicite long-float
    double x8=455.54;x8=x8+x7;     printf("%lf\n",x8);     // conversion implicite float-double
    char x9='a';x5=x9+x5;          printf("%d\n",x5);      // conversion implicite char-int
    printf("%d\n",(int)x7);                                 // cast float-int
    x7=((float)x8)/x7;             printf("%f\n",x7);      // cast double-float
    return 0; }
```

Chapitre 3

Contrôle

On peut voir le corps de chaque fonction comme une suite de *commandes*. Dans une grande partie des programmes étudiés jusqu'à maintenant on a une *liste* de commandes qu'on exécute *une fois* dans l'ordre. On va présenter des opérateurs qui permettent d'exécuter les commandes selon un ordre plus élaboré. Par exemple :

- On exécute une commande seulement si une certaine condition logique est satisfaite.
- On répète l'exécution d'une commande tant qu'une certaine condition logique est satisfaite.
- On arrête l'exécution d'une suite de commandes pour sauter directement à l'exécution d'une commande plus éloignée.

Pratique de la programmation

Pour apprendre à *écrire*, il est aussi important de connaître la *grammaire* que de lire les *classiques*. De la même façon, pour apprendre à *programmer*, il convient de maîtriser les *règles du langage* et en même temps d'étudier un certain nombre d'*exemples classiques*. En essayant de reproduire les 'classiques', vous comprendrez mieux les *règles* du langage et vous développerez votre propre *style* de programmation. Dans ces notes de cours, on va examiner un certain nombre d'algorithmes classiques. Le lecteur est averti que leur programmation correspond au *style* de l'auteur de ces notes. Des variations et des améliorations sont certainement possibles et encouragées !

3.1 Commandes de base et séquentialisation

Les commandes de bases comprennent l'affectation d'une valeur à une variable, la commande d'écriture (printf) et de lecture (scanf), l'appel et le retour de fonction (return). Il est aussi possible de composer les commandes pour obtenir des commandes plus complexes. Le premier opérateur de composition est la *séquentialisation* qui dans de nombreux langages est dénoté par le point virgule :

$$C_1; C_2 \ .$$

L'interprétation de cette commande composée est qu'on exécute d'abord C_1 et ensuite C_2. On notera que l'opération de séquentialisation est *associative* :

$$(C_1; C_2); C_3 \equiv C_1; (C_2; C_3)$$

il est donc inutile de mettre les parenthèses.

3.2 Branchement

Un deuxième exemple d'opérateur de composition de commandes est le branchement. La forme de base est :

$$\mathsf{if}\ (b)\{C_1\}\ \mathsf{else}\ \{C_2\}$$

L'interprétation est qu'on évalue une *condition logique* b. Si elle est vraie on exécute C_1 et sinon C_2. Dans la syntaxe de C, on admet aussi une version sans branche else :

$$\mathsf{if}\ (b)\{C_1\}$$

qui est équivalente à :

$$\mathsf{if}\ (b)\{C_1\}\ \mathsf{else}\ \{\mathsf{skip}\}$$

où skip est une abréviation pour une commande qui ne fait rien d'observable.

Conditions logiques

Le langage C dispose d'un certain nombre d'opérateurs pour construire des conditions logiques. Notamment :

==	égalité
<	inégalité
||	disjonction
&&	conjonction
!	négation

Par exemple, on peut écrire :

$$(\mathsf{x} == \mathsf{y})\ \&\&\ (\mathsf{x} < \mathsf{z} + 5\ \ ||\ \ \mathsf{x}! = \mathsf{y} + \mathsf{z})$$

Remarque 3 *En C, l'évaluation d'une condition logique se fait de gauche à droite et de façon paresseuse, c'est-à-dire dès qu'on a déterminé la valeur logique de la condition on omet d'évaluer les conditions qui suivent. Ainsi, en C, on peut écrire la condition logique :*

$$(\mathsf{x}! = 0)\ \&\&\ (\mathsf{y}/\mathsf{x} == 3) \tag{3.1}$$

qui ne produit pas d'erreur même si x est égal à 0. En effet, si x est égal à 0 alors la première condition $\mathsf{x}! = 0$ est fausse et ceci suffit à conclure que la condition logique est fausse. Le problème avec ce raisonnement est qu'en C une expression logique peut être vraie, fausse, produire une erreur, produire un effet de bord (par exemple imprimer une valeur) et même faire boucler le programme. Il en suit que la conjonction en C n'est pas commutative. Par exemple, la condition :

$$(\mathsf{y}/\mathsf{x} == 3)\ \&\&\ (\mathsf{x}! = 0) \tag{3.2}$$

n'est pas équivalente à la condition (3.1) ci-dessus.

Exemple 10 *Pour pratiquer le branchement, on considère la conception d'un programme qui lit trois coefficients a, b, c et imprime les zéros du polynôme $ax^2 + bx + c$. Une solution possible est la suivante :*

```
int main() {
double a,b,c,delta,root,sol1,sol2;
printf("Entrez coeff a : "); scanf("%lf",&a);
printf("Entrez coeff b : "); scanf("%lf",&b);
printf("Entrez coeff c : "); scanf("%lf",&c);
delta = b*b-(4*a*c);
if (a==0 && b==0)              // degré 0
   {if (c==0){printf("Tout nombre est une solution\n");}
    else {printf("Pas de solution");}}
    else if (a==0)             // degré 1
   {sol1=-c/b; printf("L'unique solution est :%lf\n",sol1);}
    else if (delta==0)         // degré 2
   {sol1=-b/(2*a); printf("L'unique solution est :%lf\n",sol1);}
         else if (delta<0) {printf("Pas de solution\n");}
         else {root = sqrt(delta);
               sol1=(-b+root)/(2*a);
               sol2=(-b-root)/(2*a);
               printf("Les deux solutions sont :%lf,%lf\n",sol1,sol2);}
return 0;}
```

La première partie de la fonction main *lit les coefficients. Dans la deuxième partie on trouve un certain nombre d'instructions de branchement imbriquées qui nous permettent de distinguer les différentes situations qui peuvent se présenter. Il est fortement conseillé de visualiser d'abord avec un schéma qui peut prendre la forme d'un arbre binaire les différentes possibilités. Une fois qu'on a vérifié la correction du schéma on procédera à son codage en* C.

Exemple 11 *On souhaite concevoir un programme qui reçoit en entrée le nombre de notes de* 50*,* 20 *et* 10 *euros dont on dispose ainsi qu'une somme s à payer. Si possible, le programme doit imprimer une façon de payer (exactement) la somme s avec les notes dont on dispose. Sinon, le programme imprime un message qui dit que le payement de la somme n'est pas possible. Pour simplifier le problème, on va supposer qu'on dispose d'au moins une note de* 10 *euros. Dans ce cas, la stratégie suivante permet de résoudre le problème : on paye autant que possible, c'est-à-dire sans dépasser la somme s, avec des notes de* 50*, ensuite avec des notes de* 20 *et enfin avec des notes de* 10*. Le lecteur est invité à vérifier que sans l'hypothèse sur les notes de* 10 *euros, cette stratégie ne permet pas toujours de trouver une solution. Un codage possible de la stratégie en* C *est ci-dessous où on laisse au lecteur le soin de compléter les parties qui concernent la lecture des paramètres et l'impression du résultat.*

```
int main() {
int n50, n20, n10, s, p50, p20, p10;
/* on lit n50, n20, n10 et s */
/* calcul */
 if ((s/50) <= n50) {p50=(s/50);} else {p50=n50;} ;  s=s-(50*p50);
 if ((s/20) <= n20) {p20=(s/20);} else {p20=n20;} ;  s=s-(20*p20);
 if ((s/10) <= n10) {p10=(s/10);} else {p10=n10;} ;  s=s-(10*p10);
/* impression resultat */
 return 0; }
```

3.3 Boucles

Une boucle permet d'exécuter une commande un nombre arbitraire de fois. L'opérateur while est probablement le plus utilisé pour construire une boucle. Sa forme est :

$$\mathsf{while}(b)\{C\}\ .$$

La commande résultante évalue la condition logique b et elle termine si elle est fausse. Autrement, elle exécute la commande C et ensuite elle recommence à exécuter la commande $\mathsf{while}(b)\{C\}$. Ainsi, d'un point de vue conceptuel on a l'équivalence suivante :

$$\mathsf{while}(b)\{C\} \equiv \mathsf{if}(b)\{C; \mathsf{while}(b)\{C\}\} \ .$$

Exemple 12 *On programme l'algorithme d'Euclide pour le calcul du* pgcd *(exemple 3) en utilisant une boucle* while*.*

```
int main(){
int a,b;
/* lire a et b */
int aux;
while (b!=0){
  aux=b;
  b=a%b;
  a=aux;}
/* imprimer a */
return 0;}
```

Tant que b *n'est pas* 0*, on remplace* a *par* b *et* b *par* a mod b*. Cependant, le langage* C *ne permet pas d'effectuer deux affectations en même temps. Pour cette raison, on introduit une variable auxiliaire* aux *qui garde la valeur originale de* b *pendant qu'on remplace* b *par* a mod b*. Il s'agit d'une technique standard pour permuter le contenu de deux variables.*

Exemple 13 *On utilise la boucle* while *pour programmer un exemple de* recherche dichotomique*. Le principe général de la recherche dichotomique est qu'à chaque itération soit on trouve l'élément recherché soit on divise par deux la taille de l'espace de recherche. On applique ce principe au problème du calcul d'une approximation à un ϵ près de la racine carrée d'un nombre flottant $x \geq 1$.*[1] *A priori, on sait que $\sqrt{x} \in [1, x]$. Plus en général, si on sait que $\sqrt{x} \in [\mathsf{low}, \mathsf{high}]$ avec $1 \leq \mathsf{low} < \mathsf{high}$ on peut appliquer le raisonnement suivant :*

- *Si $|\mathsf{high} - \mathsf{low}| \leq \epsilon$ on connaît $\sqrt{x}$ à un ϵ près.*
- *Sinon, on calcule le carré du milieu de l'intervalle $[\mathsf{low}, \mathsf{high}]$ et on le compare à x. Si la valeur est plus grande il faut continuer la recherche dans la moitié gauche de l'intervalle et autrement dans la moitié droite.*

Une programmation possible de la méthode est la suivante.

```
int main (){
/* lire x */
double low = 1;
double high = x;
double mid;
while ((high-low)> eps)
 {mid = (high+low)/2;
  if (mid*mid>x){high=mid;} else {low=mid;}};
/* imprimer x */
return 0;}
```

1. Bien sûr on pourrait aussi utiliser la fonction de bibliothèque sqrt pour résoudre ce problème.

Boucle for

Pour améliorer la lisibilité du programme, on utilise aussi une boucle dérivée for avec la forme :

$$\mathsf{for}(C_1; b; C_2)\{C\} \tag{3.3}$$

En première approximation, la boucle for est équivalente à :

$$C_1; \mathsf{while}(b)\{C; C_2\}\ . \tag{3.4}$$

Dans une bonne pratique de la boucle for, on utilise la commande $\mathsf{C_1}$ pour initialiser la boucle et la commande C_2 pour modifier les variables dont dépend la condition logique b (la condition d'arrêt). Typiquement, il s'agit d'incrémenter ou décrémenter un indice et, en lisant le texte du programme, il est aisé de déterminer combien de fois le corps C de la boucle for sera itéré.

Exemple 14 *On souhaite lire un nombre naturel n et imprimer ses diviseurs propres (différents de 1 et n). Pour résoudre ce problème, on peut utiliser une boucle* for *qui va parcourir les entiers compris entre 2 et $n/2$.*

```
int main() {
int n,i;
/* lire n */
for (i=2;i<=n/2;i=i+1)
  {if (n%i==0) {printf("%d\n",i);} };
return 0;}
```

3.4 Rupture du contrôle

On présente un certain nombre de commandes qui permettent de s'extraire de la commande en exécution et de sauter à un autre point du contrôle. On les présente en ordre décroissant de puissance :

- exit(n) pour terminer l'exécution du *programme*. Convention C : on utilise $n = 0$ pour indiquer une terminaison normale.
- return pour terminer l'exécution d'une *fonction*.
- break pour terminer l'exécution de la *boucle* dans laquelle on se trouve.
- continue pour *reprendre l'exécution au début de la boucle* dans laquelle on se trouve. [2]

Exemple 15 *Dans la boucle suivante on va imprimer $5, 4, 3, 2, 1$. En particulier le décrément* x- - *après* continue *n'est jamais exécuté.*

```
int x=5;
while (x>0){printf("%d\n",x) ; x--; if (x>0){continue;}; x--;}
```

La boucle suivante ne terminerait pas sans break.

```
int acc=0; int i;
for(i=1;i<=n;i--){acc=i+acc; if (i<-100){break;};};
```

Remarque 4 *Il faut utiliser* break *et* continue *seulement si on se trouve dans une boucle (ou dans une commande* switch *qui sera discutée dans la section 3.5 qui suit).*

2. Si on se trouve dans une boucle for, continue va quand même exécuter la commande d'incrément/décrément. Pour cette raison, la transformation de la boucle for en boucle while décrite dans (3.4) doit être raffinée.

3.5 Aiguillage switch

La commande switch (*aiguillage*) permet aussi d'effectuer des branchements dans le calcul. Elle est présentée ici car elle est utilisée souvent en combinaison avec les commandes break ou return. Voici un exemple :

```
switch(x){
case 0 : printf("%d\n",x);
case 1:  printf("%d\n",x+1);
default: printf("%d\n",x+2); };
```

Si x est 0 on imprime $0, 1, 2$ si x est 1 on imprime 2 et 3 et autrement on imprime $x + 2$. Le switch évalue une expression entière qui donne une valeur n et ensuite exécute toutes les branches à partir de celle de la forme case n (si elle existe) et la branche default autrement. En pratique, on a souvent besoin d'exécuter *seulement* la branche qui correspond à case n. On obtient ce comportement en insérant une commande break à la fin de chaque branche. Ainsi, notre exemple devient :

```
switch(x){
case 0 : printf("%d\n",x); break;
case 1:  printf("%d\n",x+1); break;
default: printf("%d\n",x+2); };
```

Dans ce cas, si x est 0 on imprime 0, si x est 1 on imprime 2 et autrement on imprime $x + 2$.

3.6 Énumération de constantes

Pour améliorer la lisibilité d'un programme qui depend d'un certain nombre de valeurs entières constantes, on peut regrouper ces constantes dans une déclaration. Par exemple, on peut définir :

```
typedef enum {ZERO, ONE} bool;

bool not(bool x){
  switch(x){
  case ZERO: return ONE;
  case ONE: return ZERO;
  default: return x;
  }}
```

Par défaut, le compilateur associe aux noms une suite d'entiers croissants : $0, 1, 2, \ldots$ Dans l'exemple, on associe donc l'entier 0 à ZERO e l'entier 1 à ONE. Notez que la spécification de C n'exige pas que l'argument ou le résultat de la fonction not soit bien un ZERO ou un ONE. Par exemple, avec gcc l'appel not(3) ne produit pas d'erreur au moment de la compilation ou de l'exécution et retourne 3.

Chapitre 4

Fonctions

Un programme C est composé d'une liste de fonctions qui peuvent s'appeler mutuellement. Les fonctions sont un élément essentiel dans la modularisation et le test d'un programme.

Si une tâche doit être répétée plusieurs fois c'est une bonne pratique de lui associer une fonction ; ceci permet de produire un code plus compact tout en clarifiant le fonctionnement du programme.

Aussi si une tâche est trop compliquée il est probablement utile de la décomposer en plusieurs fonctions. Le code de chaque fonction devrait tenir dans une page (20-30 lignes) et avant de tester le programme dans son intégralité, il convient de s'assurer de la fiabilité de chaque fonction.

4.1 Appel et retour d'une fonction

Comme indiqué dans la section 1.2, une fonction est un *segment de code* identifié par un *nom*. En C, la forme d'une fonction est la suivante :

```
t f(t1 x1, ..., tn xn)
{corps de la fonction}
```

La première ligne spécifie l'*interface* (ou *en tête*) de la fonction, à savoir le nom de la fonction (f), le type du résultat (t) et les types et les noms des arguments (t1 x1,...,tn xn). Quand on *appelle* une fonction on définit les *valeurs* de ses arguments. Par exemple, dans l'appel $f(e_1, \ldots, e_n)$ on évalue les expressions $e_1, \ldots, e_n$ et on affecte leurs valeurs aux variables $x1, \ldots, xn$. On dit que l'appel d'une fonction est *par valeur*. Un *appel de fonction est une expression* dont le type est le type du résultat de la fonction. Donc l'appel $f(e_1, \ldots, e_n)$ a type t. Par ailleurs, le corps de la fonction contient des commandes return e où e est une expression de type t.

Exemple 16 *Voici un programme simple composé de* 3 *fonctions et d'une variable globale* pi. *Une variable globale est une variable dont la déclaration n'est pas dans le corps d'une fonction. Une variable globale est visible dans toutes les fonctions du programme sauf si elle est cachée par une déclaration locale (plus de détails dans la section 4.2). On remarquera que les appels de fonction peuvent être imbriqués comme dans* imprimer(1.0, circonference(1.0)) ; *Dans ce cas, il faut d'abord exécuter l'appel interne (*circonference(1.0)*) et ensuite celui externe* imprimer$(1.0, 6.28\cdots)$

```
double pi = M_PI;                   //constante pi définie dans math.h
double circonference(double r){
  return 2 * pi * r;}
void imprimer(double r, double c){
  printf("La circonference de %lf est %lf\n", r, c);}
int main(){
  printf("%lf\n",pi);
  imprimer(1.0, circonference(1.0));
  imprimer(2.0, circonference(2.0));
  return 0;}
```

4.2 Portée lexicale

Dans un programme, en particulier dans un programme avec plusieurs fonctions, la même variable peut être déclarée plusieurs fois (et avec plusieurs types).

Une bonne pratique consiste à utiliser des noms différentes pour les arguments et les variables locales et si possible à éviter d'utiliser le même nom pour les variables locales et les variables globales. A défaut, la variable locale couvrira la globale.

Exemple 17 *Voici un programme composé de* 2 *variables globales et* 3 *fonctions. Quels sont les entiers imprimés ? Voici des indices. Dans* (4)*,* x *est l'argument de la fonction alors que dans* (5) y *est la variable globale. Dans* (7)*, l'affectation n'a pas d'effet sur le* x *de la fonction* main *ni sur le* x *global. Dans* (9)*, la déclaration de* y *cache la variable globale dans le segment de code délimité par les accolades. Ainsi, dans* (11) *on se réfère à la variable globale et dans* (12) *on peut déclarer à nouveau une variable* y*. Dans* (16)*, on passe la valeur de la variable* x *du* main*, ainsi l'incrément dans* (7) *n'a pas d'effet sur la variable du* main*.*

```
int x=5;              //(1)
int y=6;              //(2)
void imprimer(int x){ //(3)
  printf("%d\n",x);   //(4)
  printf("%d\n",y);}  //(5)
void portee(int x){   //(6)
  x=x+1;              //(7)
 imprimer(x);         //(8)
 {int y=10;           //(9)
  imprimer(y);}       //(10)
  imprimer(y);        //(11)
  int y=20;           //(12)
  imprimer(y);}       //(13)
int main(){           //(14)
  int x = 4;          //(15)
  portee(x);          //(16)
  imprimer(x);        //(17)
  return 0;}          //(18)
```

Exercice 2 *Dans le programme suivant on trouve* 10 *appels à la fonction* imprimer*. Pour chaque appel vous devez prévoir combien de fois il sera exécuté et avec quelles valeurs.*

```
int x=5;
int y=6;
void imprimer(int x){printf("%d\n",x);}
void portee(int x){
```

```
  x=x+1; imprimer(x);                                  // (1)
  {int y=10; imprimer(y);}                             // (2)
  imprimer(y);                                         // (3)
  int y=20; imprimer(y);}                              // (4)
int main(){
  int x = 4;
  portee(x);
  imprimer(x);                                         // (5)
  controle();
  return 0;}
void controle(){
  if (x>0){imprimer(--x);}                             // (6)
  else {imprimer(x++);};                               // (7)
  int acc=1, n=2; int k;
  for(k=1;k<=n;k++) {acc=k*acc;imprimer(acc);};        // (8)
  int i=2;
  while(i>0){acc=acc-i;i--;imprimer(acc);              // (9)
             if (i==1){break;} else{ continue;}}
             imprimer(f(3));                           // (10)
             return ;}
int f(int x){
  switch(x){
  case 0: return 1;
  case 1: return 2;
  default: return f(x-1)*f(x-2); } }
```

4.3 Argument-résultat, Entrée-sortie

Une fonction C *doit* prendre n *arguments* ($n \geq 0$) et rendre un *résultat* (éventuellement de type void). Par ailleurs, comme effet de bord, elle *peut* aussi *lire* des valeurs (avec scanf par exemple) et *imprimer* des valeurs (avec printf par exemple). Il faut bien comprendre que :

— prendre en argument est différent de lire une valeur.

— rendre un résultat est différent d'imprimer une valeur.

Un *argument* est passé par la fonction appelante alors que la *valeur lue* vient de l'écran ou d'un fichier. De même une fonction rend le *résultat* à la fonction appelante alors qu'elle *imprime une valeur* à l'écran ou dans un fichier. Prendre en argument/rendre un résultat est donc une interaction entre deux fonctions alors que lire une valeur/imprimer une valeur est une interaction entre une fonction et l'utilisateur (ou un fichier externe au programme).

4.4 Méthode de Newton-Raphson

La méthode de Newton-Raphson est une méthode élémentaire utilisée en calcul numérique pour trouver le zéro d'une fonction dérivable f. On commence par un point x_0 'assez proche d'un point x tel que $f(x) = 0$. Au pas i, on détermine x_{i+1} par la formule :

$$f'(x_i) = \frac{f(x_i) - 0}{x_i - x_{i+1}}$$

dont dérive :

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)} \ . \tag{4.1}$$

Le calcul sur les flottants étant approché, on fixe un niveau de précision ϵ souhaité et on arrête l'itération dès que (cf. recherche dichotomique de l'exemple 13) :

$$|x_i - x_{i+1}| < \epsilon \ .$$

Notons au passage qu'il faut étudier f et le point initial x_0 pour s'assurer de la convergence vers un zéro de la fonction. En effet, la méthode peut ne pas converger même si f est un polynôme. Considérons une mise en oeuvre pour la fonction $f(x) = x^2 - a$ où $a \geq 0$. Il faut : (1) lire la valeur a, (2) itérer l'opération (4.1) jusqu'au niveau de précision souhaité et (3) imprimer le résultat. Par ailleurs, il convient de prévoir une fonction C qui correspond à la fonction mathématique f. Ainsi si l'on souhaite adapter le programme à une autre fonction mathématique il suffira de modifier la fonction C correspondante. En suivant ces considérations, une mise en oeuvre possible est la suivante.

```
#define epsilon 10E-10
double a;
double fun(double x){return x*x - a;}
double itere(double x){
  double xnext;
  while (1)
    {xnext = x - fun(x)/(2*x);
     if (fabs(xnext-x)<epsilon){return xnext;}
     else {x=xnext;} }}
int main(){
  double x;
  // lire a
  x=itere(a);
  //imprimer x
  return 0;}
```

4.5 Intégration numérique

Pour calculer une approximation numérique de l'intégrale d'une fonction f dans l'intervalle $[a, b]$ avec $a < b$ on peut découper l'intervalle $[a, b]$ en n intervalles de taille $(b - a)/n$ et approximer la surface de chaque petit intervalle par la surface du trapèze. Plus précisément, si l'on veut approximer :

$$\int_a^b f(x)\mathrm{d}x,$$

on fixe le nombre n d'*intervalles* et $h = \frac{(b-a)}{n}$. Soit :

$$x_i = a + ih \qquad 0 \leq i \leq n \ .$$

La *surface* S_i du trapèze déterminé par x_i et x_{i+1} est :

$$S_i = \frac{(f(x_i) + f(x_{i+1}))h}{2} \ .$$

En *additionnant les surfaces* des trapèzes on dérive :

$$\Sigma_{i=0,\ldots,n-1} S_i = \frac{h}{2}(2(\Sigma_{i=1,\ldots,n-1} f(x_i)) + f(a) + f(b)) \ .$$

Il convient de dissocier le calcul de la somme des surfaces des trapèzes. Ainsi la structure d'un programme pour ce problème pourrait être celle ci-dessous où l'on suppose que f est la fonction de bibliothèque *sinus*. Comme pour la méthode de Newton-Raphson, on pourrait ajouter des fonctions pour lire et imprimer et on pourrait encapsuler la fonction mathématique dont on calcule l'intégrale dans une fonction C séparée.

```
double integral(double a,double b,int n){
  double h=(b-a)/n;
  double acc=0;
  double x=a+h;
  int i;
  for (i=1;i<=n-1;i++){acc=acc+sin(x);x=x+h;}
  acc= (2*acc+sin(a)+sin(b));
  acc= (acc*h)/2;
  return(acc);}
int main() {
  double a,b,val; int n;
  /* lire a, b, n */
  val=integral(a,b,n);
  /* imprimer résultat */
  return 0;}
```

4.6 Conversion binaire-décimal

Si $d_n \cdots d_0$ est un nombre en base B (donc $d_i \in \{0, 1, \ldots, B-1\}$), la valeur du nombre est :

$$\Sigma_{i=0,\ldots,n} d_i B^i \ .$$

Pour convertir un *binaire* en *décimal* il suffit d'appliquer la formule. Par exemple, 101 a comme valeur :

$$1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 = 5 \ .$$

D'autre part, pour convertir un *décimal* en *binaire* il suffit d'itérer l'opération de quotient par 2 et reste. Par exemple :

$$\begin{array}{lll} 19 & = 2 \cdot 9 + \mathbf{1} & \text{(bit le moins significatif)} \\ 9 & = 2 \cdot 4 + \mathbf{1} & \\ 4 & = 2 \cdot 2 + \mathbf{0} & \\ 2 & = 2 \cdot 1 + \mathbf{0} & \\ 1 & = 2 \cdot 0 + \mathbf{1} & \text{(bit le plus significatif)} \end{array}$$

Donc :

$$\begin{array}{rl} 19 & = 2 \cdot (2 \cdot (2 \cdot (2 \cdot (2 \cdot 0 + \mathbf{1}) + \mathbf{0}) + \mathbf{0}) + \mathbf{1}) + \mathbf{1} \\ & = 2^4 \cdot \mathbf{1} + 2^3 \cdot \mathbf{0} + 2^2 \cdot \mathbf{0} + 2^1 \cdot \mathbf{1} + 2^0 \cdot \mathbf{1} \end{array}$$

La représentation de 19 en base 2 est 10011.

Exemple 18 *Considérons maintenant un problème de mise en oeuvre. On souhaite effectuer deux opérations.*

1. *Lire un nombre de type* int *et imprimer sa représentation binaire (aussi de type* int*). Par exemple :*

On lit	*On imprime*
19	10011

2. *Lire un nombre de type* int *dont les chiffres varient dans* $\{0,1\}$*, le* voir comme un nombre binaire *et imprimer sa représentation décimale (aussi de type* int*). Par exemple :*

On lit	*On imprime*
10011	19

Le lecteur remarquera que dans cet exemple on utilise un sous-ensemble des valeurs de type int *(celles dont les chiffres varient sur* 0 *et* 1*) pour représenter les nombres binaires.*

Dans une mise en oeuvre il est naturel d'introduire une fonction pour chaque conversion. Remarquons que la solution ci-dessous utilise la fonction de bibliothèque assert *de la bibliothèque* assert.h*. La commande* assert(b) *évalue la condition logique* b*. Si la condition est fausse le calcul s'arrête et un message d'erreur est émis qui permet d'identifier l'assertion qui n'est pas valide. Avec la fonction* assert*, on a une façon simple et fort utile de documenter et tester un programme.*

```
int dec_to_bin (int d){
  int q,r;
  if (d==0) return 0;
  else {r=d%2; q=d/2; return (r+10*dec_to_bin(q));}}
int bin_to_dec (int b){
  int q,r;
  if (b==0) return 0;
  else {r=b%10; q=b/10;
  assert((r==0) || (r==1));
  return (r+2*bin_to_dec(q));}}
int main(){
  printf("Entrez 10 pour décimal et 2 pour binaire\n");
  int choix; int x;
  scanf("%d",&choix);
  assert((choix==2)||(choix==10));
  if (choix==10){printf("Entrez un nombre décimale\n");}
  else {printf("Entrez un nombre binaire\n");}
  scanf("%d",&x);
  if (choix==10) printf("%d\n",dec_to_bin(x));
  else printf("%d\n", bin_to_dec(x));
  return 0;}
```

Chapitre 5

Fonctions récursives

Un programme C est composé d'une liste de fonctions qui peuvent s'appeler mutuellement. En particulier, une fonction peut s'appeler elle même ; il s'agit alors d'un exemple de *fonction récursive* (n fonctions qui s'appellent mutuellement sont aussi des fonctions récursives). On a déjà examiné dans l'exemple 3 la programmation de l'algorithme d'Euclide par une fonction récursive. Les fonctions récursives permettent de programmer aisément les définitions par récurrence qu'on trouve souvent en mathématiques. Aussi, un certain nombre d'algorithmes qui suivent une stratégie diviser pour régner se programment naturellement de façon récursive ; par exemple, la recherche dichotomique et certains algorithmes de tri qu'on étudiera dans la section 8.1 suivent cette stratégie. On a vu dans l'exemple 12 que l'algorithme d'Euclide peut se programmer aussi avec une boucle ; on dira aussi de façon *itérative*. La récursion utilisée dans la programmation de l'algorithme d'Euclide est de type *terminal* dans le sens qu'après l'appel récursif il ne reste plus rien à faire et la fonction retourne immédiatement. [1] Il se trouve que pour la *récursion terminale* (*tail recursion* en anglais), un compilateur optimisant peut générer automatiquement un programme itératif équivalent. Dans la section 5.1, on pratique la programmation récursive et itérative dans le cadre du problème de l'évaluation de polynômes.

Comme on l'a vu dans la section 1.2, l'appel et le retour de fonction manipule implicitement une *pile de blocs d'activation*. Il s'avère que dans certaines situations cette structure de données permet une programmation élégante et compacte. Dans la section 5.2, on illustre une telle situation avec le problème de la tour d'Hanoï.

Enfin, il y a aussi des situations dans lesquelles la programmation d'une définition par récurrence à l'aide d'une fonction récursive peut générer un programme particulièrement inefficace. Dans la section 5.3, on examine différentes stratégies pour contourner ce problème dans le contexte du calcul de la suite de Fibonacci.

5.1 Évaluation de polynômes

Considérons le problème de l'évaluation d'un polynôme de degré n :

$$p(x) = a_0 + a_1 x + \cdots + a_n x^n$$

dans un point x. Un premier algorithme peut consister à calculer les sommes partielles :

$$a_0, \qquad a_0 + a_1 x, \qquad a_0 + a_1 x + a_2 x^2, \cdots$$

1. Il s'agit d'une intuition, on ne donnera pas de définition formelle de la récursion terminale.

en calculant en parallèle les puissances de x :

$$x^0, \qquad x^1 = x \cdot x^0, \qquad x^2 = x \cdot x, \qquad x^3 = x \cdot x^2, \ldots$$

Pour ce calcul, il faut donc effectuer $2 \cdot n$ multiplications ainsi que n sommes. Cependant le coût d'une somme est bien inférieur à celui d'une multiplication et donc on peut considérer que le coût du calcul dépend essentiellement du nombre de multiplications.

Règle de Horner

La règle de Horner est un autre algorithme pour évaluer un polynôme de degré n dans un point qui demande seulement n multiplications. On définit :

$$\begin{aligned} h_0 &= a_n \\ h_i &= h_{i-1}x + a_{n-i} \quad 1 \leq i \leq n \ . \end{aligned}$$

On remarque que :

$$h_i = a_n x^i + a_{n-1}x^{i-1} + \cdots + a_{n-i+1}x + a_{n-i} \ .$$

Donc $p(x) = h_n$ et on peut calculer $p(x)$ avec seulement n *multiplications* !

Exemple 19 *Pour mettre en oeuvre l'évaluation d'un polynôme on va faire l'hypothèse que le programme lit le degré du polynôme, un point où il faut évaluer le polynôme et les coefficients du polynôme. Pour l'instant, on ne dispose pas d'une structure de donnée pour mémoriser (de façon simple) n entiers où n est variable ; les tableaux qui seront discutés dans le chapitre 7 feront l'affaire. Il s'agit donc de lire les coefficients et en même temps de faire progresser l'évaluation du polynôme.* [2] *Pour mettre en oeuvre l'algorithme on a 4 choix possibles. En effet, on peut choisir entre la méthode d'évaluation directe et la méthode de Horner et aussi entre une programmation par récursion et une par itération. On présente ci-dessous la méthode de Horner programmée de façon récursive et itérative et on laisse en exercice le même problème pour la méthode directe. Notez que dans la version récursive on lit les coefficients dans l'ordre $a_0, a_1, \ldots, a_n$ alors que dans la version itérative on procède dans l'ordre inverse.*

```
double horner_rec(int i, double x, int n){
  double a;
  printf("Entrez coefficient %d\n",i);
  scanf("%lf",&a);
  if (i==n){return a;}
  else return (a+ x* horner_rec(i+1,x,n));}
double horner_it(double x, int n){
  double a;  double b;
  printf("Entrez coefficient %d\n",n);
  scanf("%lf",&a);
  int i;
  for (i=1;i<=n;i=i+1)
  {printf("Entrez coefficient %d\n",(n-i));
  scanf("%lf",&b);
  a=b+x*a;}
  return a;}
```

2. On voit ici un exemple d'algorithme *en ligne* (*on line* en anglais) dans lequel le programme ne dispose pas d'assez de mémoire ou de temps pour mémoriser toutes les entrées avant de commencer le calcul. Typiquement, on trouve ce type d'algorithme dans des situations où il faut traiter des grandes masses de données et/ou le processeur qui traite ces données a un pouvoir de calcul limité.

5.2 Tour d'Hanoï

Le jeu de la tour d'Hanoï est bien connu. On dispose de 3 pivots et de n disques de diamètre différent qu'on peut enfiler dans les pivots. Au début du jeu, tous les disques sont enfilés sur le premier pivot par ordre de diamètre décroissant (le plus petit diamètre est au sommet).

Une action élémentaire du jeu consiste à déplacer 1 disque du sommet d'une pile au sommet d'une autre pile en gardant la propriété qu'un disque n'est jamais au dessus d'un disque de diamètre inférieur.

Le problème est de trouver une suite d'actions élémentaires qui permettent de transférer la pile de n disques du pivot 1 au pivot 2 (par exemple).

Pour $n = 1$, une suite est $1 \to 2$. Pour $n = 2$, une suite est $1 \to 3; 1 \to 2; 3 \to 2$. Pour $n = 3$, ça devient déjà plus compliqué, mais heureusement la solution du problème s'exprime naturellement de façon *récursive* :

$$\begin{array}{ll} \mathsf{Hanoi}(i,j,1) & = i \to j \\ \mathsf{Hanoi}(i,j,n) & = \mathsf{Hanoi}(i,k,n-1); i \to j; \mathsf{Hanoi}(k,j,n-1)\ , \end{array}$$

où i, j, k sont 3 pivots *distincts*. Le raisonnement est le suivant : pour déplacer n disques du pivot i au pivot j ($i \neq j$), on peut commencer par déplacer $n-1$ disques du pivot i au pivot k ($k \neq i$ et $k \neq j$). Ensuite, on déplace le disque de diamètre maximal qui se trouve au pivot i au pivot j et on termine en déplaçant $n-1$ disques du pivot k au pivot j. Une possible mise en oeuvre en C est la suivante :

```
void hanoi (int n, int p1, int p2){
  int p3=troisieme(p1,p2);
  if(n==1){imprimer(p1,p2);}
  else {hanoi(n-1,p1,p3); imprimer(p1,p2); hanoi(n-1,p3,p2);}
  return;}
```

On laisse au lecteur le soin de programmer la fonction troisieme qui prend $p1, p2 \in \{0, 1, 2\}$ tels que $p1 \neq p2$ et rend l'entier $p \in \{0, 1, 2\}$ différent de $p1$ et $p2$. On notera que chaque appel à la fonction hanoi avec $n > 1$ génère deux appels à la même fonction. En particulier, quand on commence à calculer hanoi(n − 1, ...) on utilise la pile des blocs d'activations pour se souvenir qu'il reste encore à exécuter imprimer(...) ainsi qu'un deuxième appel hanoi(n − 1, ...). Le lecteur est invité à modifier le code ci-dessus pour qu'il trace chaque appel à la fonction hanoi en imprimant un petit message.

5.3 Suite de Fibonacci

La suite de Fibonacci est définie par :

$$\begin{array}{ll} f(0) & = 0 \\ f(1) & = 1 \\ f(n+2) & = f(n) + f(n+1)\ . \end{array}$$

Il y a des mathématiques non-triviales autour de cette suite... mais ici on s'y intéresse parce que elle illustre un *problème de mise en oeuvre* qu'on rencontre parfois dans les définitions récursives. Une mise en oeuvre directe de la fonction f pourrait être la suivante.

```
int fibo_rec(int n){
  switch(n){
  case 0: return 0 ;
  case 1: return 1;
  default: return fibo_rec(n-2)+fibo_rec(n-1);}}
```

Cette solution est particulièrement inefficace car on recalcule plusieurs fois la fonction f sur les mêmes arguments. Dans le cas de la suite de Fibonacci, il est facile de concevoir une version itérative dans laquelle on calcule $f(0), f(1), f(2), \ldots$ exactement une fois et au pas $i \geq 2$ on se souvient de la valeur de la fonction f dans $i-1$ et $i-2$.

```
int fibo_it(int n){
  int x=0;
  int y=1;
  switch(n){
  case 0:  return 0;
  case 1:  return 1;
  default: {int z=0; int i;
            for (i=2;i<=n;i++){z=x+y; x=y; y=z;}
            return(z);}
}}
```

Il est intéressant de noter qu'on peut mettre en oeuvre cette même idée en utilisant une fonction récursive un peu plus générale (fonction fibo_aux).

```
int fibo_aux(int n,int x, int y,int i){
  if (i==n) return(y);
  else return fibo_aux(n,y,x+y,i+1);}
int fibo_rec_eff(int n){
  switch(n){
  case 0: return 0;
  case 1: return 1;
  default: return fibo_aux(n,0,1,1);}
}
```

Il existe aussi une technique générale dite de *mémoïsation* qui permet de transformer automatiquement une fonction récursive. On mentionne l'idée générale sans aller dans les détails car on ne dispose pas encore des structures de données nécessaires. On associe à la fonction une structure de données dans laquelle on *mémoïse* tous les arguments passés à la fonction ainsi que les valeurs retournées. Chaque fois qu'on appelle la fonction avec un argument on vérifie d'abord dans la structure si la fonction a été déjà appelée avec le même argument et dans ce cas on retourne directement le résultat. Autrement, on effectue le calcul et on mémorise le résultat dans la structure.

Chapitre 6

Complexité asymptotique

On introduit la notion de complexité asymptotique d'un algorithme. Il s'agit d'une mesure qui n'est pas très sensible aux détails de la mise en oeuvre et qui permet d'avoir une première estimation de l'efficacité d'un algorithme.

6.1 O-notation

Définition 1 (O-notation) *Soient $f, g : \mathbf{N} \to \mathbf{N}$ deux fonctions sur les nombres naturels. On dit que f est $O(g)$ si :*

$$\exists\, k, n_0 \geq 0 \;\; \forall\, n \geq n_0 \;\; f(n) \leq k \cdot g(n) \;\; .$$

Exemple 20 *Voici des exemples et des non-exemples.*

3457	*est*	$O(1)$
$25 \cdot n + 32$	*est*	$O(n)$
$7 \cdot n \cdot log(n) + 1$	*est*	$O(n \cdot \log_2(n))$
$n \cdot \sqrt{n} - 50$	*n'est pas*	$O(n \cdot log(n))$
3^n	*n'est pas*	$O(2^n + n^5)$.

Définition 2 (fonction de coût) *Soit A un algorithme qui termine. On associe à A une fonction de côut $c_A : \mathbf{N} \to \mathbf{N}$ telle que, pour tout n, $c_A(n)$ est le coût maximal d'une exécution de l'algorithme A sur une entrée de taille au plus n.*

Typiquement, la taille d'une entrée est le nombre de bits nécessaires à sa représentation et le coût d'une exécution est le *temps* mesuré comme le nombre d'*étapes élémentaires* de calcul. Ce qui constitue une étape élémentaire dépend du modèle de calcul. Par exemple, on peut considérer qu'un accès à la mémoire principale ou la multiplication de deux entiers sur 64 bits prennent un temps borné par une constante. D'autre part, dans certaines applications les données ne peuvent pas tenir en mémoire principale ou alors on est amené à traiter des entiers avec un grand nombre de chiffres. Dans ces cas, le coût de ces opérations sera fonction de la taille de la mémoire nécessaire à l'exécution du programme ou de la taille des entiers à traiter, respectivement. On remarque que la fonction c_A est bien définie car A termine et il y a un nombre fini d'entrées possibles de taille au plus n. On note aussi que par définition c_A est croissante : si $n \leq n'$ alors $c_A(n) \leq c_A(n')$.

Définition 3 (complexité asymptotique) *Un algorithme A est $O(g)$ si sa fonction de coût c_A est $O(g)$.*

Remarque 5 *La notation O nous donne une* information synthétique sur l'efficacité *d'un algorithme/programme. Mais notez que :*

- *Il s'agit d'une* borne supérieure*.*
- *On considère le* pire des cas*.*
- *On* cache les constantes*. Un algorithme qui prend $3n^2$* msec *est utilisable, un algorithme qui prend $2^{80}n$* msec *ne l'est pas.*
- *Le coût d'une opération élémentaire sur une* vraie machine *peut varier grandement. Par exemple on peut avoir un facteur 10^2 entre un* cache hit *(la donnée est en mémoire* cache*) et un* cache miss *(elle n'y est pas). Les optimisations effectuées par le compilateur peuvent avoir un impact important sur la complexité observée.*
- *Dans les calculs en virgule flottante, on doit aussi se soucier de la stabilité numérique des opérations.*

Pour toutes ces raisons, dans les applications, la borne O doit être confortée par une analyse plus fine *et des* tests*.*

6.2 Exposant modulaire

On considère un exemple d'analyse de complexité. On suppose que les entrées sont des nombres naturels représentés en base 2. Donc la taille d'un entier m est approximativement $\log_2(m)$. Le lecteur peut vérifier que l'algorithme standard qui calcule l'*addition* de deux nombres est $O(n)$ et que celui qui calcule la *multiplication* est $O(n^2)$ (n étant la taille de l'entrée). Au passage, on remarquera que pour représenter l'addition et la multiplication de deux nombres sur n bits il faut $n+1$ bits et $2 \cdot n$ bits, respectivement.

Considérons maintenant la situation pour la fonction d'*exponentiation*. Avec n bits on représente les nombres dans l'intervalle $[0, 2^n - 1]$. Si on prend $x \in [0, 2^n - 1]$ on aura $2^x \in [1, 2^{2^n-1}]$ et il faudra environ 2^n bits pour représenter le résultat. Avec une représentation standard des nombres, tout algorithme qui calcule la fonction exponentielle prendra au moins un temps exponentiel. En effet la simple écriture du résultat peut prendre un temps exponentiel.

Soient a et e des nombres naturels. Combien de multiplications faut il pour calculer l'exposant a^e ? Un algorithme possible est de calculer :

$$a_1 = a,\ a_2 = (a_1 \cdot a),\ \cdots,\ a_e = (a_{e-1} \cdot a)\ .$$

Cet algorithme effectue $e-1$ multiplications ce qui est *exponentiel* dans $\log_2(e)$ (à savoir la taille de e !). Mais il y a une autre méthode de calcul dites des *carrés itérés*. Soit

$$e = \Sigma_{i=0,\ldots,k} e_i 2^i$$

l'*expansion binaire* de e. Donc $e_i \in \{0, 1\}$. On applique les *propriétés de l'exposant* pour dériver :

$$\begin{aligned} a^e &= a^{\Sigma_{i=0,\ldots,k} e_i 2^i} \\ &= \Pi_{i=0,\ldots,k} (a^{2^i})^{e_i} \\ &= \Pi_{0 \le i \le k, e_i = 1} (a^{2^i})\ . \end{aligned}$$

On a alors l'*algorithme* suivant :

1. On calcule a^{2^i} pour $0 \leq i \leq k$. En remarquant que
$$a^{2^{i+1}} = (a^{2^i})^2 \ .$$
Ainsi k opérations d'*élévation au carré* sont nécessaires.
2. On détermine a^e comme le produit des a^{2^i} tels que $e_i = 1$. Au plus k *multiplications* sont nécessaires.

On arrive ainsi à une situation qui semble contradictoire : le calcul de l'exposant est forcement *exponentiel* mais on peut le calculer avec un nombre *linéaire* de multiplications. Le fait est que les multiplications opèrent sur des données dont la taille peut doubler à chaque itération. Donc à la dernière itération on peut devoir multiplier deux nombres dont la taille est exponentielle en la taille des données en entrée. Mais pas tout est perdu ! On peut contrôler la taille des données si l'on passe à l'arithmétique modulaire. L'exposant modulaire :

$$(a^e) \mod m$$

prend en entrée 3 entiers : la base a, l'exposant e et le module m. On suppose $0 \leq a, e \leq m$. Pour représenter l'entrée on a donc besoin d'environ $3 \cdot k$ bits où $k = \log_2(m)$. Soit $k = \log_2(m)$. La *multiplication* de deux nombres de k bits demande $O(k^2)$. Le calcul du *reste* de la division d'un nombre de $2k$ bits (la multiplication de 2 nombres de k bits) par un nombre de k bits (le module) peut aussi se faire en $O(k^2)$. Le calcul de l'exposant modulaire demande au plus $2k$ multiplications et calculs du reste. On doit donc effectuer $O(k)$ opérations dont le coût est $O(k^2)$ ce qui donne $O(k^3)$.

La borne $O(k^3)$ est une *borne supérieure*. En effet, on peut faire un peu mieux. Par exemple, avec l'algorithme de Karatsuba on peut multiplier deux nombres de k chiffres en $O(k^{1,59})$, au lieu de $O(k^2)$. Par ailleurs, l'algorithme présenté est *pratique* ; il est couramment utilisé dans les applications cryptographiques avec $k \approx 10^3$.

Exercice 3 *Programmez l'algorithme des carrés itérés.*

Chapitre 7

Tableaux

En mathématiques, un *vecteur* de dimension n sur un domaine D est un élément de D^n. Le *tableau* est la structure de données qui correspond au vecteur. Un vecteur de vecteurs est une matrice. De la même façon il est possible de représenter une structure de données à deux dimensions en déclarant un tableau de tableaux.

7.1 Déclaration et manipulation de tableaux

Pour *déclarer un tableau* x de type T et de dimension n on écrit en C :

$$\mathsf{T}\ \mathsf{x}[\mathsf{n}]; \tag{7.1}$$

Ceci a l'effet de réserver un segment de mémoire suffisant pour contenir n données de type T et d'associer l'adresse de base du segment au nom x. On notera que le nom d'un tableau est associé à une adresse et non pas à son contenu.

Dans une déclaration comme (7.1), le contenu du tableau x n'est pas défini. Comme pour les variables de type primitif, il est une erreur de lire un tableau avant de l'avoir défini. En C, il est aussi possible de déclarer un tableau et en même temps de l'initialiser. Par exemple,

$$\mathsf{int}\ \mathsf{a}[10] = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\};$$

déclare un tableau a avec 10 cellules et initialise la i-ème cellule avec la valeur i pour $i = 0, 1, \ldots, 9$.

On peut *écrire* ou *lire* le i-ème élément du tableau en utilisant la notation x[i] à condition que $0 \leq \mathsf{i} \leq (n-1)$. C'est au programmeur de respecter les bornes. En particulier, on notera qu'on commence à compter de 0 et que donc un accès à x[n] produit une erreur.

Exemple 21 *On met en garde le lecteur sur le fait que ce type d'erreur peut passer inaperçu et provoquer un comportement bizarre du programme. Par exemple, avec le compilateur* gcc *le programme ci-dessous compile et imprime* 10.

```
int main() {
  int a[4];
  int b[4];
  b[0]=10;
  printf("%d\n",a[4]);
  return 0;}
```

Exemple 22 *Voici un programme qui lit le degré et les coefficients d'un polynôme et évalue le polynôme dans un point. Par opposition au programme considéré dans la section 5.1, on utilise maintenant un tableau pour mémoriser tous les coefficients. Il est donc possible de lire complètement les données en entrée avant d'évaluer le polynôme. On laisse au lecteur le soin de modifier le programme ci-dessous de façon à utiliser la règle de Horner.*

```
int main() {
  int n;
  double x;
  printf("Entrez degré polynôme\n");
  scanf("%d",&n);
  printf("Entrez  point\n");
  scanf("%lf",&x);
  double a[n+1];
  int i;
  for (i=0;i<=n;i++){
     printf("Entrez coefficient %d\n", i);
     scanf("%lf",&a[i]);}
  double s=a[0];
  double y=1;
  for (i=1;i<=n;i++){
     y=x*y; s=a[i]*y+s;}
  printf("La valeur du polynôme est : %lf\n",s);
  return 0;}
```

7.2 Passage de tableaux en argument

Une fonction peut compter parmi ses *arguments* une variable de type tableau. Par exemple :

$$\mathsf{void\ f(int\ x[])\{\ x[0] = 3;\}}$$

Comme on l'a déjà remarqué, la *valeur* d'une variable de type tableau est une *adresse de mémoire* (et non pas le contenu du tableau). Par exemple, si on appelle la fonction f comme dans :

$$\mathsf{int\ y[3];\ f(y);}$$

on va modifier le tableau de l'appelant (y).

Exemple 23 *Considérons un petit exemple qui permet de comparer les variables de type tableau aux variables de type primitif. Dans* (7)*, on appelle* f *en lui passant l'adresse du tableau* a *et la valeur de la variable* x *de la fonction* main*. Dans* (2)*, la fonction* f *incrémente la variable* x *de* f *et l'élément* a[0] *du tableau* a *de la fonction* main*. Dans* (8)*, on imprime* 0 *car la variable* x *du* main *n'a pas été modifiée alors que dans* (9) *on imprime* 1*.*

```
void f (int a[], int x){              //(1)
  x=x+1;                              //(2)
  a[0]=a[0]+1;                        //(3)
int main(){                           //(4)
  int x=0;                            //(5)
  int a[10]={0,1,2,3,4,5,6,7,8,9};    //(6)
  f(a,x);                             //(7)
  printf("x=%d\n",x);                 //(8)
  printf("a[0]=%d\n",a[0]);           //(9)
  return 0;}                          //(10)
```

Remarque 6 *On peut se demander pourquoi on passe les variables de type primitif par valeur et les variables de type tableau par adresse. La raison est que les tableaux peuvent occuper beaucoup de mémoire et qu'autant que possible il est préférable de les partager plutôt que de les dupliquer. En cas de nécessité, il n'est pas compliqué de dupliquer un tableau : il suffit de déclarer un tableau dans la fonction appelée et d'y recopier le tableau dont la fonction appelante a fournit l'adresse. Dans le cas de la fonction* f *de l'exemple 23 ci-dessus, on aura par exemple :*

```
void f (int a[], int x){
   x=x+1;
   int b[10];
   int i;
   for (i=0;i<10;i++){b[i]=a[i];}
   b[0]=b[0]+1;}
```

Exemple 24 *On souhaite écrire une fonction qui prend en argument un tableau et sa taille et imprime le minimum et le maximum du tableau. Une solution possible est la suivante.*

```
void minmax (int a[], int n){
  int min, max, i;
  min=a[0];
  max=a[0];
  for (i=1;i<n;i++){if (a[i]<min){min=a[i];}
                    else {if (max<a[i]){max=a[i];}}}
  printf("min=%d\n",min);
  printf("max=%d\n",max);
return;}
```

En suivant la discussion ci-dessus, on notera que la fonction minmax *utilise le tableau dont l'adresse lui est communiquée par la fonction appelante.*

Dans le pire des cas, la fonction minmax *effectue* $2 \cdot (n-1)$ *comparaison (trouvez un tel cas !). Il est possible d'utiliser un autre algorithme qui lit les éléments du tableau par couple et les compare au* min *et au* max*. Vérifiez qu'on peut effectuer cette opération avec au plus* 3 *comparaisons et dérivez un algorithme qui effectue* $\frac{3}{2}n$ *comparaisons dans le pire des cas.*

7.3 Génération aléatoire de nombres

Il est très utile de générer de façon automatique et aléatoire les entrées d'un programme. Par ailleurs, certains programmes dits *probabilistes* ont besoin de nombres aléatoires pendant le calcul. En pratique, tout langage de programmation dispose d'un générateur de nombres (plus ou moins) aléatoires.

En C, la bibliothèque ⟨stdlib.h⟩ contient une fonction rand() qui génère un nombre "aléatoire" compris entre 0 et RAND_MAX (≥ 32767). En pratique, si $n << \text{RAND\_MAX}$ et on cherche un nombre "aléatoire" dans l'intervalle $[0, n-1]$, on calcule rand()%n.

Techniquement la fonction rand est basée sur une *congruence linéaire* et génère *toujours la même suite* (SIC). Si l'on veut changer la suite générée il faut initialiser un *germe* en exécutant par exemple la commande :

```
srand((unsigned)(time(NULL)));
```

qui va faire dépendre la suite du *temps courant* (time est une fonction de la bibliothèque ⟨time.h⟩).

La qualité des générateurs aléatoires des langages de programmation est très variable. En particulier, le générateur du langage C qu'on vient de décrire rend service pour le *test* ou la *simulation* mais il n'est *pas du tout* adapté aux applications cryptographiques.

7.4 Primalité et factorisation

Soit $n \geq 2$. Si n n'est pas premier alors il y a un premier p tel que $p \mid n$ et $p \leq \sqrt{n}$. Donc tout nombre n qui *n'est pas premier* s'écrit comme :

$$n = i \cdot j$$

où : $2 \leq i \leq \sqrt{n}$ et $i \leq j \leq n/i$. La liste des *premiers inférieurs à un* n donné peut être générée avec un algorithme ancien connu comme *crible d'Ératosthène* :

$P[i] =$ true pour $i = 2, \ldots, n$

pour $i = 2, \ldots, \lfloor\sqrt{n}\rfloor$
  pour $j = i, \ldots, n/i$
    $P[i \cdot j] =$ false

Exemple 25 *Pour* $n = 15$*, on obtient :*

$i\backslash j$	2	3	4	5	6	7
2	4	6	8	10	12	14
3		9	12	15		

Exercice 4 *Estimez en fonction de* n *le nombre de fois qu'on exécute l'affectation* $P[i \cdot j] =$ false*. Soit* $\pi(n)$ *la cardinalité des nombres premiers inférieurs à* n*. On sait que* $\pi(n) \approx n/\ln(n)$*. Comparez votre estimation avec la cardinalité des nombres composés inférieurs à* n *(à savoir* $n - \pi(n)$*).*

Exemple 26 *Voici une fonction* filtre *qui prend en entrée un tableau de* short *de* $n + 1$ *éléments et marque avec* 1 *les nombres premiers et avec* 0 *les autres.*

```
void filtre (short f[], int n){
  int i,j;
  int r =(int)(sqrt(n));
    for (i=2;i<=n;i++){f[i]=1;}
    for (i=2;i<=r;i++)
      {for (j=i; j<= n/i; j++){f[i*j]=0;}}}
```

On a donc une méthode pour générer un segment initial des nombres premiers. Considérons maintenant le problème de savoir si un nombre n est premier. Par exemple, prenons $n = 15413$. On calcule,

$$\lfloor\sqrt{15413}\rfloor = 124 \ .$$

Avec le crible d'Ératosthène, on peut calculer les premiers inférieurs à 124 :

2	3	5	7	11	13	17	19	23	
29	31	37	41	43	47	53	59	61	67
71	73	79	83	97	101	103	107	109	113

Le nombre n est premier si et seulement si aucun de ces nombres divise n. On a donc une méthode qu'on appelle *division par essai* pour savoir si un nombre n est premier. On sait qu'il y a environ $m/log(m)$ nombres premiers inférieurs à m. La méthode de division par essai demande donc environ $\sqrt{n}/log(\sqrt{n})$ divisions ce qui n'est pas très efficace (mais on connaît plusieurs algorithmes efficaces pour savoir si un nombre est premier).

On rappelle que tout nombre $n \geq 2$ admet une factorisation unique comme produit de nombres premiers. On peut *itérer* la division par essai pour trouver une factorisation complète :

1. On trouve p_1 tel que p_1 divise n.
2. Si $n' = n/p_1$ est premier on a trouvé la factorisation et autrement on itère sur le nombre n'.

Par exemple, pour $n = 15400$ on trouve :

$$n = 2 \cdot 2 \cdot 2 \cdot 5 \cdot 5 \cdot 7 \cdot 11 \ .$$

On a donc un algorithme pour *factoriser un nombre*. Voici une mise en oeuvre possible de l'algorithme de factorisation où l'on suppose que la fonction imprimer_factorisation reçoit en argument un tableau de short premier où les nombres premiers ont été marqués en utilisant la fonction filtre du crible d'Ératosthène. Le tableau premier est initialisé une fois et réutilisé dans tous les appels de la fonction imprimer_factorisation.

```
void imprimer_factorisation(short premier[], int n){
  int m = (int)(sqrt(n));
  int i;
  for (i=2; i<=m; i++){
    if (premier[i] && (n%i==0)){printf("%d ",i); break;}}
  if (i>m) {printf("%d",n); return;}
  else imprimer_factorisation(premier, n/i);}
```

7.5 Tableaux à plusieurs dimensions

On peut déclarer des tableaux de tableaux de tableaux... Par exemple :

```
int m=3; int n=5; int p=7;
int a[m][n][p];
a[0][4][5]=1;
```

En C, on peut omettre seulement la dimension du premier tableau. Ainsi la première déclaration qui suit est admise mais la deuxième ne l'est pas.

```
void produit (int a[][5], int b[5][10]){...} // admise
void produit (int a[][], int b[][]){...}     // pas admise
```

En pratique, une bonne méthode consiste à passer la dimension du tableau en paramètre. Par exemple, une fonction C qui calcule le produit de deux matrices a et b de dimension $n \times n$ et écrit le résultat dans la matrice c pourrait être la suivante :

```
void multiplier (int n, int a[n][n], int b[n][n], int c[n][n]){
  int i,j,k;
  for (i=0;i<n;i++){
```

```
    for (j=0;j<n;j++){
    c[i][j]=0;
      for (k=0;k<n;k++)
      {c[i][j]=c[i][j]+a[i][k]*b[k][j];}
}}}
```

On notera que cette fonction contient 3 boucles for imbriquées et qu'elle effectue $O(n^3)$ multiplications.

Exemple 27 *On souhaite imprimer les éléments d'un tableau de tableaux* a *de type* int a[m][n] *avec la contrainte que l'élément* a[i][j] *doit être imprimé avant l'élément* a[k][l] *si* $i+j < k+l$. *Par exemple, si* a *est comme suit :*

$$\begin{array}{cccc} 4 & 5 & 7 & 3 \\ 3 & 1 & 9 & 10 \\ 8 & 2 & 1 & 4 \end{array}$$

en supposant a[0][0] $= 8$, a[0][1] $= 2$, ..., *une impression qui respecte la contrainte énoncée (il y en a d'autres) est :* $8, 2, 3, 4, 1, 1, 9, 5, 4, 10, 7, 3$ *; on imprime donc 'par diagonale'.*

Pour traiter ce problème, on peut remarquer que la valeur $k = i + j$ *varie entre* 0 *et* $(m + n - 2)$. *Pour un* k *fixé, la première coordonnée* i *varie entre* 0 *et* $min(k, m - 1)$, *alors que la deuxième est déterminée par* $k - i$. *On a donc l'algorithme suivant :*

$$\begin{array}{l} \textit{pour } k = 0, \ldots, (m + n - 2) \\ \quad \textit{pour } i = 0, \ldots, min(k, m - 1) \\ \quad \textit{si } (k - i) \leq (n - 1) \textit{ imprime } a[i][k - i] \end{array}$$

Pour le a *en question, avec* $m = 3$ *et* $n = 4$, *on imprime :*

$$8, 2, 3, 1, 1, 4, 4, 9, 5, 10, 7, 3$$

La fonction C *suivante met en oeuvre l'algorithme.*

```
void imprime_diag(int m, int n, int a[m][n]){
  int k; int i;
  for (k=0; k<=(m+n-2); k++){
    int min=(m-1); if (k<min){min=k;}
    for (i=0; i<=min;i++){
      if ((k-i)<=(n-1)){printf("%d ", a[i][k-i]);}}}}
```

Chapitre 8

Tri et permutations

Le tri d'une suite finie d'éléments selon un certain ordre est une *opération fondamentale*. En faisant l'hypothèse que la suite est représentée par un tableau, on présente et on analyse la complexité de 3 algorithmes de tri. Une *permutation* sur un ensemble fini admet aussi une représentation naturelle en tant que tableau. Dans ce contexte, on étudie la composition, l'inversion, la génération aléatoire et l'énumération de permutations.

8.1 Tri à bulles et par insértion

Dans cette section, on présente 2 algorithmes de tri :

1. Tri à bulles (*bubble sort*, en anglais).
2. Tri par insertion (*insertion sort*, en anglais).

Dans la prochaine section on discutera le tri par fusion (*merge sort*, en anglais). D'autres algorithmes de tri existent dont le tri rapide ou par partition (*quicksort*, en anglais) et le tri par tas (*heapsort* en anglais) ont des performances proches du tri par fusion.

Par défaut, on fait l'hypothèse que la suite est mémorisée dans un *tableau*. Alternativement, on peut aussi envisager de représenter la suite par une *liste* (une structure de données qui sera discutée dans la section 12.1) et dans ce cas il peut être nécessaire de reconsidérer certains détails.

Tri à bulles

On peut écrire une fonction bulles(i) qui compare les $i-1$ couples aux positions :

$$(1,2),(2,3),(3,4),\cdots(i-1,i)$$

et les permute si elles ne sont pas en ordre croissant. Le coût est linéaire en $i-1$. A la fin de l'exécution de bulles(i) on est sûr que l'élément le plus grand se trouve à la position i. Pour trier, il suffit donc d'exécuter :

$$\mathsf{bulles(n)}, \mathsf{bulles(n-1)}, \ldots, \mathsf{bulles(2)} \ ,$$

pour un coût qui est :

$$\Sigma_{i=1,\ldots,n-1} i = \frac{n(n-1)}{2} \ .$$

Soit $O(n^2)$. Voici un exemple de fonction C qui prend en argument un tableau et sa taille et le trie selon le principe décrit ci dessus.

```
void tri_bulles(int a[], int n){
  int aux, i,j;
  for (i=(n-1);i>=1;i--){
    for (j=0;j<i;j++){
      if (a[j]>a[j+1]){aux=a[j]; a[j]=a[j+1];a[j+1]=aux; }}}
}
```

Tri par insertion

On peut écrire une fonction insert(i) qui, en supposant les éléments aux positions $i+1, \ldots, n$ en ordre croissant, va insérer l'élément en position i à la bonne place. Le coût est linéaire en $n-i$.

Pour trier il suffit donc d'exécuter :

$$\mathsf{insert}(n-1), \mathsf{insert}(n-2), \ldots, \mathsf{insert}(1) \ ,$$

pour un coût qui est :

$$1 + 2 + \cdots + (n-1) = \frac{n(n-1)}{2} \ .$$

Soit encore $O(n^2)$. Une mise en oeuvre de l'algorithme est ci-dessous. Le lecteur est invité à analyser en détail la fonction ins qui effectue l'insertion d'un élément dans un tableau.

```
void ins(int a[], int n, int j){
  int k=a[j];
  int i=j+1;
  while (i<n && k>a[i]){a[i-1]=a[i];i++;}
  a[i-1]=k;}
void tri_ins(int a[], int n){
  int j;
  for (j=n-2;j>=0;j--){ins(a,n,j);}}
```

8.2 Tri par fusion

On a examiné deux algorithmes de tri en $O(n^2)$. Peut-on faire mieux ? On va appliquer une stratégie diviser pour régner dont on a déjà vu un exemple dans le cadre de la recherche dichotomique. Une façon d'appliquer cette stratégie donne lieu au *tri par fusion* (*mergesort*, en anglais) qui a été proposé par Von Neumann autour de 1945.

— Si le tableau a *taille* 1, le tableau est trié.
— Sinon, on sépare le tableau en *deux parties égales* et on les trie.
— Ensuite on fait une *fusion* des deux tableaux triés.

Le coeur de l'algorithme est la *fonction de fusion* de deux ensembles ordonnés. L'idée naturelle est de *parcourir en parallèle* les deux ensembles par ordre croissant (par exemple) et de sélectionner à chaque pas le *minimum* entre les deux. Si l'on représente les ensembles comme des *listes* (section 12.1) il est facile de construire la liste fusion *en place* (sans allocation de mémoire). Cependant, si l'on représente les ensembles comme des *tableaux* il est *beaucoup plus compliqué* (mais possible) de travailler en place. Une solution simple, consiste à utiliser un *tableau auxiliaire*. Avant de commencer la fusion on *copie* les deux tableaux ordonnés dans

le tableau auxiliaire et ensuite on écrit la solution dans le tableau de départ. Une mise en oeuvre en C pourrait être la suivante.

```
void fusion(int t[], int i, int j, int k){
  assert((i<=j)&&(j<k));
  int aux[k+1];
  int p;
  for(p=i;p<=k;p++){aux[p]=t[p];}
  p=i;
  int q=j+1;
  int r=i;
   while (r<=k){
     if ((p<=j) && (aux[p]<=aux[q])){t[r]=aux[p];p++;r++; continue;}
     if ((q<=k) && (aux[p]>aux[q])) {t[r]=aux[q];q++;r++; continue;}
     if (p>j){t[r]=aux[q];q++;r++; continue;}
     if (q>k){t[r]=aux[p];p++;r++; continue;}}}
void trifusion(int t[], int i, int j){
  assert(i<=j);
  if (i<j){
     int m=(i+j)/2;
     trifusion(t,i,m);
     trifusion(t,m+1,j);
     fusion(t,i,m,j);}}
```

Complexité du tri par fusion

Quelle est la complexité asymptotique du tri par fusion ? Soit $C(n)$ une borne supérieure au temps nécessaire pour trier par fusion un tableau de taille n. On pose la *récurrence* suivante :

$$\begin{aligned} C(1) &= 1 \\ C(n) &= 2 \cdot C(n/2) + n \end{aligned}$$

qui veut dire qu'un problème de taille n génère deux sous-problèmes de taille $n/2$ et effectue un travail de combinaison (la fusion) de complexité proportionnelle à n.

Pour simplifier le raisonnement, supposons que $n = 2^k$. On a :

$$\begin{array}{lll} 2^0 & \text{problèmes de taille} & 2^k \\ 2^1 & \text{problèmes de taille} & 2^{k-1} \\ \cdots & & \\ 2^k & \text{problèmes de taille} & 2^0 \end{array}$$

La somme du travail de *combinaison* à chaque niveau est *constant* et égal à n. Comme on a $k = \log_2(n)$ niveaux, le travail total est $O(n \log n)$.

Remarque 7 *La* solution de relations de récurrence *est un sujet très vaste (c'est la version discrète des équations différentielles !). Par exemple, on sait traiter toutes les récurrences de la forme :*

$$C(n) = a \cdot C(n/b) + O(n^c) \ .$$

Exercice 5 *En* C*, on peut utiliser la fonction* clock() *de la librairie* time.h *pour estimer le temps d'exécution d'un programme comme dans l'exemple suivant :*

```
clock_t begin = clock();
/* tri fusion */
clock_t end = clock();
double time_spent = (double)(end - begin) / CLOCKS_PER_SEC;
```

Programmez une fonction qui imprime une estimation du temps moyen d'exécution du tri par fusion sur des entrées de taille 2^i pour $i = 6, \ldots, m$. Pour estimer le temps moyen, on fera tourner la fonction sur n entrées de taille 2^i générées avec probabilités uniforme (la section 8.3 explique comment faire). A titre indicatif, on peut prendre $n = 2^6$ et $m \leq 16$ (m à adapter en fonction des performances de votre machine). Comparez le temps d'exécution du tri par fusion avec celui du tri à bulles ou du tri par insértion.

Calcul du nombre d'inversions

On va étudier une application remarquable de la fonction de fusion. On dispose d'un tableau t *non-ordonné* de n éléments. Une *inversion* est un couple (i, j) tel que :

$$1 \leq i < j \leq n \qquad t[i] > t[j]$$

On souhaite calculer le *nombre d'inversions* dans t qui est un nombre compris entre 0 et $\frac{n(n-1)}{2}$.

Exercice 6 *Proposez un algorithme en $O(n^2)$ pour calculer le nombre d'inversions. Programmez une fonction* C *qui correspond à l'algorithme d'en tête :* `int inversions(int n, int t[n])`.

Comme il y a $O(n^2)$ inversions, tout algorithme qui compte les inversions une par une prendra dans le pire des cas $O(n^2)$. Pour améliorer la complexité il nous faut donc une méthode pour compter plusieurs inversions en même temps. Il se trouve qu'il est possible de modifier la fonction `fusion` ci-dessus de façon telle que l'algorithme de tri par fusion qui l'utilise calcule le nombre d'inversions. Considérons les 4 cas de la boucle `while` de la fonction `fusion` (section 8.1). Dans le deuxième cas, on inverse l'élément `a[q]` avec tous les éléments `a[p]`, ..., `a[j]` et il faut donc ajouter au compteur $j - p + 1$ inversions. Il n'y a pas d'inversion dans les autres 3 cas. En supposant que l'addition d'entiers 32 bits se fait en $O(1)$, on ne change pas la complexité de l'algorithme du tri par fusion. On a donc toujours $O(nlog(n))$. Voici une application de la méthode à la séquence $7, 6, 5, 4, 3, 2, 1, 0$:

Séquences à fusionner	Nombre inversions
$76, 54, 32, 10$	$4 \cdot 1 = 4$
$6745, 2301$	$2 \cdot 4 = 8$
54670123	$1 \cdot 16 = 16$
	Total $28 = \frac{8 \cdot 7}{2}$

8.3 Permutations

Une permutation sur l'ensemble $\{0, 1, \ldots, n-1\}$ est une fonction bijective $p : \{0, 1, \ldots, n-1\} \rightarrow \{0, 1, \ldots, n-1\}$. On représente une telle permutation par un tableau d'entiers de taille n qui contient les entiers $\{0, 1, \ldots, n-1\}$ exactement une fois. Soit id la permutation identité et $\circ$ la composition de permutations. L'ensemble des permutations sur l'ensemble $\{0, 1, \ldots, n-1\}$ est un groupe commutatif. En particulier, chaque permutation admet une

permutation inverse par rapport à la composition. Un point fixe d'une permutation p est un élément $i \in \{0, \ldots, n-1\}$ tel que $p(i) = i$.

Exercice 7 *Programmez une fonction* C *d'en tête* void comp(int n, int r[n], int p[n], int q[n]) *qui prend en entrée deux permutations (représentées par les tableaux* p *et* q*) et écrit dans le premier tableau* r *la représentation de la permutation composition '$p \circ q$'. Précisez la complexité asymptotique en temps de votre fonction.*

Exercice 8 *Programmez une fonction* C *d'en tête* void inv(int n, int p[n], int q[n]) *qui prend en entrée une permutation (représentée par le tableau* p*) et écrit dans le tableau* q *la représentation de la permutation inverse de* p*. Précisez la complexité asymptotique en temps de votre fonction.*

Exercice 9 *Programmez une fonction* C *d'en tête* int nbpointfixe(int n, int p[n]) *qui prend en entrée une permutation (représentée par le tableau* p*) et retourne le nombre de points fixes de* p*. Précisez la complexité asymptotique en temps de votre fonction.*

Permutations aléatoires

On rencontre souvent le problème suivant : à partir d'un générateur aléatoire de nombres, il faut concevoir un programme qui génère des structures avec une certaine distribution. Ici on considère le problème de générer des permutations avec une distribution uniforme.

Premier essai Considérez la fonction `permall` suivante qui génère une permutation d'un tableau. On suppose que rand()%n nous donne un entier dans $[0, n-1]$ avec une *distribution uniforme.*

```
void permall (int t[],int n){
    int i; for (i=0;i<n;i++){
    int j=(rand()%n);
    int temp=t[i]; t[i]=t[j]; t[j]=temp;}}
```

La fonction `permall` génère-t-elle une permutation avec une distribution uniforme ? La réponse est négative !

Analyse

- Chaque chemin d'exécution demande la génération de n nombres entiers dans $[0, n-1]$.
- On a donc n^n chemins possibles et chaque chemin a probabilité $\frac{1}{n^n}$.
- Comme on a $n!$ permutations, si la distribution était uniforme on devrait avoir $\frac{1}{n!} = \frac{k}{n^n}$ pour $k \in \mathbf{N}$. Soit : $n^n = kn!$
- Contradiction ! Par exemple, en prenant $n = 3$.

Deuxième essai Considérez la fonction `permplace` suivante :

```
void permplace (int t[],int n){
    int i; for (i=0;i<n;i++){
    int j=(rand()%(n-i))+i;
    int temp=t[i]; t[i]=t[j]; t[j]=temp;}}
```

Analyse Une *k-séquence* d'un ensemble X de cardinalité n ($n \geq k$) est une liste de k-éléments différents de X. Il y a $\frac{n!}{(n-k)!}$ k-séquences d'un ensemble de n éléments, car :

$$n \cdots n-k+1 = \binom{n}{k} k! = \frac{n!}{(n-k)!} \ .$$

On suppose que les éléments du tableau `t` sont tous différents. On montre par *récurrence* sur $k = 0, 1, \ldots, n$ que la propriété suivante est satisfaite à la k-ème itération de la boucle `for`.

Proposition 1 *Pour toute k-séquence S de l'ensemble* $\{\mathtt{t[0]}, \ldots, \mathtt{t[n-1]}\}$ *on a* $\mathtt{t[0]} \cdots \mathtt{t[k-1]} = S$ *avec probabilité* $\frac{(n-k)!}{n!}$.

PREUVE. Pour $k = 0$, S est la séquence vide et $\mathtt{t[0]} \cdots \mathtt{t[k-1]}$ est aussi la séquence vide. Par ailleurs $\frac{(n-0)!}{n!} = 1$.

On suppose la propriété vraie pour $k < n-1$. Soit $S = S'v$ une $(k+1)$-séquence. On sait que $\mathtt{t[0]} \cdots \mathtt{t[k-1]} = S'$ avec probabilité $\frac{(n-k)!}{n!}$. Par ailleurs, on a $\mathtt{t[k]} = v$ avec probabilité $\frac{1}{n-k}$ puisque l'élément est choisi parmi les $n-k$ qui ne sont pas déjà dans S'. Donc la probabilité que $\mathtt{t[0]} \cdots \mathtt{t[k]} = S$ est :

$$\frac{(n-k)!}{n!} \frac{1}{n-k} = \frac{(n-(k+1))!}{n!} \ .$$

On peut donc conclure que la fonction `permplace` génère une permutation du tableau avec une probabilité uniforme : chaque n-séquence est générée avec probabilité $\frac{1}{n!}$. □

Exercice 10 *Adaptez l'algorithme pour générer avec probabilité uniforme un échantillon de k éléments choisis parmi n éléments.*

Énumérer les permutations

On considère maintenant le problème de concevoir un programme qui énumère toutes les permutations sur $\{0, 1, \ldots, n-1\}$. Par exemple, pour $n = 3$ le programme pourrait imprimer :

```
0 1 2
0 2 1
1 0 2
1 2 0
2 0 1
2 1 0
```

Pour préparer le terrain on peut d'abord considérer le problème suivant : énumérer toutes les *fonctions* sur $\{0, \ldots, n-1\}$. Une fonction sur $\{0, 1, \ldots, n-1\}$ peut être représentée par un tableau avec n éléments qui varient sur $\{0, \ldots, n-1\}$. Énumérer les fonctions revient alors à énumérer de tels tableaux. On peut suivre l'algorithme suivant : au pas i on écrit dans la cellule i du tableau la valeur j pour $j = 0, 1, \ldots, n-1$. Pour chaque j, on vérifie si $i = n-1$. Si c'est le cas on imprime le tableau et sinon on incrémente i et on recommence. Voici un codage (à compléter) de cet algorithme en C.

```
void fonct(int g[], int i, int n){
  int j;
  for (j=0;j<n;j++){
    g[i]=j;
```

```
    if (i==(n-1)){ /* imprimer fonction */ }
    else {fonct(g,i+1,n);}}}
int main() {
  int n;
  /* lire n */
  int g[n];
  fonct(g,0,n);
  return 0;}
```

Exercice 11 *Programmez une fonction* C *qui vérifie si une fonction sur* $\{0, 1, \ldots, n-1\}$ *est une permutation. Modifiez le programme ci-dessus pour qu'il imprime seulement les permutations.*

Le programme pour énumérer les permutations dérivé de l'exercice 11 n'est pas particulièrement efficace car il énumère toutes les fonctions sur $\{0, 1, \ldots, n-1\}$ pour ensuite imprimer seulement les permutations. En général, on aura n^n fonctions et seulement $n!$ permutations. Par exemple, pour $n = 7$, on a $7^7 = 823543 >> 5040 = 7!$.

Une approche plus efficace consiste donc à détecter aussi tôt que possible les fonctions partiellement spécifiées qui n'ont aucune chance de devenir des permutations. Une condition nécessaire et suffisante pour qu'une fonction partiellement spécifiée sur $\{0, 1, \ldots, n-1\}$ puisse devenir une permutation est qu'il n'y ait pas un élément répété dans l'image de la fonction (pour construire une permutation il faut utiliser les éléments dans $\{0, 1, \ldots, n-1\}$ exactement une fois). On va donc introduire un deuxième tableau dans lequel on va se souvenir des éléments déjà utilisés dans la construction d'une permutation. Une programmation possible est la suivante.

```
void perm(int p[], short f[], int i, int n){
  int j;
  for (j=0;j<n;j++)
  {if (f[j]) {f[j]=0;     /* j n'est plus disponible */
              p[i]=j;
      if (i==(n-1)){/* imprimer permutation */}
              else {perm(p,f,i+1,n);}
              f[j]=1;     /* j à nouveau disponible */ }}}
int main() {
  int n;
  /* lire n */
  int g[n];
  short f[n];
  int j; for (j=0;j<n;j++){f[j]=1;}  /* tous j disponibles */
  perm(p,f,0,n);
  return 0;}
```

Remarque 8 *Comme pour le problème du déplacement de la tour d'Hanoi (section 5.2) et du tri par fusion (section 8.1), la simplicité de ce programme repose sur l'utilisation d'une fonction récursive. Le lecteur est invité à modifier le programme pour qu'il trace tous les appels à la fonction* perm.

Chapitre 9

Preuve et test de programmes

Dans ce chapitre on introduit la problématique de la *preuve* et du *test* de programmes. On évoquera quelques idées générales sans aller dans les détails. En effet, il faudrait un cours entier pour traiter le sujet de façon systématique.

9.1 Preuve d'algorithmes

On considère qu'un algorithme est une description mathématique d'un procédé de calcul et qu'un programme est la mise-en-oeuvre de ce procédé dans un langage de programmation. On s'attend donc à que la preuve d'un algorithme soit plus facile que la preuve du programme correspondant car il faut se soucier de moins de détails. Notamment, on a pas besoin d'un modèle formel de l'exécution du programme. En particulier, la pratique de la preuve d'algorithmes et de programmes programmes passe par l'étude d'un certains nombre de méthodes de déduction automatique et par l'apprentissage d'au moins un assistant de preuve.

On s'intéresse d'abord à la preuve d'algorithmes dont voici les ingrédients principaux.

— Un modèle abstrait des *états* du calcul dont certains sont identifiées comme états terminaux.

— Des *règles de calcul* pour transformer les états.

— Une preuve que si on part d'un état avec une certaine propriété (la *pré-condition*) et on itère les règles de calcul alors si on arrive à un état terminal on satisfait une autre propriété (la *post-condition*). Cette partie de la preuve s'articule autour de la définition d'un *invariant*. En première approximation, un invariant est un ensemble d'états dont on ne peut pas sortir en appliquant les règles de calcul.

— Une preuve qu'à partir d'un état qui satisfait la pré-condition on arrivera bien à un état terminal (l'algorithme *termine*). Cette partie de la preuve se base sur l'interprétation du calcul dans un *ordre bien fondé*. Un ordre bien fondé est un ordre dans lequel toute suite strictement décroissante est finie.

On aborde les différentes notions évoquées (états, règles de calcul, invariant, interprétation dans un ordre bien fondé) dans le cadre d'un exemple concret : le tri par insertion.

Modèle des données On fixe un ensemble Σ avec un ordre total. On modélise la suite des valeurs à trier comme un mot $w \in \Sigma^*$. On écrit ϵ pour le mot vide, $w \cdot w'$ pour la concaténation de mots et $|w|$ pour la longueur d'un mot.

Spécification On voit le tri par insertion comme une fonction isort sur les mots. Cette fonction doit satisfaire la propriété suivante : pour toute séquence $w \in \Sigma^*$, $\mathsf{isort}(w)$ est

une *séquence croissante* et une *permutation* de w.

Sous-spécification Pour décrire le calcul de la fonction isort on introduit une fonction d'insertion :

$$\mathsf{ins} : \Sigma \times \Sigma^* \to \Sigma^* \ .$$

Cette fonction doit satisfaire la propriété suivante : pour tout $a \in \Sigma$, pour toute *séquence croissante* w, $\mathsf{ins}(a, w)$ est une *séquence croissante* et une *permutation* de $a \cdot w$.

Algorithme pour ins On décrit un algorithme pour l'insertion par récurrence sur la longueur de la séquence w en entrée :

$$\begin{array}{ll} \mathsf{ins}(a, \epsilon) & = a \\ \mathsf{ins}(a, b \cdot w) & = \begin{cases} a \cdot b \cdot w & \text{si } a \leq b \\ b \cdot \mathsf{ins}(a, w) & \text{autrement.} \end{cases} \end{array}$$

Algorithme pour isort Dans le même style, on décrit un algorithme pour le tri :

$$\begin{array}{ll} \mathsf{isort}(\epsilon) & = \epsilon \\ \mathsf{isort}(a \cdot w) & = \mathsf{ins}(a, \mathsf{isort}(w)) \ . \end{array}$$

Ces définitions sont assez concrètes pour induire des règles de calcul. Par exemple, le tri du mot $3 \cdot 2 \cdot 1$ pourrait correspondre aux étapes de calcul suivantes :

$$\begin{array}{ll} \mathsf{isort}(3 \cdot 2 \cdot 1) & \to \mathsf{ins}(3, \mathsf{isort}(2 \cdot 1)) \\ \to \mathsf{ins}(3, \mathsf{ins}(2, \mathsf{isort}(1))) & \to \mathsf{ins}(3, \mathsf{ins}(2, \mathsf{ins}(1, \mathsf{isort}(\epsilon)))) \\ \to \mathsf{ins}(3, \mathsf{ins}(2, \mathsf{ins}(1, \epsilon))) & \to \mathsf{ins}(3, \mathsf{ins}(2, 1)) \\ \to \mathsf{ins}(3, 1 \cdot \mathsf{ins}(2, \epsilon)) & \to \mathsf{ins}(3, 1 \cdot 2) \\ \to 1 \cdot \mathsf{ins}(3, 2) & \to 1 \cdot 2 \cdot \mathsf{ins}(3, \epsilon) \\ \to 1 \cdot 2 \cdot 3 \ . & \end{array}$$

Le prédicat 'séquence croissante' On considère maintenant une définition formelle des prédicats évoqués dans la spécification. Le prédicat séquence croissante est les plus petit prédicat unaire sur les mots qui satisfait les conditions suivantes :

$$\frac{}{ord(\epsilon)} \qquad \frac{}{ord(a)} \qquad \frac{a \leq a' \quad ord(a' \cdot w)}{ord(a \cdot a' \cdot w)}$$

Le prédicat 'permutation' La formalisation de la relation de permutation n'est pas aussi directe. On peut définir :

— Une fonction $\mathsf{elim}(a, w)$ qui *élimine* la première occurrence de a dans la séquence w (si elle existe).

— Un prédicat *occurrence* $occ(a, w)$ qui vérifie si a est dans la séquence w.

— Une prédicat binaire *plongement* $pl(w, w')$ tel que :

$$\frac{}{pl(\epsilon, w)} \qquad \frac{pl(w, \mathsf{elim}(a, w')) \qquad occ(a, w')}{pl(a \cdot w, w')}$$

et enfin définir la permutation comme un plongement dans les deux sens :

$$perm(w, w') \equiv (pl(w, w') \wedge pl(w', w)) \ .$$

Remarque 9 *On notera qu'il est aussi facile de se tromper dans la description de l'algorithme que dans sa spécification. Dans notre cas, l'algorithme est comparable en taille et complexité à sa spécification. Par ailleurs, la façon de spécifier a un impact sur la preuve !*

Preuve de correction La preuve de correction d'un algorithme se décompose souvent en deux parties : une preuve de terminaison du calcul et une preuve qu'un certain prédicat est un invariant (est préservé) par le calcul. Dans notre cas, la preuve de terminaison est très simple et elle se résume à l'observation que les fonctions ins et isort sont bien définies par récurrence sur la taille du mot en entrée. On verra dans la section 9.2 des preuves de terminaison plus compliquées. Concernant la formalisation de l'invariant, on s'attend, entre autres, qu'à chaque état du calcul on manipule une permutation de la suite initiale d'éléments (voir l'exemple de calcul ci-dessus). Plus précisément, on peut montrer par récurrence sur $|w|$:

$$\forall\, w \in \Sigma^*, a \in \Sigma \ \ (\ ord(w) \text{ implique } (ord(\mathsf{ins}(a,w)) \text{ et } perm(a \cdot w, \mathsf{ins}(a,w))))) \ ) \ .$$

Ensuite, on dérive par récurrence sur $|w|$:

$$\forall\, w \in \Sigma^* \ \ (\ ord(\mathsf{isort}(w)) \text{ et } perm(w, \mathsf{isort}(w)) \ ) \ .$$

9.2 Terminaison

Dans ces notes de cours, les preuves de terminaison sont assez *directes*. Cependant, en général le problème de savoir si un programme termine est *indécidable* et il est aussi possible de construire des programmes simples dont la terminaison est un *problème ouvert*.

Exemple 28 *Voici un* problème difficile *connu comme fonction* 91 *de McCarthy. La fonction suivante termine-t-elle ?*

```
int f(int n){
    if (n > 100){return n - 10;}
        else {return f(f(n+11));}}
```

Exemple 29 *La fonction suivante, connue comme fonction de Collatz, termine-t-elle ? Il s'agit d'un* problème ouvert.

```
void collatz(int n){
  if (n>1){if (n%2==0){collatz(n/2);}
                else {collatz(3*n+1);}}}
```

Une stratégie générale pour prouver la terminaison d'un programme est d'interpréter ses états de calcul dans un ensemble bien fondé.

Définition 4 (ensemble bien fondé) *Un* ensemble bien fondé *(*well-founded *en anglais) est un couple* $(W, >)$ *où :*

1. W *est un* ensemble.
2. $>\subseteq W \times W$ *est une* relation transitive.
3. *Il n'existe* pas de séquence infinie $w_0 > w_1 > w_2 > \cdots$ *dans* W *(en particulier, pour tout* $w \in W$, $w \not> w$ *!)*

Exemple 30 *Voici des exemples d'ensembles bien fondés.*

— *L'ensemble* $\mathbf{N}$ *des* nombres naturels *avec l'ordre standard*
— *L'ensemble* $\mathbf{N} \cup \{+\infty\}$.
— *L'ensemble* $\mathbf{N} \times \mathbf{N}$ *avec l'*ordre produit.

— *L'ensemble des* formules *du calcul propositionnel ordonnées selon leur taille. Et des non-exemples.*

— *L'ensemble* $\mathbf{Z}$ *des nombres* entiers *avec l'ordre standard.*

— *L'ensemble des nombres* rationnels positifs *avec l'ordre standard.*

— *L'ensemble*

$$A = \bigcup \{\mathbf{N}^k \mid k \geq 1\} \ ,$$

avec un ordre $>$ *tel que :*

$$(y_1, \ldots, y_m) > (x_1, \ldots, x_n) \text{ ssi } \exists\ k \leq min(n, m)\ (x_1 = y_1, \ldots, x_{k-1} = y_{k-1}, y_k > x_k) \ .$$

Comment prouver la terminaison d'un programme ? Revenons au modèle d'exécution du programme. Un *état* décrit, à un niveau d'abstraction adapté, la configuration de la machine à un certain moment du calcul. Le *calcul* (déterministe) d'un programme à partir d'un état initial peut donc être vu comme une suite (éventuellement infinie si le programme boucle) d'états. Soit A l'*ensemble des états possibles* et pour $a, a' \in A$ écrivons $a \to a'$ si le programme va *avec un pas de calcul* de l'état a à l'état a'.

Une *condition suffisante (et nécessaire)* pour montrer la terminaison du programme est de trouver un *ordre bien fondé* $(W, >)$ et une *interprétation* $\mu : A \to W$ telle que :

$$a \to a' \text{ implique } \mu(a) > \mu(a') \ .$$

En effet, dans ce cas un *calcul infini* : $a_0 \to a_1 \to a_2 \cdots$, implique *une suite descendante infinie* ce qui est contradictoire avec l'hypothèse que $(W, >)$ est bien fondé : $\mu(a_0) > \mu(a_1) > \mu(a_2) > \cdots$

Exemple 31 *Considérons la terminaison de programmes* while *de la forme suivante (inspirée par la recherche dichotomique) :*

$$\begin{array}{l} \mathsf{while}(u > l + 1)\{ \\ r = (u + l)/2; \\ \mathsf{if}(b)\{u = r;\}\ \mathsf{else}\ \{l = r;\};\} \end{array}$$

Ici on suppose que les variables u, l, r *prennent comme valeurs des* nombres naturels *et que la* condition logique b *donne toujours un résultat et ne modifie pas* u, l, r.

Pour montrer la terminaison il suffit de montrer que la boucle while *est exécutée un nombre fini de fois. L'exécution du* corps de la boucle *dépend et affecte les variables* l, r, u. *Donc on peut supposer qu'un* état *est un* triplet de nombres naturels $(x, y, z) \in \mathbf{N}^3$, x, y, z *étant les valeurs des variables* l, r, u.

Si $z > (x + 1)$ *et selon la branche* then *ou* else *suivie, une itération de la boucle engendre les transformations suivantes :*

$$\begin{array}{lcll} (x, y, z) & \to & (x, (x + z)/2, (x + z)/2) & \textit{(branche then)} \\ (x, y, z) & \to & ((x + z)/2, (x + z)/2, z) & \textit{(branche else)} \end{array}$$

Prenons comme ordre bien fondé *les* nombres naturels *avec l'ordre standard et définissons :*

$$\mu(x, y, z) = (z - x) \ .$$

Il est un exercice *(facile) de vérifier que si* $z > (x+1) \geq 1$ *alors :*

$$\begin{array}{lcll} (z-x) & > & \mu(x,(x+z)/2,(x+z)/2) & = (x+z)/2 - x \\ (z-x) & > & \mu((x+z)/2,(x+z)/2,z) & = z-(x+z)/2\ . \end{array}$$

Donc si le programme bouclait on aurait une suite descendante infinie dans $\mathbf{N}$ *ce qui est impossible !*

Exercice 12 *Considérez la relation suivante sur* $\mathbf{N}\times\mathbf{N}$ *:*

$$(x,y) >_l (x',y') \text{ si } x > x' \text{ ou } (x = x' \text{ et } y > y')\ .$$

Montrez que :

1. *La relation* $>_l$ *est* transitive.
2. *L'ensemble* $\{(x,y) \mid (2,2) >_l (x,y)\}$ *est* infini.
3. *Néanmoins* $(\mathbf{N}\times\mathbf{N}, >_l)$ *est un ordre* bien fondé. [1]

Exercice 13 *Les programmes* while *suivants terminent-t-ils en supposant que les variables varient sur les nombres naturels positifs ?*

$$\mathsf{while}(m \neq n)\{\ \mathsf{if}(m > n)\{m = m - n;\}\ \mathsf{else}\ \{n = n - m;\}\ \}\ ,$$

$$\mathsf{while}(m \neq n)\{\ \mathsf{if}(m > n)\{m = m - n;\}\ \mathsf{else}\ \{h = m; m = n; n = h;\}\ \}\ .$$

9.3 Preuve de programmes

Est-ce raisonnable de considérer les fonctions isort et ins de la section 9.1 comme des programmes ? La réponse est positive si les mots sont un type primitif. Par exemple, voici les fonctions ins et isort dans un langage fonctionnel de la famille ML.

```
let rec ins a x = match x with
  [] -> [a];
| b::w -> if (a<=b) then a::b::w else b::(ins a w);;

let rec isort x = match x with
  []   -> []
| a::w -> ins a (isort w) ;;
```

Considérons maintenant la mise-en-oeuvre de l'algorithme de tri par insertion dans le langage C en supposant que les éléments à trier sont mémorisés dans un tableau partagé. Dans ce cas, chaque fonction spécifie une série de transformations qui modifient le tableau. Par exemple, voici les fonctions ins et isort en C

```
void ins(int a[], int n, int j){
  int k=a[j];
  int i=j+1;
  while (i<n && k>a[i]){a[i-1]=a[i];i++;}
  a[i-1]=k;}

void isort(int a[], int n){
  int j;
  for (j=n-2;j>=0;j--){ins(a,n,j);}}
```

1. $>_l$ est un exemple d'*ordre lexicographique*.

La spécification et la preuve sont similaires mais il y a maintenant *beaucoup plus de détails* dont il faut se soucier ! Par exemple, considérons la fonction ins. Il *n'est pas vrai* qu'à chaque pas de calcul, le tableau a contient une permutation de son contenu initial. Si on fait ins(a, 4, 0) sur le tableau {5,1,4,10} on a :

5	1	4	10
1	1	4	10
1	4	4	10
1	4	5	10

Il faut donc *trouver un invariant plus général* pour mener à bien la preuve.

Dans certains domaines (logiciels critiques), on assiste à l'introduction de techniques de preuve formelle de programmes. Ces preuves comportent un grand nombre de détails et elles sont développées en utilisant des *assistants de preuve*. Par exemple, l'outil Frama-C, développé au CEA (`http://frama-c.com/`), est un assistant de preuve spécialisé pour traiter des programmes C. Un assistant de preuve comporte un certain nombre de stratégies qui permettent d'automatiser la synthèse de certaines portions relativement simples de la preuve et d'un petit programme qui est capable de vérifier la correction de l'intégralité de la preuve.

9.4 Test de programmes

Il faut prendre la preuve d'algorithmes et de programmes avec un grain de sel. Voici quelques raisons pour se méfier.

— On fait des erreurs dans la *spécification du problème*.

— La preuve de certains algorithmes sont de nature *très combinatoire* (on oublie des cas...).

— Le passage de la spécification et modèle de l'algorithme à la spécification et modèle du programme entraîne de nombreux *erreurs* (choix des structures de données,...) et *approximations* (flottants,...)

— Par ailleurs, le modèle de programmation peut être *ambigu* (les manuels sont informels...) et/ou pas forcement cohérent avec la *mise-en-oeuvre*.

Pour toutes ces raisons, en pratique il faut toujours *tester* le programme. Le *but* du test est de trouver des erreurs ; en général le test ne peut pas prouver la correction d'un programme. Notez que ce principe implique qu'il n'y a pas de réponse claire à la question : à quel moment peut-on arrêter de tester un programme ? En général, ça dépend du temps dont on dispose.

Un avantage et un inconvénient du test est que ce qu'on teste est le *code compilé* sur *une certaine machine*. Le même programme avec un compilateur et/ou une machine différents pourrait avoir un comportement différent.

On peut remarquer que le travail fait pour la preuve de l'algorithme est souvent bénéfique pour le test. En particulier, pour tester il faut une *spécification* de ce que le programme est censé faire. Pour certains tests (*white box*), il est aussi utile d'avoir un *modèle du langage de programmation*, à savoir un modèle de comment le programme exécute. Par exemple, pour tester tous les branchements du programme. La construction du modèle est un travail pour les *experts*. Il doit être *simple* et en même temps permettre de *prédire* correctement le comportement d'une grande partie des programmes.

Une bonne pratique consiste à garder au moins un test pour chaque erreur trouvée dans le programme et à exécuter à nouveau ce test à chaque modification du programme ; dans ce contexte on parle de *test de non-régression*.

Une autre bonne pratique consiste à *automatiser la génération et la vérification des tests*. Par exemple, dans le cas du tri par insertion on pourrait générer des permutations aléatoires avec la méthode de la section 8.3 et vérifier le résultat en utilisant un autre algorithme de tri ou alors en gardant une bijection entre les éléments dans le tableau en entrée et ceux dans le tableau trié.

Une fois qu'on a acquis une certaine confiance en la correction fonctionnelle du programme, on s'attachera à tester sa *performance*. Par exemple, on peut utiliser la fonction clock décrite dans l'exercice 5 pour estimer son temps d'exécution sur des entrées de taille croissante. Pour des programmes qui allouent dynamiquement de la mémoire dans le tas (voir chapitre 12), on cherchera aussi à détecter des *fuite de mémoire*, à savoir des situations dans lesquelles des segments de mémoire qui ne sont plus utilisés par le programme ne sont pas récupérés.

En général, le programme qu'on teste va interagir avec d'autres programmes qui ne respectent pas forcement sa spécification. Ainsi il est utile de tester le comportement du programme sur des entrées qui ne sont pas prévues par la spécification. On dira qu'un programme est *robuste* s'il est capable de continuer à fonctionner dans un environnement hostile.

Chapitre 10

Types structure et union

Le langage C comporte un certain nombre de types primitifs. Aussi les tableaux et les pointeurs nous permettent de créer des nouveaux types. Par exemple, avec `int a[];` on déclare `a` comme un tableau d'entiers. Dans ce chapitre, on va introduire des nouvelles façons de construire des types.

10.1 Structures

Souvent on a besoin d'*agréger des données de types différents*. Par exemple, le nom (string) et l'âge (int) d'un patient. On pourrait définir un nouveau *type produit* :

$$\text{fiche} = \text{string} \times \text{int} \ .$$

Un élément de type fiche serait donc un *couple*. En utilisant les *projections* on pourrait accéder au premier et deuxième composant. En programmation, on préfère donner des *noms mnémoniques* aux projections. On parle alors de *structures* (en C) ou d'*enregistrements* (*records* en anglais) dans d'autres langages.

La déclaration du type fiche en C pourrait prendre la forme suivante où l'on suppose que 10 caractères suffisent pour représenter un nom :

```
struct fiche {char nom[10] ; int age;} ;
```

Si x est une valeur de type struct fiche on peut accéder au premier composant avec x.nom et au deuxième avec x.age. On insiste sur le fait que le nom du type est struct fiche et non pas fiche. Cependant, il est possible d'utiliser le nom fiche en posant :

```
typedef struct fiche fiche;
```

La *valeur* d'une variable de type fiche est son contenu et non pas l'adresse de mémoire où ce contenu est mémorisé. De ce point de vue, les variables de type structure se comportent comme les variables de type primitif (int, float,...) et non pas comme les variables de type tableau. Si l'on souhaite utiliser une fonction pour modifier une structure on doit soit passer l'adresse de la structure (comme dans (1)) soit recevoir une copie modifiée de la structure (comme dans (2)). Si on procède comme dans (3), la structure de la fonction appelante n'est pas modifiée. Ainsi, dans (4) le nom imprimé sera georges.

```
fiche f(fiche p){strcpy(p.nom,"frank"); return p;}

void g(fiche *p){strcpy((*p).nom,"georges");}

void main(){
  fiche p;
  strcpy(p.nom,"marius"); p.age=27;
  p=f(p);                                          \\(1)
  g(&p);                                           \\(2)
  f(p);                                            \\(3)
  printf("nom=%s,age=%d\n",(p.nom),(p.age));}      \\(4)
```

10.2 Rationnels

Dans cet exemple on illustre l'utilisation du type structure. On utilise le type :

```
struct rat{int num; int den;};
typedef struct rat rat;
```

pour représenter les nombres rationnels avec les conditions suivantes : (1) le champ num représente le numérateur et le champ den le dénominateur, (2) le champ num est un entier et le champ den est toujours un entier positif et (3) si le champ num est différent de 0 alors le plus grand commun diviseur des champs num et den est 1. Dans la suite un *rationnel* est une valeur de type struct rat qui respecte ces conditions. En particulier, une fonction qui prend en argument un rationnel n'a pas à vérifier ces conditions et une fonction qui rend comme résultat un rationnel doit assurer ces conditions. L'intérêt de cette représentation des rationnels par rapport à celle usuelle qui utilise les flottants est qu'en l'absence de débordements les 4 opérations arithmétiques peuvent être calculées de façon exacte (sans approximations).

Exercice 14 *Programmez une fonction d'en tête* void imp_rat(rat r) *qui prend en argument un rationnel et l'imprime (d'une façon agréable à lire) sur la sortie standard. Ensuite, programmez les fonctions suivantes qui déterminent si un rationnel est égal à un autre rationnel et si un rationnel est plus petit ou égal qu'un autre rationnel.*

```
short eq(rat r, rat s)      // égalité
short leq(rat r, rat s)     // plus petit ou égal
```

Il est utile d'avoir une fonction build qui prend deux nombres entiers n et d avec $d \neq 0$ et rend comme résultat un rationnel qui correspond à $\frac{n}{d}$. Une façon élégante de résoudre ce problème est d'utiliser la fonction pgcd considérée dans l'exemple 3. Dans la suite on suppose aussi que abs(n) est la valeur absolue de l'entier n.

```
rat build(int n, int d){
  assert (d!=0);
  rat r;
  if (n==0){r.num=0; r.den=1; return r;}
  int div = pgcd(abs(n), abs(d));
  if (d<0){n=-n;d=-d;}
  if (div>1){n=n/div; d=d/div;}
  r.num =n; r.den=d; return r;}
```

On termine cet exemple avec la programmation de 4 opérations arithmétiques sur les nombres rationnels : la somme, l'inverse additive (l'opposé), la multiplication et l'inverse multiplicative (si elle existe).

```
rat sum(rat r, rat s){
  int d = r.den * s.den;
  int n = r.num * s.den + s.num * r.den;
  return build(n,d);}
rat op(rat r){
  r.num= -r.num; return r;}
rat mul(rat r, rat s){
  int n= r.num * s.num;
  int d = r.den * s.den;
  return build(n,d);}
rat inv(rat r){
  assert (r.num != 0);
  return(build(r.den,r.num));}
```

10.3 Points et segments

On peut imbriquer les déclarations de type. En particulier, on peut déclarer des types structures qui contiennent des types structures. On développe un exemple qui illustre cette possibilité.

Un point (rationnel) dans l'espace cartésien en dimension 2 est représenté par une valeur de type :

```
struct point {rat x; rat y;};
typedef struct point point;
```

et un segment (rationnel) est représenté par une valeur de type :

```
struct segment {point q1; point q2;};
typedef struct segment segment;
```

Les champs `q1` et `q2` correspondent aux points qui déterminent les deux extrémités du segment. Ces extrémités font partie du segment et on peut avoir des segments dégénérés où les deux extrémités coïncident.

On commence par programmer une fonction `allign` qui prend en argument 3 points et retourne 1 s'ils sont alignés (il y a une droite qui passe par les 3 points) et 0 autrement. La fonction utilise une fonction `eqp` pour vérifier l'égalité de deux points et elle distingue 3 cas. Dans (1), au moins deux points sont égaux et donc les 3 points sont alignés. Dans (2), `p1` et `p2` ont la même abscisse et donc les points sont alignes si et seulement si `p3` a la même abscisse que `p1`. Dans (3), on sait que les 3 points sont différents et `p1` et `p2` n'ont pas la même abscisse. On peut donc calculer la droite qui passe par `p1` et `p2` et vérifier si `p3` est sur la droite.

```
short allign(point p1, point p2, point p3){
  if (eqp(p1,p2) || eqp(p1,p3) || eqp(p2,p3)){return 1;}          //(1)
  if (eq(p1.x,p2.x)){return eq(p1.x,p3.x);}                       //(2)
  rat a = mul(sum(p2.y,op(p1.y)),inv(sum(p2.x,op(p1.x))));        //(3)
  rat b = sum(p1.y,op(mul(a,p1.x)));
  return eq(p3.y, sum(mul(a,p3.x),b));}
```

Ensuite, on programme une fonction dist qui prend en argument deux points et calcule leur distance Euclidienne exprimée en tant que valeur de type float.

```
float dist(point p1, point p2){
  rat ry = sum(p2.y, op(p1.y));
  rat rx = sum(p2.x,op(p1.x));
  rat r = sum(mul(ry,ry),mul(rx,rx));
  float f = (float)(r.num)/(float)(r.den);
  return sqrt(f);}
```

On illustre la combinaison de tableaux et de structures en programmant une fonction mindist qui prend en argument un tableau de n points ($n \geq 2$) et retourne la distance Euclidienne minimale entre deux points du tableau.

```
float mindist(int n, point t[n]){
  assert(n>=2);
  float min=dist(t[0],t[1]);
  int i,j;
  for(i=0; i<n;i++){
    for (j=i+1; j<n;j++){
      float d= dist(t[i],t[j]);
      if (d<min){min=d;}}}
  return min;}
```

Pour un autre exemple de combinaison de tableaux et de structures on programme le calcul du barycentre de n points $p_1, \ldots, p_n$ avec la même masse. On rappelle que dans ce cas le barycentre est égal à la somme (vectorielle) des points multipliée par le scalaire $\frac{1}{n}$.

```
point barycentre(int n, point t[n]){
  point s;
  s.x=build(0,1); s.y=build(0,1);
  int i;
  for (i=0;i<n;i++){
    s.x=sum(t[i].x,s.x); s.y=sum(t[i].y,s.y);}
  rat r = build(1,n);
  s.x=mul(s.x,r);
  s.y=mul(s.y,r);
  return s;}
```

Enfin on programme une fonction app qui prend en argument un segment et un point et rend 1 si le point est sur le segment (extrémités comprises) et 0 autrement. Pour résoudre ce problème, on utilise la propriété suivante : si $x, y \in \mathbf{R}^n$ sont deux vecteurs de nombres réels alors $z \in \mathbf{R}^n$ est dans le segment déterminé par x, y si et seulement si $z = \lambda \cdot x + (1 - \lambda) \cdot y$ où $\lambda \in \mathbf{R}$ et $0 \leq \lambda \leq 1$. On distingue 3 cas. Dans (1), p1 = p2 et donc il faut que p = p1. Dans (2), p1 et p2 sont différents mais ont la même abscisse ; on calcule le λ en utilisant les ordonnées. Dans (3), on est dans la situation symétrique où p1 et p2 sont différents et n'ont pas la même abscisse ; on peut donc calculer le λ en utilisant les abscisses.

```
short app(point p, segment seg){
  point p1=seg.q1;
  point p2=seg.q2;
  if (eqp(p1,p2)){return eqp(p1,p);}                    //(1)
  rat lam;
  if (eq(p1.x,p2.x)){                                   //(2)
```

```
  lam = mul(sum(p.y, op(p2.y)), inv(sum(p1.y,op(p2.y))));}
  else {                                                    //(3)
  lam = mul(sum(p.x, op(p2.x)), inv(sum(p1.x,op(p2.x))));}
  return (leq(build(0,1),lam) && leq(lam,build(1,1)));}
```

10.4 Unions

Parfois on souhaite disposer d'une variable qui peut prendre une valeur de types différents. Par exemple, le nom ou l'âge d'un patient. On pourrait définir un nouveau *type union (disjointe)* :

$$\mathsf{fiche} = \mathsf{string} + \mathsf{int}\ .$$

Un élément de type fiche serait alors soit un string soit un int. A nouveau, on préfère donner des *noms mnémoniques*. Voici une déclaration possible du type fiche en C :

```
union fiche {char nom[10] ; int age;} ;
```

Comme pour les structures, si x a type union fiche on accède à sa valeur en écrivant x.nom ou x.age. Aussi la valeur d'une variable de type union est son contenu (pas son adresse).

Du point de vue de l'utilisation de la mémoire, il peut être intéressant d'utiliser un type union au lieu d'un type structure. Par exemple, pour mémoriser une valeur de type union fiche il faut 10 octets alors que pour mémoriser une valeur de type struct fiche il faut $14 = 10 + 4$ octets.

Dans le langage C, les types unions ne protègent pas le programmeur de certains erreurs. En effet, rien nous empêche d'écrire :

```
union fiche x;
(x.nom)[0]='b';
x.age=x.age+1;
```

En général, le compilateur ne sait pas prévoir si une variable de type union fiche contiendra un tableau de caractères ou un entier. En principe, il est possible de : (i) intégrer dans une valeur de type union une information qui nous permet de déduire son type et (ii) vérifier au moment de l'exécution la cohérence des opérations qu'on effectue sur des valeurs de type union.

Exemple 32 *On suppose qu'une figure est soit un cercle soit un triangle qu'on représente avec les types suivants.*

```
struct point {float x; float y;};
typedef struct point point;

struct cercle {point centre; float rayon;};
typedef struct cercle cercle;

struct triangle {point p1; point p2; point p3;};
typedef struct triangle triangle;
```

On programme une fonction qui prend en argument une figure et retourne son perimètre ; la programmation de la fonction distance dist *est omise.*

```
enum lfig {CERCLE, TRIANGLE};
typedef enum lfig lfig;

union ufig {cercle c; triangle t;};
typedef union ufig ufig;

struct figure {lfig l; ufig u;};
typedef struct figure figure;

float perim(figure f){
  switch(f.l){
  case CERCLE: return 2 * M_PI * f.u.c.rayon;
  case TRIANGLE: return dist(f.u.t.p1,f.u.t.p2)+dist(f.u.t.p1,f.u.t.p3)+dist(f.u.t.p2,f.u.t.p3);
  default: exit(1);
  }}
```

Et voici deux appels possibles à la fonction perim *:*

```
  figure f;
  f.l=CERCLE;
  f.u.c.centre.x=0; f.u.c.centre.y=0; f.u.c.rayon=1;
  printf("%f\n",perim(f));
  f.l=TRIANGLE;
  f.u.t.p1.x=1; f.u.t.p2.x=0; f.u.t.p3.x=-1;
  f.u.t.p1.y=0; f.u.t.p2.y=1; f.u.t.p3.y=0;
  printf("%f\n",perim(f));
```

Chapitre 11

Pointeurs

Dans les chapitres précédents on a évoqué l'utilisation de pointeurs (ou adresses de mémoire) dans le cadre de l'utilisation de la fonction scanf (section 2.3) et pour le passage de tableaux comme arguments d'une fonction (section 7.2). Dans ce chapitre, on va examiner d'autres utilisations possibles des pointeurs en C.

11.1 Pointeurs de variables

On a déjà vu que l'opérateur & permet de récupérer l'adresse d'un variable. Donc si x est une variable &x est l'adresse associée à la variable. On peut aussi bien appliquer l'opérateur & à un élément d'un tableau comme dans &(x[3]). Par contre, l'application de l'opérateur & à une entité qui n'a pas une adresse associée produit une erreur. Par exemple &3 n'est pas une expression correcte.

Il existe aussi un deuxième opérateur '*', dit de déréférencement, qui étant donné une adresse permet de récupérer le contenu de l'adresse. En particulier, si x est une variable alors l'évaluation de l'expression $*(\&x)$ donne exactement le même résultat que l'évaluation de la variable x.

Le langage C a une notation un peu particulière pour indiquer les types des pointeurs. Par exemple, plutôt que dire : 'la variable p a le type des pointeurs à int', en C on dit : 'le déréférencement de la variable p a le type int'. Ainsi, la déclaration de la variable p a la forme :

int * p .

De la même façon, pour déclarer un tableau d'entiers on écrit :

int a[]

et pour déclarer une fonction f qui prend en argument un pointeur à un entier et retourne un pointeur à un entier on écrit :

int * f(int * p){...}

On va maintenant considérer différentes utilisation des pointeurs de variables.

Exemple 33 *Dans* (1)*,* p *est un pointeur d'entier qui reçoit l'adresse de* x *dans* (2)*. Dans* (3)*, le contenu de l'adresse* p *(donc la valeur de* x*) est affecté à* y *et donc dans* (4) *on imprime* 1*. Dans* (5)*,* p *prend l'adresse de* z[0] *et donc dans* (6)*, on affecte à* z[0] *la valeur* 1 *qu'on imprime dans* (7)*.*

```
main(){
  int x=1, y=2, z[10], *p;    \\(1)
  p=&x;                       \\(2)
  y=*p;                       \\(3)
  printf("y=%d\n",y);         \\(4)
  p=&z[0];                    \\(5)
  *p=1;                       \\(6)
  printf("z[0]=%d\n",*p);}    \\(7)
```

Exemple 34 *La fonction* f *déclare* x *et* y *comme des pointeurs d'entiers. Ainsi,* f *est capable de permuter le contenu des variables* a *et* b *de la fonction appelante* main*.*

```
void f(int *x, int*y){
  int aux;
  aux=*x;
  *x=*y;
  *y=aux;}
main(){
  int a=1, b=2;
  f(&a,&b);
  printf("a %d\n",a);
  printf("b %d\n",b);}
```

Exemple 35 *La fonction* f *retourne comme résultat un pointeur à sa variable locale* x*. Il convient d'éviter ce type de programme ! La* fonction appelante *ne devrait jamais accéder les variables locales de la* fonction appelée *car ces variables risquent fort d'être compromises quand la fonction appelée retourne. La situation inverse est par contre admissible et on en a déjà vu un exemple avec la fonction* scanf*.*

```
int *f(){int x=1; return &x;}
main(){int *p=f(); printf("*p=%d\n",*p);}
```

11.2 Pointeurs de tableaux

En C, on peut utiliser les pointeurs pour manipuler les tableaux. Ainsi, les fonctions suivantes ont le même effet :

```
void f(int a[]){a[3]=5;}
void g(int *p){*(p+3)=5;}
```

La notation pour les tableaux semble plus lisible et autant que possible elle est à notre avis à préférer. Une particularité de C est de permettre une forme limitée d'arithmétique sur les pointeurs. En particulier, il est possible :

— d'obtenir un pointeur en additionnant un pointeur avec un entier.

— d'obtenir un entier en calculant la différence de deux pointeurs.

Ainsi si b et h sont des pointeurs alors l'expression $b + (h - b)/2$ dénote un pointeur alors que l'expression $(b + h)/2$ est refusée par le compilateur.

Exemple 36 *On programme une fonction qui effectue une recherche dichotomique d'un entier* x *sur un segment de mémoire censé contenir une suite croissante de* n *entiers à partir de l'adresse* t*.*

```
int dicho(int *t,int n,int x){
int *b=t;
int *h=t+(n-1);
while(1){
  int *m=b+(h-b)/2;
  if (*m==x){return 1;}
  if ((*m<x)&&(m!=h)){b=m+1; continue;}
  if ((*m>x)&&(m!=b)){h=m-1;continue;}
  return 0;
}}
```

11.3 Pointeurs de char

La bibliothèque *ctype.h* contient un certain nombre de fonctions qui permettent de classifier et manipuler les valeurs de type char : caractères alphabétiques, minuscules, majuscules, chiffres, espaces,... Notez que la frontière entre caractères et entiers est assez floue. Par exemple, les fonctions en question acceptent en argument et retournent des valeurs de type int.

Les pointeurs sont souvent utilisés pour manipuler des *suites de caractères*. Plusieurs langages ont un type de base string et des fonctions de bibliothèque. En C, on préfère exposer les *détails de la représentation* : ainsi une suite de caractères est un *pointeur de* char qui se termine par un caractère spéciale $'\backslash 0'$. De façon équivalente, c'est un *tableau de* char dont la fin est marquée par $'\backslash 0'$. L'inconvénient de cette approche 'bas niveau' est que c'est à la charge du programmeur d'allouer des tableaux assez grands !

Exemple 37 *Dans* (4) *on utilise la directive* %s *pour lire une chaîne de caractères. La chaîne ne doit pas dépasser* 10 *caractères (en comptant aussi le symbole spécial* \0*). Ce programme utilise deux fonctions de la bibliothèque* string.h*, à savoir* strcpy *(copie) et* strcat *(concaténation). Dans* (7)*, on copie* i *dans* o *et dans* (8) *on concatène* middle *à* o*. De même, dans* (9) *on concatène* i *à* o*. Enfin, dans* (10) *on imprime* o *avec la directive* %s*. Ce type de programmation est* fragile *à cause des débordements possibles des tableaux de caractères qui ne sont pas remarqués par le compilateur. Ainsi, il est possible que dans* (10) *on imprime aussi le contenu du tableau* b *qui à priori n'a rien à voir avec le contenu du tableau* o*.*

```
main(){                          //(1)
  char i[10];                    //(2)
  printf("Entrez une chaine\n"); //(3)
  scanf("%s",i);                 //(4)
  char o[20];                    //(5)
  char b[10]="philippe";         //(6)
  strcpy(o,i);                   //(7)
  strcat(o,"middle");            //(8)
  strcat(o,i);                   //(9)
  printf("%s\n",o);}             //(10)
```

11.4 Fonctions de fonctions et pointeurs de fonctions

Il n'est pas rare de rencontrer des fonctions qui prennent des *fonctions comme argument* et/ou qui rendent une *fonction comme résultat*. Par exemple, en analyse les opérations de dérivation et d'intégration prennent une fonction en argument et rendent une fonction

comme résultat. Dans certains langages de programmation, il est possible d'écrire et de typer directement ces fonctions d'*ordre supérieur*. Dans le langage C, on considère qu'une fonction est l'adresse d'un segment de code et on utilise les *pointeurs de fonction* pour manipuler ces adresses. [1]

Exemple 38 *Voici une fonction* intmap *qui attend en argument un pointeur de fonction* f *de* int *vers* int *et un tableau de* n *entiers et applique la fonction* f *à chaque élément du tableau.*

```
void intmap(int(*f)(int), int n, int t[n]){     \\(1)
  int i;                                        \\(2)
  for(i=0;i<n;i++){                             \\(3)
    t[i]=(*f)(t[i]);                            \\(4)
  }
}
```

On peut appeller la fonction intmap *en lui passant en argument, par exemple, une fois une fonction pour élever au carré et une autre fois une fonction pour élever au cube comme dans le code suivant.*

```
int square(int x){return x*x;}
int cube(int x){return x*x*x;}
void main(){
  int t[5] = {4,4,1,0,3};
   intmap(square,5,t);
   (...)
   intmap(cube,5,t);
   (...)
}
```

Exemple 39 *On peut adapter la fonction* intmap *de l'exemple précédent aux chaînes de caractères. Voici une fonction* stringmap *qui attend un pointeur de fonction* f *de* char *vers* char*, un pointeur à une chaîne de caractères* c *et une longueur* l *et applique la fonction* f *à chaque caractère en prenant en compte la longueur* l *et le caractère spécial qui marque la fin de la chaîne.*

```
void stringmap(char(*f)(char), char * c, int l){
  unsigned i=0;
  while((i<l)&&(*(c+i)!='\0')){*(c+i)=(*f)(*(c+i));i++;}
}
```

11.5 Fonctions génériques et pointeurs vers void

Les fonctions intmap et stringmap des exemples 38 et 39 sont suffisamment similaires pour envisager d'écrire une seule fonction map. On parle alors de fonctions *génériques* ou *polymorphes*. Certains langages de programmation, ont un système de typage assez puissant pour exprimer les caractéristiques communes de intmap et stringmap. En C, le mécanisme de base pour écrire des fonctions génériques consiste à utiliser un pointeur vers void en sachant que :

— Tout pointeur est converti implicitement à un pointeur vers void.

1. En général, ce point de vue est insuffisant et il est necéssaire d'ajouter de l'information pour représenter l'environnement dans lequel la fonction définie.

— Tout pointeur vers void peut être converti explicitement par le programmeur à un pointeur d'un type arbitraire.

Par exemple, la fonction swap ci-dessous prend un tableau de pointeurs vers void et permute les premiers deux pointeurs du tableau. On peut déclarer un tableau tint de pointeurs d'entiers et appeler swap((void *)tint) et aussi déclarer un tableau de pointeurs de caractères tchar et appeler swap((void *)tchar).

```
void swap(void * t[2]){
  void * aux;
  aux=t[0];
  t[0]=t[1];
  t[1]=aux;}
```

En général, le programmeur utilise l'opérateur de cast pour autoriser certaines manipulations. Dans ce cas, c'est à la charge du programmeur de s'assurer que l'utilisation des pointeurs est cohérente. Considérons le programme suivant.

```
int void_inc(void  *p){          //(1)
  int x=*(int*)(p);              //(2)
  return x+1;}                   //(3)
void main(){                     //(4)
  int y=1;                       //(5)
  printf("%d\n", void_inc(&y));  //(6)
  char a='a';                    //(7)
  printf("%d\n", void_inc(&a));} //(8)
```

Dans (2), la fonction void_inc prétend que p pointe vers un entier mais dans (8) la fonction main lui passe en argument un pointeur vers un caractère. Ainsi, avec mon compilateur le programme imprime 2 et 75780194 !

Cet exemple montre qu'en faisant des cast, le programmeur peut introduire des erreurs de typage qui ne seront pas détectés par le compilateur. Avec toutes ces reserves, voici une façon d'écrire une fonction map générique.

Exemple 40 *Avec les reserves évoquées ci-dessus, voici une façon de programmer et d'utiliser une fonction* map *générique.*

```
void mapgen(void * tab, int n, size_t t, void (*f)(void *)){ //(1)
  int i;
  for(i=0;i<n;i++){f(tab+(i*t));}                             //(2)
}
void square(void *x){                                         //(3)
  int *a=(int *)x;
  *a=(*a)*(*a);
}
void main(){
  int a[3]={4,5,6};
  mapgen(a,3,sizeof(int),square);                             //(4)
}
```

Dans (1)*, la fonction* mapgen *attend un pointeur (vers un tableau)* tab*, le nombre d'éléments* n *du tableau, leur taille (en octets)* t *et un pointeur de fonction* f *qui attend un pointeur et ne retourne pas de résultat (la fonction* f *agit donc par effet de bord). Pour utiliser la fonction* mapgen *pour élever au carré un tableau d'entiers, on commence par déclarer dans* (3) *une*

fonction square *du type attendu par* mapgen. *La fonction convertit explicitement un pointeur vers* void *en pointeur vers* int *et ensuite élève au carré son contenu. Notez que dans* (4) *le compilateur ne fait pratiquement aucune vérification ; c'est l'utilisateur qui doit assurer la cohérence des arguments fournis à mapgen.*

Exemple 41 *On aimerait écrire une fonction de* tri *qui prend en argument un tableau d'éléments de type* T *et un prédicat de comparaison* cmp : T × T → Bool *et qui trie le tableau par ordre croissant d'après l'ordre défini par le prédicat. Il s'agit donc de combiner la notion de pointeur de fonction avec celle de fonction générique. On illustre l'approche dans le cadre de la fonction de tri* qsort *qui se trouve dans la bibliothèque* stdlib.h. *Le type de la fonction* qsort *est le suivant :*

```
void  qsort(void *base, size_t n, size_t size,
            int (*cmp)(const void *,const void *))
```

- base *pointe à un tableau (du moins on l'espère car son type est pointeur vers* void*).*
- size_t *est un type prédéfini d'entiers non-signés. La fonction* sizeof(T) *donne la taille (en octets) d'une valeur de type* T.
- n *est la taille du tableau.*
- size *est la taille d'un élément du tableau.*
- const *indique que l'argument n'est pas modifié (est constant).*
- cmp *prend deux arguments et rend une valeur* négative, 0 *ou* positive *si le premier est* plus petit, égal *ou* plus grand *que le second.*

Il est possible d'appliquer la fonction de tri qsort *à des tableaux de type différent et avec des prédicats de comparaison différents. Par exemple, pour trier de façon croissante un tableau d'entiers avec l'ordre standard on peut déclarer une fonction de comparaison* cmp_int *et un tableau d'entiers* t *et appeler la fonction* qsort. *Mais on peut aussi déclarer une fonction de comparaison sur les caractères* cmp_char *avec un ordre alphabétique décroissant et un tableau de caractères* a *et y appliquer la même fonction* qsort.

```
int cmp_int(const void  *p,const void *q){
  int x=*(const int*)(p);
  int y=*(const int*)(q);
    if (x<y){return -1;}
    else {if (x==y){return 0;}
          else {return 1;}}}
int cmp_char(const void *p, const void *q){
  char x=*(const char*)(p);
  char y=*(const char*)(q);
  if (x>y){return -1;}
  else {if (x==y){return 0;}
        else {return 1;}}}
void main(){
  int t[5] = {4,4,1,0,3};
  qsort(t, (size_t)5, sizeof(int),cmp_int);
  char a[4] = {'a', 'd', 'c', 'a'};
  qsort(a, (size_t)4, sizeof(char), cmp_char); }
```

11.6 Pointeurs de fichiers

Les pointeurs de fichiers permettent de gérer les entrées sorties en utilisant des fichiers plutôt que l'écran comme on l'a fait jusqu'à maintenant.

Par exemple, supposons que l'on souhaite lire les entrées d'un fichier input et imprimer les sorties dans un fichier output.

Une première solution consiste à utiliser les opérateurs de redirection de Unix comme dans :

```
./a.out < input > output
```

Une deuxième solution consiste à utiliser les opérateurs C de la bibliothèque stdio.h qui remplacent l'entrée standard par le fichier input et la sortie standard par output. Cette deuxième solution est plus flexible et elle ne dépend pas du système d'exploitation. Voici un exemple de programme.

```
void main(){
  int x; FILE * f;
  f=fopen("input","r");         //f pointe vers input
  fscanf(f,"%d",&x);
  fclose(f);                    //f ne pointe plus vers input
  f=fopen("output","w");        //f pointe vers output
  fprintf(f,"%d\n",x+1);
  fclose(f);            }       //f ne pointe plus vers output
```

Supposons maintenant que l'on souhaite passer à l'exécutable des *arguments*. Par exemple, les noms des fichiers qu'il doit lire/écrire comme dans :

```
./a.out input output
```

Jusqu'à maintenant, on a supposé que la fonction main ne prend par d'arguments. Cependant, il est possible de déclarer des arguments pour cette fonction comme dans l'exemple suivant.

```
void main(int argc, char *argv[]){
  int x; FILE * f;
  f=fopen(argv[1],"r");
  fscanf(f,"%d",&x);
  fclose(f);
  f=fopen(argv[2],"w");
  fprintf(f,"%d\n",x+1);
  fclose(f);}
```

La variable argc représente le nombre d'arguments qu'on passe à l'exécutable (2 dans notre exemple) et la variable argv est un tableau de chaînes de caractères (techniquement un tableau de pointeurs de char). Par convention, le premier élément de ce tableau argv[0] est réservé pour le nom de l'exécutable. Les noms des fichiers input et output qu'on passe en argument à l'exécutable a.out sont donc mémorisés dans argv[1] et argv[2] respectivement.

Exercice 15 *Voici un exercice qui permet d'utiliser les différents aspects des pointeurs décrits dans ce chapitre. Il s'agit de reprogrammer la fonction* sort *de* Unix.

- *Ouvrez un fichier* input.
- *Comptez le nombre de lignes dans le fichier* input *et le nombre maximum de caractères par ligne.*
- *Allouez un tableau qui contient tous les caractères du fichier et un tableau qui contient les pointeurs au debut de chaque ligne dans le premier tableau.*
- *Utilisez la fonction de bibliothèque* qsort *pour trier le tableau de pointeurs en suivant l'ordre alphabétique.*
- *Imprimez les lignes ordonnées dans un fichier* output.

Chapitre 12

Listes et gestion de la mémoire

En C, il est possible de déclarer des types *structure* qui contiennent des types pointeurs à la structure qu'on est en train de déclarer. Il s'agit d'une forme de définition *récursive* (au niveau des types plutôt qu'au niveau des fonctions). Ce type de déclarations ouvre la possibilité de représenter des données avec des formes plus ou moins élaborées : des *listes*, des *arbres*, des *graphes*,... Dans ce chapitre introductif, on se limitera à considérer les *listes* qu'on peut visualiser comme des suites d'éléments constitués d'une valeur et d'un pointeur vers le prochain élément de la suite. Il est naturel de considérer des listes dont la taille varie dynamiquement pendant le calcul et dans ce contexte on est amené à reconsidérer le modèle mémoire de C. Il est aussi naturel d'adapter aux listes les algorithmes qui utilisent les tableaux et de représenter des ensembles finis comme des listes.

12.1 Listes

En C, on peut déclarer par exemple :

```
struct node{int val; struct node *next;} ;
struct tnode{int tval; struct tnode *left; struct tnode *right;} ;
```

La structure node contient un champ next qui est un pointeur à une structure node et la structure tnode contient deux champs left et right qui sont des pointeurs à une structure tnode. On peut utiliser node pour représenter des listes et on verra plus tard qu'on peut utiliser tnode pour représenter des arbres binaires. En C on peut voir une liste non vide (d'entiers) comme une collection de valeurs de type struct node avec adresses $\ell_1, \ldots, \ell_n$ telle qu'il existe une permutation π sur $\{1, \ldots, n\}$ avec la propriété que :

$$\begin{array}{ll} \ell_{\pi(1)} \rightarrow \mathsf{next} & = \ell_{\pi(2)} \\ \ell_{\pi(2)} \rightarrow \mathsf{next} & = \ell_{\pi(3)} \\ \cdots & \\ \ell_{\pi(n-1)} \rightarrow \mathsf{next} & = \ell_{\pi(n)} \\ \ell_{\pi(n)} \rightarrow \mathsf{next} & = \mathsf{NULL} \end{array}$$

La notation $\ell \rightarrow$ next indique le contenu du champ next de la structure node mémorisée à l'adresse ℓ. On note qu'en C on peut écrire p−> val à la place de (∗p).val.

La valeur NULL est un pointeur (adresse) prédéfini de C. Le pointeur NULL habite tous les types pointeur mais c'est une erreur d'essayer d'accéder un champ du pointeur NULL. Par

exemple, le programme suivant compile mais produit une erreur au moment de l'exécution car dans (1) on cherche à lire le champ val de NULL.

```
void main(){
  struct node {int val; struct node *next;} ;
  struct node x,y;
  x.val=3;
  y.val=4;
  x.next = &y;
  printf("%d", (*(x.next)).val);
  x.next = NULL;
  printf("%d",(*(x.next)).val);}                //(1)
```

12.2 Allocation de mémoire

Les listes permettent une gestion de la mémoire plus flexible. Supposons que l'on doit lire et mémoriser une suite d'entiers dont on ne connaît pas le nombre à l'avance. Une approche possible serait d'allouer un tableau... mais comment décider la taille du tableau ? On risque de ne pas avoir un tableau assez grand ou d'utiliser seulement une petite partie du tableau. Une solution plus flexible consiste à allouer une structure qui par exemple a le type :

```
struct tabnode{int t[1000]; struct tabnode * next;};
```

Une telle structure peut mémoriser jusqu'à 1000 entiers et au cas où elle serait saturée il est possible d'allouer une autre structure du même type et de la connecter à la précédente. De cette façon on pourra continuer à lire et mémoriser la suite d'entiers tant que la mémoire (virtuelle) de l'ordinateur contient un segment suffisant à contenir une valeur de type struct tabnode (typiquement 4004 octets). On peut remarquer qu'on paye un petit prix pour cette flexibilité : environ 1 octet sur 1000 est utilisé pour mémoriser les pointeurs du champ next.

Dans notre exemple, on doit allouer dynamiquement (pendant le calcul) des structures de type tabnode. En C, pour *allouer* une structure on utilise la fonction malloc de la bibliothèque stndlib.h. [1] Il convient d'encapsuler la fonction malloc dans une fonction C. Par exemple, pour allouer une structure de type node on peut utiliser la fonction suivante.

```
struct node {int val; struct node * next;};
typedef struct node node;

node *allocate_node(int v){
  node *p=malloc(sizeof(node));
  (p->val)=v;
  (p->next)=NULL;
  return p;}
```

La fonction sizeof est une autre fonction de bibliothèque qui prend en entrée un type et retourne un nombre naturel qui indique le nombre d'octets nécessaires pour mémoriser une valeur du type en question. Par example, pour le type node ce nombre est typiquement 8. La fonction malloc retourne un pointeur vers void qui est l'adresse de base du segment de mémoire alloué. Remarquons aussi que la fonction allocate_node initialise les champs val et next de la structure.

1. D'autres fonctions avec des fonctionnalités comparables sont calloc et realloc.

12.3 Récupération de mémoire

Le langage C n'a *pas de ramasse miettes* (*garbage collector*, en anglais). La récupération de la mémoire allouée avec malloc est *à la charge du programmeur* et elle est possible avec la fonction de bibliothèque free. [2]

Si p est un pointeur à un bloc de mémoire (par exemple, un pointeur à une structure) alors free(p) a l'effet de libérer le bloc et donc de le rendre réutilisable dans les prochains appels à malloc.

Il est *catastrophique* :

— d'appeler free(p) et ensuite d'accéder au bloc pointé par p en lecture ou écriture.

— d'exécuter plusieurs fois free(p).

Utilisez free seulement si vous n'avez *pas assez de mémoire* et si vous êtes sûrs que l'élément libéré ne sera *pas utilisé* dans la suite du calcul.

Remarque 10 *L'introduction de* malloc *et* free *nous oblige à* raffiner *notre* modèle de la mémoire*. On peut maintenant distinguer* 3 zones de mémoire*.*

— *Une* zone statique *où l'on mémorise les* données globales *dont la vie termine avec la terminaison du programme.*

— *Une* pile *où l'on mémorise les* données locales *à un appel de fonction dont la vie termine avec le retour de la fonction. La machine s'occupe de récupérer automatiquement cet espace mémoire.*

— *Un* tas *où le programmeur alloue de la mémoire avec* malloc *et la récupère avec* free*.*

12.4 Tri par insertion avec des listes

Une suite finie d'éléments se représente aisement comme une liste et dans ce cadre on peut adapter aux listes les algorithmes développés pour les tableaux. On considère le cas du tri par insertion (section 8.1). On suppose la déclaration du type struct node ci-dessus. La fonction isort prend une liste d'entiers et la trie par ordre croissant en utilisant la fonction auxiliaire d'insertion ins. On fait un calcul *en place* (*in place*, en anglais) à savoir on n'alloue pas des nouvelles structures mais on se limite à modifier les pointeurs des champs next des structures existantes. La complexité du tri par insertion sur les listes est toujours quadratique.

```
node * ins(node * list, node * n){
  assert(n!=NULL);
  if (list==NULL){(n->next)=NULL; return n;};
  if ((list->val)>=(n->val)){(n->next)=list; return n;};
  (list->next)=ins(list->next,n); return list;}
node * isort(node * list){
  if (list==NULL){return list;};
  return ins(isort(list->next), list);}
```

12.5 Ensembles finis comme listes

On considère le problème de représenter les sous-ensembles finis d'un certain ensemble ordonné (et pas forcement fini). Dans la suite nous traiterons des ensembles finis de nombres

2. Alternativement, on peut utiliser des bibliothèques, voir par exemple [BW88].

entiers avec l'ordre standard. Les opérations dont l'on souhaite disposer sur ces ensembles finis sont les suivantes :

— création de l'ensemble vide (emp).
— insertion d'un élément (ins).
— test d'appartenance d'un élément (mem).
— élimination d'un élément (rem).
— impression de l'ensemble (pri).

On choisit de représenter un ensemble fini par une liste. Ce choix à l'avantage de la simplicité mais d'autres solutions plus efficaces (arbres binaires de recherche, tables de hachage, listes à enjambements,...) sont possibles.

Comme dans la section 12.1, nous ferons l'hypothèse que chaque noeud est représenté par une structure avec 2 champs avec noms val pour une valeur entière et next pour un pointeur. En C, on va supposer la déclaration de type structure suivante :

```
struct node{int val; struct node *next;};
typedef struct node node;
```

On rappelle aussi la fonction qui alloue une structure node en utilisant la fonction malloc.

```
node *allocate_node(int v){
  node *p=(node *)(malloc(sizeof(node)));
  (*p).val=v;
  (*p).next=NULL;
  return p;}
```

On va supposer que les entiers dans l'ensemble sont mémorisés dans la liste par ordre croissant. Un escamotage qui permet de simplifier la programmation des opérations consiste à créer deux noeuds sentinelles qui contiennent des entiers non-standard $-\infty$ et $+\infty$. En pratique, on peut utiliser les constantes INT_MIN et INT_MAX de la bibliothèque limits.h. La liste qui correspond à l'ensemble vide va donc contenir deux noeuds avec valeurs INT_MIN et INT_MAX. La fonction C qui permet de créer l'ensemble vide est la suivante.

```
node * emp(){
  node *head = allocate_node(INT_MIN);
  node *tail = allocate_node(INT_MAX);
  (*head).next=tail;
  return head;}
```

Les deux opérations plus compliquées sont celles pour insérer et éliminer. Elles ont une structure assez similaire qui consiste à faire glisser deux pointeurs pred et curr dans la liste jusqu'à trouver le point où l'action d'insertion ou d'élimination doit avoir lieu. On remarquera que l'insertion utilise la fonction malloc et l'élimination la fonction free.

```
void ins(int v, node *list){
  node *pred=list;
  node *curr=(list->next);
  while ((curr->val)<v){pred=curr; curr=(curr->next);}
  if((curr->val)!=v){
    node * new=allocate_node(v);
    (new->next)=curr;
    (pred->next)=new;}
  return;}
```

```
short rem(int v, node *list){
  node *pred=list;
  node *curr=(list->next);
  while ((curr->val)<v){pred=curr; curr=(curr->next);}
  if((curr->val)==v){(pred->next)=(curr->next); free(curr);} //FREE
  return 0;}
```

Exercice 16 *Programmez les fonctions pour tester l'appartenance et pour imprimer un ensemble.*

Exercice 17 *Reprogrammez les fonctions* ins *et* rem *en supposant que maintenant on n'a pas de noeuds sentinelles et que donc l'ensemble vide correspond à la liste vide.*

Exercice 18 *On souhaite représenter des ensembles d'entiers avec au plus n éléments. Fixez une représentation de ces ensembles par des tableaux d'entiers et étudiez la mise en oeuvre des opérations* emp, ins, mem, rem *et* pri *évoquées au début de cette section.*

Exercice 19 *Un multiensemble est un ensemble où chaque élément peut être dupliqué un certain nombre de fois. Formellement, un multiensemble sur un ensemble support A est une fonction $m : A \to \mathbf{N}$. Le nombre naturel $m(a)$ indique le nombre de copies disponibles de l'élément a ; on dit aussi la* multiplicité *de a. Un multiensemble m est* fini *si $\{a \in A \mid m(a) > 0\}$ est fini. On peut effectuer sur les multiensembles finis des opérations similaires à celles évoquées pour les ensembles finis. La différence est que l'opération d'insertion augment de 1 la multiplicité d'un élément et l'opération d'élimination la diminue de 1 (si elle est positive). On prend le support A comme l'ensemble des entiers. Proposez une représentation des multiensembles d'entiers par des listes.*

Chapitre 13

Piles et queues

On introduit deux exemples élémentaires de *structures de données* : les piles et les queues. Une structure de données est un peu l'analogue informatique d'une structure algébrique (groupes, anneaux,...) : on y trouve des données et un certain nombre de fonctions pour les manipuler. Un principe de base de la modularisation des programmes consiste à concevoir des structures de données dans lesquelles on distingue une représentation *externe* visible à l'utilisateur et une représentation *interne* qui devrait être invisible à l'utilisateur. Dans ce contexte, les structures de données constituent un élément essentiel pour la *modularisation* d'un programme. On présente une technique de programmation qui permet de réaliser cette idée en C et on termine en présentant deux exemples d'application des structures de données introduites.

13.1 Piles et queues

On considère des *suites finies* sur un ensemble support A (par exemple les nombres entiers) avec opérations pour insérer et éliminer un élément de la suite. Dans ce contexte, on distingue deux structures de données : la *pile* et la *queue*. Dans les deux cas, on peut supposer que l'opération d'insertion consiste à prolonger la suite d'un élément. Ainsi l'insertion d'un élément a dans la suite $a_0, \ldots, a_{n-1}$ produit la suite $a_0, \ldots, a_{n-1}, a$. La différence apparaît alors dans l'opération d'extraction. Dans une *pile*, l'élément extrait est le dernier de la suite (si la suite est non vide) ; donc le dernier élément inséré est le premier à être extrait (*last-in first-out (LIFO)*, en anglais). Dans une *queue*, l'élément extrait est le premier de la suite (si la suite est non vide) ; donc l'élément extrait est le premier inséré (*first-in first-out (FIFO)*, en anglais).

Si on connaît le nombre maximum d'éléments dans une pile (ou dans une queue) et si ce nombre est raisonnable alors on peut stocker les éléments dans un *tableau*. Autrement, on peut utiliser une *liste*. Il est assez facile de mettre en oeuvre les opérations d'insertion et d'extraction en *temps constant* ($O(1)$). On va commenter les 4 cas possibles.

Pile comme liste

On dispose d'une variable top de type pointeur à un noeud de la liste. On peut créer une pile en initialisant top à NULL. Pour insérer un élément, on alloue avec malloc un nouveau noeud qui contient l'élément et on l'insère au sommet de la liste. Pour extraire un élément, on vérifie d'abord que top $\neq$ NULL et dans ce cas on récupère le premier noeud de la liste.

Pile comme tableau

On dispose d'un tableau p et d'une variable top de type entier qui contient l'indice de la première cellule libre du tableau. On peut créer une pile en déclarant le tableau et en initialisant top à 0. Pour insérer un élément on l'écrit dans p[top] et on incrémente top. Pour extraire un élément on vérifie d'abord que top > 0 et si c'est le cas on décrémente top et on retourne p[top].

Queue comme liste

On dispose de deux pointeurs aux noeuds de la liste : head et tail. On peut créer une queue en initialisant head et tail à NULL. On insère un élément en allouant un noeud qui est pointé par tail. On élimine un élément en récupérant le noeud pointé par head (s'il existe) et en faisant pointer head au noeud suivant (ou à NULL). Si nécessaire on mettra à jour tail aussi. Le lecteur remarquera que pour réaliser les opérations en $O(1)$ il est important de disposer d'un deuxième pointeur (tail), d'insérer à la fin de la liste et d'éliminer à son sommet. Par exemple, pour éliminer un noeud à la fin de la liste on est obligé de parcourir toute la liste ; une opération qu'on ne sait pas faire en temps constant.

Queue comme tableau

On dispose de deux variables de type entier head et tail et d'un compteur count aussi de type entier. On peut créer une queue en allouant un tableau q avec n cellules et en initialisant head, tail et count à 0. Les éléments de la queue vont être mémorisés dans les cellules du tableau comprises entre head et tail (strictement) étant entendu qu'on compte modulo n. Ainsi si n=10, head=7 et tail=2 les éléments de la queue se trouvent dans q[7],q[8],q[9],q[0],q[1]. La fonction ins retourne une valeur 0 ou 1 pour indiquer si la queue est déjà *pleine*. La fonction rem retourne une constante INT_MIN pour indiquer que la queue est *vide*.

```
short ins(int x){
  assert(n>=1);
  short r;
  if (count==n){r=0;}
  else {q[tail]=x; tail=(tail+1)%n; count++; r=1;}
  return r;}
int rem(){
  int r;
  if (count==0){r=INT_MIN;}
  else {r=q[head]; head=(head+1)%n; count--;}
  return r;}
```

13.2 Modularisation

En C, pour pallier à l'absence d'un mécanisme de définition d'une interface on effectue un découpage (assez pénible) du programme en fichiers et on décore certaines fonctions et variables avec le mot static.

Une déclaration de variable ou de fonction qui est précédée par le mot static est visible seulement dans le fichier où se trouve la déclaration. Par ailleurs, une variable static déclarée dans une fonction est initialisée une seule fois et elle garde sa valeur d'un appel au suivant.

Ainsi elle se comporte comme une sorte de variable globale qui est visible seulement par la fonction.

Comme exemple, on considère la construction d'un module pour une pile d'entiers. On commence par définir un fichier stack.h qui contient les éléments suivants :

```
#ifndef STACK_H                    \\(1)
#define STACK_H                    \\(2)

typedef int item;                  \\(4)
extern void init_stack(int);       \\(5)
extern short empty_stack();        \\(6)
extern void insert_stack(item);    \\(7)
extern item elim_stack();          \\(8)

#endif                             \\(3)
```

Le fichier stack.h pourrait être importé par d'autres fichiers plusieurs fois et dans ce cas les lignes (1–3) assurent que les définitions dans le fichier stack.h seront prises en compte une seule fois. Ces lignes ne sont pas vraiment utiles dans l'exemple en question mais c'est une bonne pratique de les mettre dans les fichiers .h pour éviter des ennuis dans des situations plus compliquées.

Dans (4), on déclare le type item des éléments qui vont constituer la pile ; dans notre cas il s'agit d'entiers. Dans (5-8), on déclare les prototypes des fonctions pour la gestion de la pile qu'on qualifie de fonctions extern.

On associe au fichier stack.h un fichier stack.c qui contient la mise en oeuvre de la pile. Par exemple, le fichier stack.c pourrait être le suivant. Notez que dans (1) on inclut le fichier stack.h. Un fichier, disons user.c qui voudrait utiliser les fonctions de la pile devrait aussi contenir cette directive.

```
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>

#include "stack.h"                 \\(1)

static item * stack=NULL;
static int head=-1;
static int size=-1;

void init_stack(int m){
  if(head==-1){
  stack=malloc(sizeof(item)*m);
  head=0;
  size=m;
  }
}
short empty_stack(){
  if(head==0){return 1;}
  if(head>0){return 0;}
  assert(0);
}
void insert_stack(item d){
  assert(head<(size-1));
```

```
  if(head>=0){
    stack[head]=d;
    head++;
  }
}
item elim_stack(){
  assert(head>0);
  head--;
  return stack[head];
}
```

On a maintenant 3 fichiers à traiter : stack.h, stack.h, user.c. En principe, la commande cc -o user user.c stack.c suffit à produire un exécutable user. Cependant, il n'est pas rare de se trouver dans des situations où il y a beaucoup plus de fichiers. Dans ces cas, il est recommandé de déclarer les dépendances entre les fichiers dans un fichier Makefile et de laisser la commande make s'occuper de la compilation. Dans le cas en question, le contenu du fichier Makefile pourrait être le suivant :

```
CC=gcc
CFLAGS=-Wall -std=c11
LDLIBS= -lm
ALL = user
user : user.o stack.o
user.o : user.c
stack.o : stack.c
```

La ligne CC spécifie le compilateur, la ligne CFLAGS les paramètres de compilation, la ligne LDLIBS les bibliothèques à charger, la ligne ALL le nom de l'exécutable et les lignes suivantes expriment les dépendances. Les fichiers .o sont des fichiers intermédiaires entre le code source et l'exécutable. Ces fichiers sont générés à partir des fichiers source et il sont ensuite combinés (on dit aussi liées) pour produire l'exécutable. [1]

13.3 Applications

On discute deux exemples d'application des structures *pile* et *queue*.

Exemple 42 *On a vu dans la section 1.2 que l'interprétation d'un programme* C *utilise une pile de blocs d'activation : à chaque appel de fonction on empile le bloc de la fonction appelée et à chaque retour de fonction on dépile le bloc qui se trouve au sommet de la pile. En particulier, cette pile permet de comprendre le fonctionnement des fonctions* récursives *(chapitre 5).*

En principe, il est possible de se passer des appels récursifs mais le prix à payer est une gestion explicite de la pile. Pour illustrer la méthode on reprend l'exemple de la tour d'Hanoï (section 5.2).

```
void hanoi (int n, int p1, int p2){
  int p3=troisieme(p1,p2);
  if(n==1){imprimer(p1,p2);}
  else {hanoi(n-1,p1,p3); imprimer(p1,p2); hanoi(n-1,p3,p2);}
  return;}
```

1. Ceci est juste un petit aperçu des possibilités offerte par l'outil make.

Pour transformer cette fonction récursive *en une fonction* itérative *(avec des boucles mais sans appels récursifs), on va introduire une pile qui contient des triplets* (n, p, p') *où* n *est la hauteur de la tour qu'on veut déplacer et* p *et* p' *sont deux pivots différents. On va donc redéfinir le type* item *du fichier* stack.h *de la section précédente comme suit :*

```
struct han {int hauteur; int pivot1; int pivot2;};
typedef struct han item;
```

On est maintenant prêt à introduire la version itérative de la fonction hanoi. *Dans* (1) *on initialise une pile assez grande (exercice !), dans* (2) *on insère dans la pile le triplet qui correspond au problème initial, à partir de* (3)*, tant que la pile est non-vide, on extrait un problème et on distingue deux situations :*

- *si la tour a hauteur* 1 *on imprime directement la solution,*
- *sinon on empile trois sous-problèmes ; le lecteur remarquera que l'ordre d'empilement est* inversé *par rapport à l'ordre des appels récursifs dans la fonction* hanoi.

```
static void hanoi_it(int n, int p1, int p2){
  init_stack(2*n);                                    //(1)
  item d={n,p1,p2};
  insert_stack(d);                                    //(2)
  while(!(empty_stack())){                            //(3)
    item c=elim_stack();
    if (c.hauteur==1){imprimer(c.pivot1,c.pivot2);}
    else {
      int p3=troisieme(c.pivot1,c.pivot2);
      item b1={c.hauteur-1, p3, c.pivot2};
      insert_stack(b1);
      item b2={1,c.pivot1,c.pivot2};
      insert_stack(b2);
      item b3={c.hauteur-1, c.pivot1, p3};
      insert_stack(b3);}
  }
}
```

Exemple 43 *On considère* n *villes* $\{v_0, \ldots, v_{n-1}\}$ *et on s'intéresse au nombre minimum de vols qui sont nécessaires pour connecter la ville* v_0 *aux villes* $v_1, v_2, \ldots, v_{n-1}$*. On dispose d'un tableau de tableaux d'entiers* c *tel que*

$$\mathsf{c[i][j]} = \begin{cases} 1 & \textit{s'il y a un vol direct de } v_i \textit{ à } v_j \\ 0 & \textit{sinon.} \end{cases}$$

Le problème est de calculer un tableau d'entiers d *tel que* $d[i]$ *est le nombre minimum de vols nécessaires à connecter la ville* v_0 *à la ville* v_i*. On appelle ce nombre l'éloignement de* v_i *de* v_0*. Par convention, ce nombre est* 0 *si* $i = 0$ *et* INT_MAX *si* $i \neq 0$ *et il est impossible d'aller de* v_0 *à* v_i*. Un algorithme possible est le suivant.*

1. *On initialise le tableau* d *comme suit :*

$$\mathsf{d[i]} = \begin{cases} 0 & \textit{si } i = 0 \\ \mathsf{INT\_MAX} & \textit{sinon.} \end{cases}$$

2. *On initialise une* queue *qui contient la ville* 0.
3. *Tant que la* queue *n'est pas vide :*

(a) on extrait une ville v_i *de la queue ; soit* $d = \mathsf{d[i]}$.

(b) on calcule l'ensemble :

$$V = \{v_j \mid \mathsf{c[i][j]} = 1 \ \textit{et} \ \mathsf{d[j]} = \mathsf{INT\_MAX}\}$$

(c) pour tout v_j *dans* V*, on pose* $\mathsf{d[j]} = d + 1$ *et on insère* v_j *dans* queue.

L'intérêt de la structure queue *dans cet exemple est qu'elle nous permet d'examiner les villes accessibles par ordre d'éloignement croissant. Notez aussi que chaque ville est insérée dans la queue au plus une fois et qu'à cette occasion son éloignement est déterminé. Vérifiez que si l'on remplace la* queue *par une* pile*, l'algorithme décrit ci-dessus n'est pas correct.*

Chapitre 14

La structure de données tas (*heap*)

Soit H un ensemble fini d'éléments que l'on peut comparer avec un *ordre total*. On cherche une façon de représenter H qui nous permet d'effectuer (au moins) les *opérations* suivantes de façon efficace :

— *insertion* d'un élément dans H.

— *élimination* du plus grand élément de H.

Si l'on garde H totalement ordonné alors on peut extraire un élément en $O(1)$ mais l'insertion d'un élément est en $O(n)$. D'autre part, si on ignore l'ordre alors on peut insérer en $O(1)$ et extraire en $O(n)$. En moyenne, on s'attend à faire autant d'insertions que d'éliminations et donc les deux solutions demandent un temps linéaire dans le nombre d'éléments dans H. La structure *tas* (ou *heap* en anglais) [1] qu'on va introduire dans ce chapitre va nous permettre d'effectuer ces opérations en temps *logarithmique*.

Le tas est un premier exemple de *structure de données* non triviale. De telles structures sont un *outil essentiel* dans la conception d'algorithmes efficaces.

14.1 Arbres binaires

La clef pour obtenir un temps logarithmique est de stocker l'ensemble H dans un *arbre* de façon à ce que les opérations d'insertion et d'élimination demandent l'examen d'une seule branche de l'arbre. On obtient une borne logarithmique en observant que la taille de chaque branche est logarithmique dans le nombre d'éléments de l'arbre.

On commence par définir exactement ce qu'on entend par arbre. L'ensemble T des *arbres binaires* est défini *inductivement*. Si, par exemple, on veut définir des arbres binaires dont les *noeuds* ont une valeur *entière* on posera la définition suivante.

Définition 5 (arbres binaires) *L'ensemble T est le plus petit ensemble tel que :*

— $\mathsf{nil} \in T$ *(l'*arbre vide*).*

— *Si $t_1, t_2 \in T$ et $n \in \mathbf{Z}$ alors $(n, t_1, t_2) \in T$.*

De tels arbres se prêtent bien à une *représentation graphique*. Si $t = (n, t_1, t_n) \in T$, on dit que t_1 et t_2 sont respectivement *le sous-arbre gauche et droite* de t. Dans la représentation

1. La structure tas (*heap*) que l'on discute ici se nomme aussi *queue de priorité* et ne devrait pas être confondue avec la mémoire tas (*heap*) dont il est question dans l'exécution de programmes avec allocation dynamique ; il s'agit simplement d'un cas d'homonymie !

graphique, on associe à t un noeud qui contient la valeur n et qui est connecté par une arête gauche et une arête droite aux représentations graphiques de t_1 et t_2, respectivement. Le noeud associé à t est le *père* des noeuds associés au noeuds t_1 et t_2 (qui eux sont les fils). Le premier noeud généré dans cette construction est désigné en tant que *racine* de l'arbre. Par convention, on ne représente pas les arbres vides (nil) et on dit qu'un noeud qui n'a pas de sous-arbres (non-vides) est une *feuille*. Le noeud racine est une feuille si et seulement si l'arbre comporte un seul noeud. Typiquement, on dessine les arbres à l'envers, c'est-à-dire avec les feuilles en bas et la racine en haut.

Dans la suite un *arbre* est un arbre d'après la définition ci-dessus. Techniquement, il s'agit d'arbres *enracinés* (on désigne un noeud racine), binaires (un noeud a au plus deux fils), ordonnés (on distingue le fils gauche du fils droit) et avec des *valeurs* associées aux noeuds. [2]

Définition 6 (hauteur) *La* hauteur h *d'un arbre est le nombre d'arêtes qu'il faut traverser dans le chemin le plus long de la racine à une feuille.*

Proposition 2 *Un arbre de hauteur h a entre $(h+1)$ et $2^{(h+1)}-1$ noeuds.*

Preuve. Pour avoir un chemin avec h arêtes il faut $(h+1)$ noeuds. D'autre part, on maximise le nombre de noeuds en supposant que chaque noeud qui n'est pas une feuille a deux fils. On atteint ainsi la borne supérieure (preuve par récurrence sur h). □

Définition 7 (arbre plein) *Un arbre est* plein *si tous les noeuds qui ne sont pas des feuilles ont deux fils.*

Définition 8 (arbre complet) *Un arbre est* complet *s'il est plein et toutes les feuilles sont à la même profondeur (la longueur du chemin de la racine à une feuille est constant).*

Définition 9 (positions) *On peut compter les* positions *des noeuds d'un arbre de la façon suivante (par convention, on compte de* 1*) :*

Niveau	*Position*
0	1
1	2, 3
2	4, 5, 6, 7
3	8, 9, 10, 11, 12, 13, 14, 15
...	...
h	$2^h, \ldots, (2^{h+1}-1)$

Définition 10 (arbre quasi-complet) *Un arbre avec n noeuds est* quasi-complet *si ses noeuds occupent (exactement) les positions $1, \ldots, n$.*

On peut représenter un arbre quasi-complet avec des pointeurs. Si l'on connaît le nombre maximal de noeuds dans l'arbre une solution plus économe en mémoire est d'utiliser un tableau. En effet, un *arbre quasi-complet avec n noeuds* est en correspondance bijective avec un *tableau de n éléments* dont les cellules $1, \ldots, n$ contiennent les valeurs associées aux noeuds :

2. On utilisera cette notion d'arbre aussi dans le chapitre 18 (arbres binaires de recherche) et la section 21.2 (compression de Huffman). Par contre dans les chapitres 23 et 24 (sur les graphes) on introduira une autre notion d'arbre.

— Les *fils* du noeud en position i (s'ils existent) sont dans les positions $2i$ et $2i+1$.
— Le *père* d'un noeud en position $i > 1$ est en position $i/2$.
— Un arbre quasi-complet avec n noeuds a *hauteur* $h = \lfloor \log_2(n) \rfloor$ et ses *feuilles* sont aux positions $(n/2)+1, \ldots, n$.

14.2 Tas et opérations sur le tas

Définition 11 (tas) *Un* tas *est un arbre quasi-complet où chaque noeud a une valeur* supérieure ou égale *à celle des fils.*

Remarque 11 *Une définition duale est possible où l'on stipule que le père a une valeur inférieure ou égale à celle des fils. Si l'on veut distinguer les deux situations on parlera de* max-tas *et de* min-tas*. Dans la suite on se focalise sur les max-tas (la racine contient la valeur la plus grande). Tout ce qu'on fait peut être adapté de façon évidente aux min-tas.*

On fait l'hypothèse que l'on dispose d'un *tableau a* avec n cellules numérotées de 1 à n et d'une variable m qui enregistre le nombre de cellules occupées. On a donc $0 \leq m \leq n$ et on peut représenter de cette façon tous les arbres quasi-complets avec m éléments. On décrit maintenant la mise-en-oeuvre des opérations principales et leur complexité.

Insertion Possible seulement si $m < n$. On incrémente m, on ajoute l'élément inséré au fond du tableau et on le fait remonter autant que nécessaire en le comparant à son père. La comparaison et éventuellement l'échange avec le père se fait en $O(1)$. Le nombre de comparaisons est borné par la hauteur de l'arbre. On a donc un coût $O(\log m)$.

Élimination Possible seulement si $0 < m$. On décrémente m et on récupère l'élément en position 1. Si $m > 0$ on place l'élément en position m en position 1 et on le fait descendre autant que nécessaire dans l'arbre en le permutant avec son fils le plus grand. A nouveau, la comparaison et éventuellement l'échange avec le fils se fait en $O(1)$. Le nombre de comparaisons est borné par la hauteur de l'arbre. On a donc un coût $O(\log m)$.

Dans la mise en oeuvre de l'opération d'élimination, il convient de définir une fonction *récursive heapify* qui effectue le travail suivant : en supposant que $t = (n, t_1, t_2)$ est un arbre quasi-complet et t_1, t_2 sont des tas elle transforme t dans un tas. En pratique, la fonction *heapify* prend en argument l'indice i de la racine de l'arbre quasi-complet et suppose que les sous-arbres de racine $2 \cdot \mathsf{i}$ et $2 \cdot \mathsf{i} + 1$, s'ils existent, sont des tas. Ainsi dans l'opération d'élimination, la phase de descente de l'élément en position 1 est réalisée par un appel heapify(1).

Plus en général, la fonction *heapify* peut être utilisée pour programmer une fonction *build-heap* qui transforme un tableau de m éléments en un tas. On sait que les éléments en position $m/2+1, \ldots, m$ sont des feuilles (et donc des tas). Il suffit donc d'appliquer la fonction *heapify* aux éléments qui se trouvent aux positions $m/2, m/2-1, \ldots, 1$. Dans ce cas, à chaque application de *heapify*(i) on sait que les sous-arbres dont les racines sont aux positions $2i$ et $2i+1$ sont déjà des tas et on fait en sorte que le sous-arbre de racine i le devienne aussi. Cette opération *build-heap* effectue $m/2$ appels a la fonction *heapify*. Le coût de chaque appel dépend de la hauteur de l'arbre. A priori, on sait que cette hauteur est bornée par $log_2(m)$ et donc le coût de *build-heap* est $O(m \cdot log(m))$. Cependant, on va voir qu'une analyse plus fine permet d'avoir une borne $O(m)$.

Proposition 3 *L'application de la fonction build-heap à un tableau de m éléments a un coût* $O(m)$.

PREUVE. Comme on l'a déjà remarqué, le coût de *heapify* est au plus la hauteur h de l'arbre et $h \leq log_2(m)$. Mais cette borne n'est pas très satisfaisante car la plus part des noeuds ne sont *pas très hauts*!

Niveau	Position	Coût
0	1	h
1	$2, 3$	$(h-1)$
2	$4, 5, 6, 7$	$(h-2)$
3	$8, 9, 10, 11, 12, 13, 14, 15$	$(h-3)$
$\ldots$	$\ldots$	$\ldots$
h	$2^h, \ldots, (2^{h+1}-1)$	0

On doit évaluer :

$$\Sigma_{i=0,\ldots,h}\ 2^i(h-i) \quad = \quad 2^h\Sigma_{i=0,\ldots,h}\ \tfrac{(h-i)}{2^{(h-i)}} \quad = \quad 2^h\Sigma_{i=0,\ldots,h}\ \tfrac{i}{2^i}\ .$$

On sait que $\Sigma_{i=0,\ldots,h}\ \frac{1}{2^i}$ est une *constante* (série géométrique). On montre que $\Sigma_{i=0,\ldots,h}\ \frac{i}{2^i}$ est une *constante* aussi. On sait que pour $0 \leq x < 1$:

$$\Sigma_{i=0,\ldots,\infty}\ x^i = \frac{1}{1-x}\ .$$

Si on *dérive*, on obtient (un théorème d'analyse assure ici que la dérivée de la somme est égale à la somme des dérivées) :

$$\Sigma_{i=1,\ldots,\infty}\ i \cdot x^{i-1} = \frac{1}{(1-x)^2}\ .$$

Si on *multiplie* par x et on *pose* $x = \frac{1}{2}$:

$$\Sigma_{i=0,\ldots,h}\ \frac{i}{2^i}\ \leq\ \Sigma_{i=1,\ldots,\infty}\ \frac{i}{2^i} = \frac{1/2}{(1-1/2)^2} = 2\ .$$

□

Remarque 12 *Dans notre analyse, on s'est limité à l'efficacité de chaque opération dans le pire des cas. Pour d'autres structures de données, d'autres analyses peuvent être plus pertinentes. Par exemple, on peut s'intéresser à l'efficacité d'une suite de n opérations (on parle d'*analyse amortie*). La raison est qu'il peut y avoir une dépendance entre les opérations. Par exemple, on peut imaginer qu'une opération coûteuse peut avoir lieu seulement si beaucoup d'opérations pas chères ont eu lieu auparavant.*

14.3 Applications

Tri par tas (*heapsort*)

En utilisant la structure tas on peut concevoir un (autre) algorithme qui trie un tableau de n éléments en $O(n \cdot log(n))$ dans le pire des cas.

— L'algorithme prend en entrée un tableau `a` avec n éléments aux positions $1, \ldots, n$.

— On appelle la fonction *build-heap* sur ce tableau avec un coût : $O(n)$. L'élément plus grand se trouve alors en a[1]. On pose $m = n$ (m est le nombre d'éléments dans le tas).
— On itère $n - 1$ fois pour un coût total qui est $O(nlogn)$:
  1. on échange a[1] avec a[m].
  2. on décrémente m et on applique heapify(1).
— A la fin de l'itération les éléments sont triés par ordre croissant.

Queue de priorité

Dans la *simulation* d'un système à événements discrets, chaque événement a une *date* (la date à laquelle l'événement doit avoir lieu). On place les événements dans un *min-heap* (le plus petit est au sommet).

Un *pas de simulation* consiste à éliminer du tas l'événement au sommet (le plus proche dans le futur) et à insérer dans le tas un nombre (qu'on suppose borné) de nouveaux événements (avec les nouvelles dates). Ainsi, dans un tas de taille m, chaque pas de simulation prend $O(\log m)$.

Codage, graphes

On trouvera d'autres utilisations de la structure tas dans la suite du cours. Notamment dans la construction de l'arbre de codage de Hufmann (section 21.2) et dans la recherche des plus courts chemins dans un graphe (section 24.3).

Chapitre 15

Diviser pour régner et relations de récurrence

Diviser pour régner (*divide and conquer* en anglais, *divide et impera* en latin) est une stratégie générale pour la conception d'algorithmes.

— Si le problème est *petit* le résoudre directement.
— Sinon, *découper* le problème en sous-problèmes de taille comparable (si possible).
— *Résoudre* les sous-problèmes en appliquant la même stratégie.
— *Dériver* une solution pour le problème de départ.

L'analyse de complexité d'algorithmes conçus en suivant cette stratégie revient à expliciter la fonction qui satisfait une certaine relation de récurrence. Dans ce chapitre on donne une méthode pour résoudre une certaine classe de relations de récurrence et on décrit des algorithmes qui appliquent la stratégie diviser pour régner. Un autre exemple majeur sera discuté dans le chapitre 16.

15.1 Problèmes et relations de récurrence

On suppose que l'on peut exprimer la solution d'un problème de taille n en fonction de :

1. La solution de a *problèmes* de *taille* n/b ($a \geq 1$ et $b > 1$)
2. Un *travail de division et combinaison* des solutions de sous-problèmes qui coûte $O(n^c)$ avec $c \geq 0$.

La complexité $C(n)$ de l'algorithme sur une entrée de taille n est alors déterminée par une *récurrence* de la forme :

$$\begin{array}{rcl} C(n) & = & a \cdot C(n/b) + O(n^c) \ , \\ C(0) & = & O(1) \ . \end{array} \tag{15.1}$$

Notation La notation $O(g)$ définie dans le chapitre 6 dénote un ensemble de fonctions qui sont bornées au sens asymptotique par la fonction g. En pratique, on abuse souvent cette notation. Par exemple, on écrit $C(n) = O(g)$ pour dire que la fonction $C(n)$ est dans $O(g)$. On écrit aussi $C(n) + O(g)$ pour indiquer une fonction qui est bornée au sens asymptotique par une fonction de la forme $C(n) + k \cdot g(n)$. Ainsi la récurrence (15.1) ci-dessus dit qu'ils

existent $n_0, k_1, k_2 \geq 0$ tels que pour tout $n \geq n_0$:

$$\begin{array}{rcl} C(n) & \leq & a \cdot C(n/b) + k_1 \cdot n^c \\ C(0) & \leq & k_2 \ . \end{array}$$

Exemple 44 *Considérons la recherche dichotomique dans un tableau ordonné.*
- *Si le tableau a* 1 élément *on le compare à l'élément recherché et on termine.*
- *Sinon, on compare l'élément recherché avec l'élément au* milieu *du tableau.*
- *S'ils sont* égaux *on termine.*
- *Sinon on* itère *la recherche sur une* moitié du tableau.

On a donc :

$$C(n) = C(n/2) + O(1) \ .$$

Exemple 45 *On considère un algorithme de tri par fusion (*mergesort*) qui opère sur un tableau.*
- *Si le tableau a* taille 1*, le tableau est trié.*
- *Sinon, on sépare le tableau en* deux parties égales *et on les trie.*
- *Ensuite on fait une* fusion *des deux tableaux triés.*

La phase de division prend $O(1)$ et la phase de fusion $O(n)$. On dérive donc une relation de récurrence :

$$C(n) = 2 \cdot C(n/2) + O(n) \ .$$

Exemple 46 *On veut* multiplier *x et y entiers sur n chiffres (pour simplifier supposons n pair). L'algorithme usuel est* quadratique *en n. On peut écrire x et y comme :*

$$\begin{array}{rcl} x & = & a \cdot 10^m + b \\ y & = & c \cdot 10^m + d \ , \end{array}$$

où a, b, c, d sont maintenant des entiers sur $m = n/2$ chiffres*. On remarque :*

$$x \cdot y = ac \cdot 10^{2m} + (ad + bc)10^m + bd \ .$$

Ceci suggère un algorithme diviser pour régner *où une multiplication de taille n est réduite à 4 multiplications de taille $n/2$ plus des additions et des décalages (pour implémenter la multiplication par une puissance de* 10*). On obtient donc :*

$$C(n) = 4 \cdot C(n/2) + O(n) \ .$$

Hélas, on a toujours une complexité quadratique *(technique de preuve à suivre) et un* algorithme plus compliqué*. Probablement la* mise-en-oeuvre *donnera un algorithme* moins efficace *que l'algorithme standard... Mais on peut faire* mieux *! Voici une vieille* remarque *(Gauss) qui a été exploitée par A. Karatsuba [KO62] :*

$$(ad + bc) = (a + b)(c + d) - ac - bd \ .$$

On peut donc calculer le facteur de 10^m avec 1 multiplication *(plus 4 additions) au lieu de* 2 multiplications*. Ce qui donne :*

$$C(n) = 3 \cdot C(n/2) + O(n) \ .$$

Cette fois il y a un gain significatif *(justification à suivre) et une bonne mise en oeuvre donne une méthode de multiplication plus efficace pour des nombres assez grands (environ* 500 *bits).*

Exemple 47 *C'est un peu la* même histoire *que pour la multiplication d'entiers mais en* dimension 2 *! On veut* multiplier deux matrices A, B *de dimension* $n \times n$ *(n pair). L'*algorithme standard *est* $O(n^3)$. *Une stratégie* diviser pour régner *commence par décomposer* A *et* B *en :*

$$A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \quad B = \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$$

où A_{ij} *et* B_{ij} *ont dimension* $n/2 \times n/2$. *Ensuite on* calcule *:*

$$C = \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix}$$

où :

$$\begin{aligned} C_{11} &= A_{11} \cdot B_{11} + A_{12} \cdot B_{21} \\ C_{12} &= A_{11} \cdot B_{12} + A_{12} \cdot B_{22} \\ C_{21} &= A_{21} \cdot B_{11} + A_{22} \cdot B_{21} \\ C_{22} &= A_{21} \cdot B_{12} + A_{22} \cdot B_{22} \ . \end{aligned}$$

On vérifie que : $C = A \cdot B$. *On a donc un* coût *:*

$$C(n) = 8 \cdot C(n/2) + O(n^2) \ .$$

Encore une fois, ceci est plus compliqué *que l'algorithme standard et a la* même complexité asymptotique. *Cependant, V. Strassen [Str69] a montré que l'on peut s'en sortir avec* 7 *multiplications (détails non-triviaux dans [CLRS09]). On a donc :*

$$C(n) = 7 \cdot C(n/2) + O(n^2) \ .$$

Ce qui mène à une amélioration significative *de la complexité asymptotique et même à un* algorithme pratique *pour* n *de l'ordre de* 10^2 *(voir [CLRS09]). Une petite course à la* meilleure borne asymptotique *a suivi. Actuellement le* record *est autour de* $O(n^{2,3})$ *mais l'algorithme en question n'est pas du tout pratique ! Rappelons au passage que si l'on travaille avec les flottants, il faut aussi évaluer la* stabilité numérique *de l'algorithme.*

15.2 Solution de relations de récurrence

Proposition 4 *Pour borner la solution de la relation de récurrence :*

$$C(n) = a \cdot C(n/b) + O(n^c), \quad C(0) = O(1),$$

il suffit de considérer le ratio *:*

$$r = \frac{a}{b^c} \ .$$

A savoir :

$$\begin{aligned} &\textit{si } r = 1 \quad \textit{alors } C(n) = O(n^c \cdot \log n) \ , \\ &\textit{si } r < 1 \quad \textit{alors } C(n) = O(n^c) \ , \\ &\textit{si } r > 1 \quad \textit{alors } C(n) = O(n^{\log_b(a)}) \ . \end{aligned}$$

Remarque 13 *Ce qu'on présente est une* version a-b-c *(ou* $\frac{a}{b^c}$*) d'un théorème plus général dû à* Akra-Bazzi *[AB98] qu'on appelle aussi* master theorem.

Avant de procéder avec la preuve, considérons l'application de la proposition aux exemples.

Dichotomie $a = 1, b = 2, c = 0, r = 1$	$C(n) = 1 \cdot C(n/2) + O(1)$ $C(n) = O(\log n)$
Fusion $a = 2, b = 2, c = 1, r = 1$	$C(n) = 2 \cdot C(n/2) + O(n)$ $C(n) = O(n \log n)$
Karatsuba $a = 3, b = 2, c = 1, r > 1$	$C(n) = 3 \cdot C(n/2) + O(n)$ $C(n) = O(n^{\log_2(3)}) \approx O(n^{1,6})$
Strassen $a = 7, b = 2, c = 2, r > 1$	$C(n) = 7 \cdot C(n/2) + O(n^2)$ $C(n) = O(n^{\log_2(7)}) \approx O(n^{2,8})$
Plus rare $a = 2, b = 2, c = 2, r < 1$	$C(n) = 2 \cdot C(n/2) + O(n^2)$ $C(n) = O(n^2)$

Preuve de la proposition 4

Pour simplifier la notation, on suppose que :
— n est une puissance de b.
— k borne les constantes du *cas terminal* et du *travail de division et de combinaison*. On a donc :

$$\begin{aligned} C(n) &\leq a \cdot C(n/b) + k \cdot n^c \ , \\ C(0) &\leq k \ . \end{aligned}$$

Considérons maintenant le travail de division et combinaison qu'on effectue à chaque niveau :

niveau	travail
0	$a^0 \cdot k \cdot n^c$
1	$a^1 \cdot k \cdot (n/b^1)^c$
$\cdots$	$\cdots$
j	$a^j \cdot k \cdot (n/b^j)^c$
$\cdots$	$\cdots$
$\log_b(n)$	$a^{\log_b(n)} \cdot k \cdot (n/b^{\log_b(n)})^c$

Il en suit que le *travail* t_j au *niveau* j est :

$$t_j = k \cdot n^c \cdot r^j$$

où $r = \frac{a}{b^c}$. Le *ratio* fait dont son apparition ! On distingue maintenant les 3 cas.

$r = 1$: travail constant à chaque niveau On a :

$$t_j = k \cdot n^c \ .$$

En additionnant le ($\log_b(n) + 1$ niveaux) on obtient :

$$C(n) = O(n^c \log n) \ .$$

$r < 1$: le travail du niveau 0 domine On a :

$$\begin{aligned} \Sigma_{j=0,\ldots,\log_b(n)} t_j &= k \cdot n^c \cdot \Sigma_{j=0,\ldots,\log_b(n)} r^j \\ &\leq k \cdot n^c \cdot \tfrac{1}{1-r} \ . \end{aligned}$$

Donc :

$$C(n) = O(n^c) \ .$$

$r > 1$ **: le travail des feuilles domine** D'abord on remarque pour $h = \log_b(n)$:

$$\Sigma_{j=0,\ldots,h} r^j = \frac{r^{h+1} - 1}{r - 1} \leq r^h \frac{r}{r - 1} \ .$$

Donc pour $k' = r/(r-1)$ on a :

$$\Sigma_{j=0,\ldots,\log_b(n)} t_j \leq k \cdot n^c \cdot r^h \cdot k' \ .$$

En explicitant h et r :

$$k \cdot k' \cdot n^c \cdot \left(\frac{a}{b^c}\right)^{\log_b(n)} = k \cdot k' \cdot n^c \cdot \frac{a^{\log_b(n)}}{n^c} \ .$$

Rappel : $\log_a(x) = \frac{\log_b(x)}{\log_b(a)}$. Donc :

$$C(n) = O(a^{\log_b(n)}) = O(n^{\log_b(a)}) \ .$$

Notez que $a^{\log_b(n)}$ est le *nombre de feuilles.* □

Remarque 14 *La proposition 4* s'applique *si l'on divise un problème de taille n dans un nombre a de sous-problèmes qui ont la même taille n/b. Elle* ne s'applique pas*, par exemple, au* tri rapide *(voir section 17.2) car les sous-problèmes n'ont pas forcement la même taille.*

Exercice 20 *On peut représenter un nombre entier de taille arbitraire comme une liste de chiffres en base B. Par simplicité, supposons $B = 10$. Programmez les opérations d'addition et multiplication de l'école primaire ainsi que la multiplication de Karatsuba et déterminez le nombre de chiffres nécessaires pour que la multiplication de Karatsuba soit plus efficace que la multiplication de l'école primaire en pratique.*

Chapitre 16

Transformée de Fourier rapide

Un polynôme peut être représenté par ses coefficients ou par un ensemble de points. On peut voir la transformée (discrète) de Fourier comme une méthode pour passer de la représentation par coefficients à celle par points alors que la transformée *inverse* de Fourier passe des points aux coefficients. Un algorithme direct permet de mettre en oeuvre ces opérations en $O(n^2)$. En choisissant les points comme les racines n-aires de l'unité, il est possible, en suivant une stratégie diviser pour régner (voir chapitre 15), de réduire la complexité à $O(n \cdot log(n))$ [CT65]. Dans ce cas, on parle de transformée (ou transformée inverse) *rapide* (*Fast Fourier Transform* ou *FFT* en anglais). Cet algorithme joue un rôle très important, par exemple, dans le traitement numérique du signal.

16.1 Polynômes et matrice de Vandermonde

Soit $p(x) = \Sigma_{k=0,\ldots,n-1} a_i x^i$ un polynôme sur un corps. On peut évaluer un polynôme dans un point en effectuant $O(n)$ multiplication et additions. En particulier, on peut utiliser la règle de Horner :

$$p(x) = a_0 + x(a_1 + x(a_1 + \cdots + (xa_n)\cdots)) \tag{16.1}$$

Il en suit qu'on peut évaluer $p(x)$ en n points $x_0, \ldots, x_{n-1}$ en $O(n^2)$.

Définition 12 (matrice Vandermonde) *La matrice de Vandermonde V_n pour les points $x_0, \ldots, x_{n-1}$ est définie par :*

$$\begin{bmatrix} 1 & x_0 & x_0^2 & \cdots & x_0^{n-1} \\ 1 & x_1 & x_1^2 & \cdots & x_1^{n-1} \\ \cdots & \cdots & \cdots & \cdots & \\ 1 & x_{n-1} & x_{n-1}^2 & \cdots & x_{n-1}^{n-1} \end{bmatrix} \tag{16.2}$$

Des manipulations standards d'algèbre linéaire permettent d'expliciter le déterminant de la matrice V_n.

Fait 1 *Le déterminant de la matrice de Vandermonde V_n est :*

$$det(V_n) = \Pi_{0 \leq i < j \leq (n-1)} (x_j - x_i) \ . \tag{16.3}$$

Il suit que si les points $x_0, \ldots, x_{n-1}$ sont différents alors la matrice V_n est inversible.

Proposition 5 *Soient (x_k, y_k) des couples de points pour $k = 0, \ldots, n-1$ et avec $x_i \neq x_j$ si $i \neq j$. Alors il existe unique un polynôme $p(x)$ de degré au plus $n-1$ tel que $p(x_k) = y_k$ pour $k = 0, \ldots, n-1$.*

PREUVE. L'assertion que $p(x_k) = y_k$ pour $k = 0, \ldots, n-1$ est équivalente à la condition $V_n a = y$, où V_n est la matrice de Vandermonde relative aux points $x_0, \ldots, x_{n-1}$, $y = (y_0, \ldots, y_{n-1})$ et $a = (a_0, \ldots, a_{n-1})$ sont les coefficients du polynôme à déterminer. Comme V_n est inversible on doit avoir $a = (V_n)^{-1} y$. □

Il est possible d'expliciter le polynôme en question en utilisant l'interpolation de Lagrange.

Définition 13 (polynôme interpolant) *Soient (x_k, y_k) des couples de points pour $k = 0, \ldots, n-1$ et avec $x_i \neq x_j$ si $i \neq j$. On définit le polynôme interpolant par :*

$$\ell(x) = \Sigma_{i=0,\ldots,n-1} y_i \, \frac{\Pi_{j \neq i}(x - x_j)}{\Pi_{j \neq i}(x_i - x_j)} \ .$$

Proposition 6 *Le polynôme $\ell(x)$ a degré au plus $(n-1)$ et il satisfait : $\ell(x_i) = y_i$ pour $i = 0, \ldots, (n-1)$.*

PREUVE. Il est clair que le degré est au plus $n-1$ et on vérifie que :

$$\frac{\Pi_{j \neq i}(x - x_j)}{\Pi_{j \neq i}(x_i - x_j)} = \begin{cases} 1 & \text{si } x = x_i \\ 0 & \text{si } x = x_k \neq x_i \ . \end{cases}$$

□

On peut aussi dériver le fait qu'un polynôme non-nul de degré $n-1$ a au plus $n-1$ racines différentes.

Proposition 7 *Un polynôme $p(x)$ de degré $n-1$ qui n'est pas nul partout admet au plus $n-1$ points $x_1, \ldots, x_{n-1}$ tels que $p(x_k) = 0$ pour $k = 1, \ldots, n-1$.*

PREUVE. Si on avait n points $x_0, \ldots, x_{n-1}$ alors on pourrait construire la matrice de Vandermonde V_n relativement à ces points et dériver $V_n a = 0$ où $a = (a_0, \ldots, a_{n-1})$ sont les coefficients du polynôme. Il en suit que $a = (V_n)^{-1} 0 = 0$ contre l'hypothèse que $p(x) \neq 0$. □

Opérations et représentation de polynômes

On peut représenter un polynôme de degré $n-1$ par ses coefficients $a_0, \ldots, a_{n-1}$ ou par sa valeur dans n points $(x_0, y_0), \ldots, (x_{n-1}, y_{n-1})$. Il est possible de passer d'une représentation à l'autre en $O(n^2)$. En particulier, la *transformée de Fourier* est le passage des coefficients aux points ce qui revient à calculer $y = V_n a$ où V_n est la matrice de Vandermonde pour les points $x_0, \ldots, x_{n-1}$ et $a = (a_0, \ldots, a_{n-1})$ est le vecteur des coefficients. La *transformée de Fourier inverse* est le passage des points aux coefficients ce qu'on peut faire en calculant les coefficients du polynôme interpolant (définition 13).

On discute les avantages et les inconvénients de ces représentations par rapport à 3 opérations fondamentales : la somme, l'évaluation dans un point et le produit.

Somme Les deux représentations permettent de calculer la somme de deux polynômes en temps linéaire : on additionne les coefficients et les ordonnées des points, respectivement.

Évaluation L'évaluation d'un polynôme dans un point est possible en temps linéaire avec la représentation par coefficients (règle de Horner). Dans la représentation par points, on peut évaluer dans un point le polynôme interpolant de Lagrange mais ce calcul est $O(n^2)$.

Produit Le produit de deux polynômes est possible en $O(n)$ avec la représentation par points (on multiplie les ordonnées des points). A noter que le produit de deux polynômes de degré au plus $n-1$ a degré au plus $2n-2$ et que pour déterminer ce polynôme il faut connaître sa valeur en $2n-1$ points. Donc pour calculer le produit 'par points' il faut connaître $2n-1$ points des polynômes à multiplier.

Dans la représentation par coefficients, le calcul des coefficients du polynôme produit est lié à l'opération de *convolution*. Si $(a_0, \ldots, a_{n-1})$ et $(b_0, \ldots, b_{n-1})$ sont les coefficients de deux polynômes de degré au plus $n-1$ alors les coefficients du polynôme produit de degré au plus $2n-2$ sont :

$$c_k = \Sigma\{a_i \cdot b_j \mid i+j=k, 0 \leq i,j \leq (n-1)\} \qquad k = 0, \ldots, 2n-2 \ . \tag{16.4}$$

Ce calcul est $O(n^2)$.

16.2 Le cercle unitaire complexe

On considère maintenant le corps des nombres complexes et on dénote par $\mathbf{i}$ la valeur de coordonnées $(0,1)$ sur le plan complexe, à savoir une des racines carrées de -1. Si $x = a + \mathbf{i}b$ est un nombre complexe on dénote par $\overline{x} = a - \mathbf{i}b$ son conjugué. Les points qui se trouvent sur le cercle de centre $(0,0)$ et rayon 1 s'expriment par :

$$\cos\theta + \mathbf{i}\sin\theta \ . \tag{16.5}$$

La valeur θ représente l'angle qui détermine le point sur le cercle. Ainsi la fonction est périodique de période 2π. En suivant Euler, la fonction s'exprime aussi comme :

$$e^{\mathbf{i}\theta} = \cos\theta + \mathbf{i}\sin\theta \ .$$

La multiplication de deux points sur le cercle revient à additionner les angles :

$$e^{\mathbf{i}\theta_1} \cdot e^{\mathbf{i}\theta_2} = e^{\mathbf{i}(\theta_1+\theta_2)} \ . \tag{16.6}$$

La valeur complexe $(1,0)$ est donc l'unité pour la multiplication :

$$e^{\mathbf{i}(2\pi)n} = 1 \qquad n \in \mathbf{Z} \ . \tag{16.7}$$

Par ailleurs, chaque élément a une inverse :

$$e^{\mathbf{i}\theta} \cdot e^{\mathbf{i}(-\theta)} = 1 \ . \tag{16.8}$$

On remarquera que :

$$e^{\mathbf{i}(-\theta)} = \cos(-\theta) + \mathbf{i}\sin(-\theta) = \cos(\theta) - \mathbf{i}\sin(-\theta) \ ,$$

est le point symétrique par rapport à l'abscisse et correspond au conjugué de $e^{\mathbf{i}\theta}$.

Il suit de ces considérations que les points $e^{\mathbf{i}\theta}$ forment un *groupe abelien*.

On peut construire un *sous-groupe* avec n éléments en considérant les points de la forme :

$$\omega^0, \ldots, \omega^{(n-1)} \qquad \text{où } \omega = e^{\mathbf{i}(2\pi)/n} \ .$$

On remarquera que $(\omega^k)^n = 1$ pour $k = 0, \ldots, n-1$. En d'autres termes, les ω^k sont exactement les racines du polynôme $p(x) = x^n - 1$. Par ailleurs, chaque ω^k a une inverse multiplicative :

$$(\omega^k)(\omega^{-k}) = (\omega^k)(\omega^{(n-k)}) = 1 \ .$$

Proposition 8 *Soit $n = 2^h$ avec $h \geq 1$ et soit $X = \{\omega^k \mid k = 0, 1, \ldots, (n-1)\}$ l'ensemble des racines n-aires de l'unité. Alors si l'on pose :*

$$X^2 = \{x^2 \mid x \in X\} = \{\omega^{2k} \mid k = 0, 1, \ldots, n-1\} \ ,$$

on a que $\sharp X^2 = \sharp X/2 = n/2$.

Ainsi on a 2 racines quadratiques, 4 racines cubiques,...

16.3 Transformée rapide

On va montrer qu'en choisissant les n points comme les racines n-aires de l'unité il est possible de calculer la transformée de Fourier en $O(nlog(n))$. Pour obtenir ce résultat, on utilise une technique *diviser pour régner* : pour calculer un polynôme de degré au plus $n-1$ on va évaluer deux polynômes de degré au plus $n/2-1$ qui sont construits en prenant les coefficients pairs et impairs, respectivement du polynôme de départ :

$$p(x) \ = \Sigma_{k=0,\ldots,n-1} a_k x^k \ = p_0(x^2) + x \cdot p_1(x^2) \quad \text{où} \quad \left\{ \begin{array}{ll} p_0(x) & = \Sigma_{k=0,\ldots,n/2-1} a_{2k} x^k \\ p_1(x) & = \Sigma_{k=0,\ldots,n/2-1} a_{2k+1} x^k \ . \end{array} \right.$$

Cette décomposition est toujours possible mais elle est avantageuse seulement si on peut réduire le nombre de points sur lesquels le polynômes p_0 et p_1 doivent être évalués. En particulier, si on pose $n = 2^h$ et on prend X comme l'ensemble des racines n-aires de l'unité on sait que $\sharp X^2 = \sharp X/2$. Donc pour évaluer un polynôme de degré $n-1$ sur les racines n-aires de l'unité il suffit d'évaluer deux polynômes de degré $n/2-1$ sur les racines $n/2$-aires de l'unité et ensuite combiner les résultats avec un coût linéaire. On a donc une relation de récurrence :

$$C(n) = 2 \cdot C(n/2) + n \ ,$$

et on sait que $C(n)$ est $O(n \cdot log(n))$ (chapitre 15). A noter qu'il est essentiel de prendre comme points les racines n-aires de l'unité. A défaut on ne pourrait pas garantir que les carrés de ces valeurs constituent un ensemble de cardinale $n/2$.

Transformée inverse rapide

Il se trouve qu'on peut appliquer la même méthode pour calculer la transformée inverse. Ce fait repose sur une caractérisation de la matrice inverse de Vandermonde. Soit V_n la matrice de Vandermonde construite à partir des points :

$$x_k = \omega^k, \qquad k = 0, \ldots, n-1 \ .$$

Soit $a = (a_0, \ldots, a_{n-1})$ le vecteur des coefficients d'un polynôme de degré au plus $n-1$, soit $x = (x_0, \ldots, x_{n-1})$ le vecteur des points et soit $y = (y_0, \ldots, y_{n-1})$ le vecteur des images des points. On a :

$$V_n a = y \ . \tag{16.9}$$

Proposition 9 *Dans les hypothèses ci dessous, la matrice inverse de V_n est*

$$(V_n)^{-1} = \frac{1}{n}\overline{V_n}$$

où $\overline{V}_n$ est la matrice obtenue en prenant les conjugué de tous les éléments de V_n.

PREUVE. On a pour $j, k, \ell \in \{0, 1, \ldots, n-1\}$:

$$V_n[j,k] = \omega^{jk} \ , \qquad \overline{V}_n[k,\ell] = \omega^{-k\ell} \ .$$

On observe que :

$$\Sigma_{k=0,\ldots,n-1}\omega^{jk}\omega^{-kj} = \Sigma_{k=0,\ldots,n-1}\omega^{(jk-jk)} = \Sigma_{k=0,\ldots,n-1}1 = n \ ,$$

et que pour $j \neq \ell$:

$$\begin{aligned}
\Sigma_{k=0,\ldots,n-1}\omega^{jk}\omega^{-k\ell} &= \Sigma_{k=0,\ldots,n-1}(\omega^{(j-\ell)})^k \\
&= \tfrac{1-(\omega^{(j-\ell)})^n}{1-\omega^{(j-\ell)}} \\
&= \tfrac{1-\omega^{n(j-\ell)}}{1-\omega^{(j-\ell)}} \\
&= \tfrac{0}{1-\omega^{(j-\ell)}} \\
&= 0 \ .
\end{aligned}$$

Donc $V_n \cdot \overline{V}_n = n \cdot I_n$, où I_n est la matrice identité de dimension n. Il suit que : $(V_n)^{-1} = \frac{1}{n}\overline{V}_n$. □

La matrice $\overline{V}_n$ est la matrice de Vandermonde relativement aux points $x_k = \omega^k$ pour $k = 0, \ldots, n-1$. Ces points sont aussi les racines n-aires de l'unité. La différence entre V_n et $\overline{V}_n$ est que dans V_n on énumère les racines à partir de 1 en sens antihoraire alors que dans $\overline{V}_n$ on procède en sens horaire.

Exemple 48 *Pour $n = 4$, les matrices V_n et $\overline{V}_n$ sont, respectivement :*

$$V_n = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & \omega & \omega^2 & \omega^3 \\ 1 & \omega^2 & 1 & \omega^2 \\ 1 & \omega^3 & \omega^2 & \omega \end{bmatrix} \ , \qquad \overline{V}_n = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & \omega^3 & \omega^2 & \omega \\ 1 & \omega^2 & 1 & \omega^2 \\ 1 & \omega & \omega^2 & \omega^3 \end{bmatrix} \ .$$

On peut donc appliquer le même algorithme diviser pour régner en $O(n \cdot log(n))$ pour calculer $\overline{V}_n y$. Ensuite on obtient a en divisant le vecteur résultat par n.

Exercice 21 *Soient $A, B \subseteq \mathbf{N}$ deux ensembles de nombres naturels. On définit leur somme cartésienne par $A + B = \{a + b \mid a \in A, b \in B\}$. On suppose qu'il existe une constante $k \geq 1$ telle que si A et B ont n éléments alors ces éléments sont bornés par $k \cdot n$. On suppose aussi que les ensembles A et B sont représentés par 2 listes de nombres naturels. Proposez un algorithme pour calculer une liste qui représente $A + B$.*

Remarque 15 *On peut appliquer la transformée rapide de Fourier pour calculer le produit de deux nombres de n chiffres. Supposons :*

$$a = \Sigma_{i=0,\ldots,n-1} a_i 10^i \ , \quad b = \Sigma_{i=0,\ldots,n-1} b_i 10^i \ .$$

On peut voir a et b comme des polynômes :

$$p_a(x) = \Sigma_{i=0,\ldots,n-1} a_i x^i \ , \quad p_b(x) = \Sigma_{i=0,\ldots,n-1} b_i x^i \ ,$$

et considérer le problème de calculer les coefficients du polynôme produit :

$$p_c(x) = \Sigma_{i=0,\ldots,2n-2} \ c_i x^i = p_a(x) \cdot p_b(x) \ .$$

Pour ce faire, soit ω une racine $2n$-aire de l'unité. On évalue les polynômes p_a et p_b sur les points $\omega^0, \ldots, \omega^{2n-1}$ avec la FFT et on calcule les produits :

$$p_a(\omega^i) \cdot p_b(\omega^i) \qquad i = 0, 1, \ldots, n-1 \ .$$

Ensuite, on calcule la FFT inverse pour obtenir les coefficients c_i du polynôme p_c. On obtient ainsi une méthode pour multiplier deux nombres avec une complexité asymptotique meilleure que celle de la multiplication de Karatsuba (exemple 46). Le calcul exact de la complexité depend d'un certain nombre de choix de conception. Si l'on représente les nombres complexes avec un couple de flottants en double précision, il faut procéder à une analyse des erreurs. En effet, suite aux erreurs d'arrondi, le résultat du calcul peut être un nombre non-entier et il faut donc s'assurer qu'en arrondissant ce nombre à un entier on obtient le résultat attendu. Une approche alternative consiste à remplacer le cercle unitaire complexe avec un corps fini *suffisamment grand; dans ce cas on parle aussi de* number theoretic transform*. On obtient ainsi une complexité asymptotique $O(n \cdot log(n) \cdot log(log(n)))$.*

Chapitre 17

Algorithmes probabilistes

On peut introduire une *composante aléatoire* dans la *conception* et/ou l'*analyse* d'un algorithme. On distingue :

Algorithme probabiliste On considère un algorithme qui à certains moments du calcul *joue à pile ou face* pour déterminer son prochain état. On définit une v.a.d. X qui associe à chaque exécution possible (sur une entrée fixée) son *coût d'exécution* et on cherche à calculer son espérance $E[X]$.

Analyse en moyenne On fait une *hypothèse sur la distribution* des entrées. On définit une v.a.d. X qui associe à chaque entrée son *coût d'exécution* et on cherche à calculer son espérance $E[X]$.

Dans ce chapitre, on présente 3 exemples majeurs de l'approche probabiliste : le tri rapide, un test de primalité et un test d'identité de polynômes. Dans les chapitres suivants, on présentera d'autres exemples de conception et/ou d'analyse probabiliste en relation avec certaines structures de données (arbres binaires de recherche, listes à enjambement, tables de hachage).

17.1 Probabilité de terminaison et temps moyen de calcul

Générateurs (pseudo-)aléatoires

Dans un *algorithme probabiliste*, on peut invoquer une fonction qu'on appelle *générateur aléatoire* qui produit un nombre dans $\{0, 1\}$ avec une *probabilité uniforme*.

En pratique, dans un langage de programmation on fait appel à une fonction de bibliothèque qui approche de façon plus ou moins fidèle le comportement d'un générateur aléatoire. En particulier en C, on génère un entier dans l'intervalle $[0, \mathsf{RAND\_MAX}]$ avec la fonction rand. Sur mon ordinateur, $\mathsf{RAND\_MAX} = 2,147,483,647$ et on considère que $\mathsf{rand}()\%2$ est un bit aléatoire avec probabilité uniforme. La fonction rand produit (de façon déterministe) une suite de nombres à partir d'un *germe* qui est crée par la fonction srand. Typiquement on utilise une fonction système time pour éviter de générer toujours le même germe. [1]

```
srand((unsigned)(time(NULL))); // initialisation suite
...rand();...rand();...        // appels fonction rand
```

1. Cette utilisation de la fonction rand est adaptée aux simulations, les tests de programmes,... mais elle n'est pas du tout recommandée pour les applications cryptographiques.

Dans les analyses, on fera l'hypothèse que les résultats d'une suite d'invocations du générateur sont *indépendants*. Donc la probabilité d'obtenir une suite $w \in \{0,1\}^*$ est $2^{-|w|}$.

Fixons un *algorithme* A (ou un programme) et une *entrée* i de l'algorithme. Une *exécution* de $A(i)$ est une *suite d'états* traversées par l'algorithme à partir de la configuration initiale (voir chapitre 9) Si l'exécution *termine* alors la suite est *finie* sinon elle est *infinie* (dénombrable). Dans les algorithmes *déterministes* l'exécution est unique, mais dans les algorithmes *probabilistes* on peut avoir plusieurs exécutions (pour la même entrée).

Probabilité de terminaison

Soit Ω l'ensemble des *exécutions finies* de $A(i)$. Pour tout $\omega \in \Omega$ on définit :

$rnd(\omega) \in \{0,1\}^*$	les bits aléatoires utilisés dans ω
$r(\omega) = \|rnd(\omega)\|$	longueur $rnd(\omega)$
$p(\omega) = 2^{-r(\omega)}$	'probabilité' de l'exécution de ω.

La *'probabilité' que l'algorithme A termine sur l'entrée i* est alors :

$$\Sigma_{\omega\in\Omega} p(\omega) = \Sigma_{\omega\in\Omega} 2^{-r(\omega)} \ .$$

Par extension, on dit qu'un algorithme A *termine avec probabilité* 1 si pour toute entrée il termine avec probabilité 1. La fonction p n'est pas forcement une probabilité, mais au moins on a la propriété suivante.

Proposition 10

$$\Sigma_{\omega\in\Omega} p(\omega) \leq 1 \ .$$

Preuve. Si w, w' sont deux *mots*, on écrit $w \leq w'$ si w est un *préfixe* de w'. Si $\omega \neq \omega'$ alors $rnd(\omega) \not\leq rnd(\omega')$. Donc :

$$R = \{rnd(\omega) \mid \omega \in \Omega\} \ ,$$

est un ensemble de mots $\{0,1\}^*$ qui sont *incomparables par rapport au préfixe*. Par exemple : $R = \{1, 01, 001, 0001, \ldots\}$.

D'abord on montre que si R est *fini* alors :

$$\Sigma_{w\in R} 2^{-|w|} \leq 1 \ .$$

Par *récurrence* sur $\sharp R$ et la *longueur du mot* le plus long dans R.

— Si $R = \{w\}$ alors $P(R) = 2^{-|w|} \leq 1$.

— Si $\sharp R > 1$ alors on définit : $R_i = \{w \mid iw \in R\}$, $i = 0, 1$. R_i est encore un ensemble de *mots incomparables* et par hypothèse de récurrence :

$$P(R) = \frac{1}{2}P(R_0) + \frac{1}{2}P(R_1) \leq \frac{1}{2} + \frac{1}{2} = 1 \ .$$

Si maintenant R est *dénombrable* on pose :

$$R_n = \{w \in R \mid |w| \leq n\} \ .$$

Comme $R_n \subseteq R_{n+1}$:

$$\Sigma_{w\in R_n} 2^{-|w|} \leq \Sigma_{w\in R_{n+1}} 2^{-|w|} \leq 1 \ .$$

Donc :

$$\Sigma_{w\in R}2^{-|w|} = \lim_{n\to+\infty}\Sigma_{w\in R_n}2^{-|w|} \leq 1 \ .$$

□

Si $\Sigma_{\omega\in\Omega}p(\omega) = 1$ alors on peut définir un *espace de probabilité discret* :

$$(\Omega, 2^{\Omega}, P)$$

avec pour $A \subseteq \Omega$, $P(A) = \Sigma_{\omega\in A}p(\omega)$. Remarquons qu'un algorithme qui termine avec probabilité 1 *n'est pas* un algorithme dont toutes les exécutions sont finies (mais les exécutions infinies ont probabilité 0).

Temps moyen de calcul

Supposons que l'algorithme A sur l'entrée i *termine avec probabilité* 1. Alors on peut définir une v.a.d. C qui associe à chaque exécution finie un *coût*. Par exemple on peut prendre :

$$C(\omega) = |\omega| \ ,$$

en considérant que la *longueur de l'exécution* correspond en gros au *temps de calcul*. Ensuite on peut calculer l'espérance $E[C]$ qui est donc le *coût moyen de l'algorithme A sur l'entrée i* :

$$E[C] = \Sigma_{\omega\in\Omega}\ |\omega| \cdot 2^{-r(\omega)} \ .$$

En résumant, pour un algorithme probabiliste A avec entrée i, soit Ω l'ensemble des *exécutions finies* et pour $\omega \in \Omega$ soit $r(\omega)$ le nombre de bits aléatoires utilisés dans ω. Alors :

— La *probabilité de terminaison* est :

$$\Sigma_{\omega\in\Omega}\ 2^{-r(\omega)} \ .$$

— Le *temps moyen de calcul* est :

$$\Sigma_{\omega\in\Omega}\ |\omega| \cdot 2^{-r(\omega)} \ .$$

Exemples

On applique les notions présentées à une série d'exemples académiques avant de passer à des exemples plus intéressants.

Exemple 49 *On étudie la terminaison et le coût moyen de la fonction suivante.*

```
void proba1(){
       while (1) {if ((rand()%2)==1){break;} }}
```

Analyse *On* suppose *que :*

$$P(\texttt{rand()\%2} == \texttt{1}) = \frac{1}{2} \ .$$

La probabilité de terminer exactement à la n-ième itération *est :*

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

Donc la probabilité de terminer dans les premières n itérations *est :*

$$\Sigma_{i=1,\ldots,n}\frac{1}{2^i} = 1 - \frac{1}{2^n} \ ,$$

et la probabilité de terminer tout court *est :*

$$\Sigma_{i=1,\ldots,\infty}\frac{1}{2^i} = 1 \ .$$

On reconnaît ici une distribution géométrique *avec paramètre* $\frac{1}{2}$. *Le* coût moyen *de l'algorithme est donc* 2 *(itérations).*

Exemple 50 *On considère maintenant la fonction suivante.*

```
void proba2(){
        long n=1;
        short stop=0;
        while (!stop) {stop=1; int i;
                       for (i=0;i<n;i++){
                       if ((rand()%2)==1){stop=0; break;}}
                       n=n+1;}}
```

Analyse *On se* souvient *que :*

$$\begin{array}{ll}(a) & 1 + x \leq e^x, \\ (b) & \Sigma_{i=1,\ldots,n}\frac{1}{2^i} = 1 - \frac{1}{2^n} \ .\end{array}$$

La probabilité p_n *de* terminer exactement à la n-ième itération *est :*

$$p_n = (\Pi_{i=1,\ldots,(n-1)}(1 - \frac{1}{2^i})) \cdot \frac{1}{2^n} \ .$$

En utilisant (a) *et* (b) *on a pour* $n \geq 1$ *:*

$$(\Pi_{i=1,\ldots,n}(1 - \frac{1}{2^i})) \leq \frac{1}{\sqrt{e}} \ .$$

Donc pour $n = 1$ *on a* $p_1 = 1/2$ *et pour* $n \geq 2$ *on a :*

$$p_n \leq \frac{1}{\sqrt{e}2^n} \ .$$

Il suit que la probabilité de terminer *est :*

$$\begin{array}{l}\Sigma_{n=1,\ldots,\infty}p_n \\ \leq \frac{1}{2} + \frac{1}{\sqrt{e}} \cdot (\Sigma_{i=2,\ldots,\infty}\frac{1}{2^i}) \\ = \frac{1}{2} + \frac{1}{\sqrt{e}} \cdot \frac{1}{2} = \frac{\sqrt{e}+1}{2\sqrt{e}} \approx 0,8 < 1 \ .\end{array}$$

Et donc la probabilité de boucler *est significative !*

Exemple 51 *Et encore une fonction.*

```
void proba3(int m){
        long k=0;
        while (k<m) {if ((rand()%2)==1){k=k+1;}} }
```

Analyse *Pour terminer on doit tirer m fois* 1. *Il s'agit de la distribution* binomiale négative. *Donc :*

1. `proba3` termine avec probabilité 1.
2. *Le* coût moyen *est :* $2m$.

A noter cette fonction dépend de l'entrée et que son coût est exponentiel *dans le nombre de bits nécessaires à représenter l'entrée* `m`.

Exemple 52 *Enfin, on considère la fonction suivante.*

```
void proba4(){
        int n=1;
        while (!((rand()%2)==1)){n=n+1;}
        short stop=0;
        while (!stop) {
            stop=1; int i;
            for (i=0;i<n;i++){if ((rand()%2)==1){stop=0;}}}}
```

Analyse *D'abord on suit une* distribution géométrique *et on affecte à la variable n un entier* $i \geq 1$ *avec probabilité* 2^{-i}. *La boucle* `for` *laisse* `stop` *à* `true` *avec probabilité* $p_n = \frac{1}{2^n}$. *La deuxième boucle* `while` *correspond donc aussi à une* distribution géométrique *avec paramètre* p_n. *La* probabilité de terminaison est donc 1 *:*

$$\begin{aligned}&\Sigma_{n=1\ldots,\infty} 2^{-n} (\Sigma_{k=1,\ldots,\infty} (1-p_n)^{(k-1)} p_n)\\ &= \Sigma_{n=1\ldots,\infty} 2^{-n}(1)\\ &= 1\ .\end{aligned}$$

On considère maintenant le temps moyen de calcul *en comptant les itérations des* 2 *boucles* `while` *et en faisant abstraction du fait que les entiers représentables par le type* `int` *de* C *sont bornés par* `2,147,483,647`. *Soient* C_1 *et* C_2 *les v.a.d.* coût *qui correspondent à la première et à la deuxième boucle* `while`. *On a :*

$$\begin{aligned}P(C_1 = n) &= \tfrac{1}{2^n}\\ E[C_1] &= 2\\ E[C_2 | C_1 = n] &= 2^n\ .\end{aligned}$$

On calcule (en utilisant les propriétés des v.a.d. conditionnelles) :

$$\begin{aligned}E[C_2] &= \Sigma_{n=1,\ldots,\infty} E[C_2 \mid C_1 = n] \cdot P(C_1 = n)\\ &= \Sigma_{n=1,\ldots,\infty} 2^n \cdot 2^{-n}\\ &= \Sigma_{n=1,\ldots,\infty} 1\\ &= \infty\ .\end{aligned}$$

Le coût moyen est donc infini *!*

Exercice 22 *Considérez la fonction* C *suivante qui effectue une recherche 'aléatoire' d'un élément dans un tableau.*

```
int rs (int x, int n, int t[n]){
    short check[n];
    int i;
```

```
for (i=0;i<n;i++){check[i]=0;}
int count=0;
while (count < n){
     int i=(rand()%n);
     if (t[i]==x){return i;}
     if (!check[i]){check[i]=1; count++;}}
return -1;}
```

Calculez (de façon exacte ou approchée) le nombre moyen d'appels à la fonction rand *dans les cas suivants :*

1. *Le tableau contient l'élément cherché* 1 *fois.*
2. *Le tableau contient l'élément cherché* k *fois.*
3. *Le tableau ne contient pas l'élément cherché.*

17.2 Tri rapide (*quicksort*)

On considère un algorithme dit de *tri rapide* (*quicksort*, en anglais) [Hoa61]. [2]

Algorithme de partition

Le tri rapide est basé sur une fonction de *partition* qui prend en entrée un ensemble fini de valeurs X et une valeur *pivot* v et génère l'ensemble X_1 des valeurs dans X strictement inférieurs à v et l'ensemble X_2 des valeurs dans X supérieurs ou égales à v. Supposons que X contienne n valeurs. Si X est représenté par une liste alors il est clair que l'on peut produire les deux listes qui représentent les ensembles X_1 et X_2 en $O(n)$. Si X est représenté par un tableau a alors il est remarquable que l'on peut générer X_1 et X_2 en temps $O(n)$ et sans effectuer d'allocation de mémoire (en anglais, on dit aussi que l'algorithme travaille *in place*). Supposons que les éléments de l'ensemble à partitionner sont mémorisés dans les cellules d'indice compris entre i et j avec $i < j$ On itère :

1. Tant que a[i] $< v$ on incrémente i. Si i 'croise' j on sort de l'itération.
2. Tant que $v \leq$ a[j] on décrémente j. Si j 'croise' i on sort de l'itération.
3. Si on arrive à ce point, on doit avoir a[i] $\geq v$ et a[j] $< v$. On permute a[i] avec a[j] et on reprend l'itération (pas 1).

Il est facile de modifier l'algorithme pour qu'à la fin de la partition il retourne l'indice à partir duquel on trouve les éléments plus grands ou égaux que le pivot (et une valeur conventionnelle s'il y en a pas). Dans la suite, on appelle cet indice le *point de partition*.

Algorithme de tri

On considère maintenant l'application de l'algorithme de partition au problème du tri. On suppose que les données à trier sont stockées dans un tableau a dans les positions comprises entre min et max et on prend a[max] comme pivot. Si min = max on a rien à faire ! Sinon :

— Soit k le point de partition par rapport au pivot.
— Si k<max on échange a[k] avec a[max] ; on met donc le pivot au point de partition.
— Si nécessaire, on calcule récursivement qsort(min,k-1) et qsort(k+1,max).

2. Cet algorithme est dans une *top* 10 d'algorithmes du XX siècle (voir `https://www.siam.org/pdf/news/637.pdf`).

Complexité dans le pire des cas et en moyenne

Le pire des cas est quand toutes les partitions sont *déséquilibrées*. Par exemple, si le tableau est *déjà ordonné* (SIC). Dans ce cas, le coût est *quadratique*. Pourtant, le qsort est un algorithme de choix pour effectuer le tri. Par exemple, il est dans la bibliothèque standard de C. Le fait est qu'en *moyenne* l'algorithme a une complexité $O(nlog(n))$ (qui est bien meilleure que quadratique !). Par ailleurs, l'opération de partition est efficace (en temps et en mémoire).

Il y a deux façons d'analyser le comportement moyen du tri rapide. La première façon (qui est celle étudiée dans la suite) est de le transformer dans un algorithme probabiliste qui à chaque appel récursif choisit le pivot de façon aléatoire. Dans cette approche on ne fait pas d'hypothèse sur la distribution des données en entrée. Ce qu'on montre est que pour toute entrée, en choisissant les pivots de façon aléatoire on aura un coût moyen en $O(nlog(n))$. Une deuxième façon de procéder est de supposer une distribution uniforme des données. Dans ce cas, on peut garder la version déterministe de l'algorithme (par exemple celle dans laquelle le pivot est toujours l'élément le plus à droite) et montrer que le coût moyen (sur toutes les entrées) est $O(nlog(n))$. L'analyse de cette deuxième approche est similaire à celle de la première et elle est omise.

Tri rapide : version probabiliste

La seule différence dans la version probabiliste du tri rapide est que pour trier les positions comprises entre min et max on commence par tirer un indice i tel que min $\leq$ i $\leq$ max avec probabilité uniforme et on permute a[i] avec a[max]. Le pivot est donc choisi avec une probabilité uniforme.

Analyse du tri rapide probabiliste

On suppose tous les éléments à trier *différents*. Pour simplifier la notation on dénote ces éléments par $1, 2, \cdots, n$. Par exemple, 2 est le deuxième plus petit élément. Au début du tri sa position est arbitraire mais à la fin du tri on sait qu'il sera en deuxième position à partir de gauche. Comme souvent dans les algorithmes de tri, on considère que le coût est proportionnel au nombre de comparaisons et on s'attache donc à compter le *nombre de comparaisons* effectuées *en moyenne* par l'algorithme. Ce nombre dépend du *choix aléatoire des pivots*. On représente un calcul par la suite des pivots choisis. Soit Ω l'ensemble de ces suites. On définit une *v.a.d.* X qui associe à chaque suite le nombre de comparaisons effectuées par le tri rapide. Le *but* est de calculer l'espérance $E[X]$.

Remarque 16 *Soient $i, j \in \{1, \ldots, n\}$ avec $i < j$ deux éléments à trier. Dans toute exécution, i et j sont comparés au plus un fois. En effet, l'algorithme compare un pivot aux autres éléments d'une partition. Donc pour comparer i et j il faut que l'un des deux soit un pivot et l'autre se trouve dans la même partition. Par ailleurs, dans la suite du calcul le pivot ne sera plus comparé à un autre élément (à la fin de la partition le pivot se trouve à la bonne place).*

On va maintenant utiliser une technique standard du calcul des probabilités : on exprime la v.a.d. X comme une somme de v.a.d. de Bernoulli dont on sait calculer l'espérance. Ensuite on utilise la linéarité de l'espérance pour dériver l'espérance de X. Pour $\omega \in \Omega$ une suite de comparaisons, on *définit* :

$$X_{i,j}(\omega) = \begin{cases} 1 & \text{si } i \text{ et } j \text{ sont comparés dans } \omega \\ 0 & \text{autrement.} \end{cases}$$

On *observe* :

$$X = \Sigma_{1 \leq i < j \leq n} X_{i,j} \ .$$

Et par *linéarité* :

$$E[X] = \Sigma_{1 \leq i < j \leq n} E[X_{i,j}] \ .$$

Il reste donc à calculer $E[X_{i,j}]$.

Définition 14 (probabilité de comparaison) *On note $P(i,j,n) = E[X_{i,j}]$ la probabilité que i et j sont* comparés *dans un tri rapide avec n éléments, où $1 \leq i < j \leq n$.*

Une première remarque est que $P(i,j,n)$ satisfait une relation de récurrence.

Proposition 11 *La fonction $P(i,j,n)$ satisfait :*

$$\begin{array}{lll} P(1,2,2) & = 1 & \\ P(i,j,n) & = \frac{2}{n} + \frac{1}{n} \cdot (& \Sigma_{k=1,\ldots,(i-1)} \ P(i-k, j-k, n-k) \ + \\ & & \Sigma_{k=(j+1),\ldots,n} P(i,j,k-1) \ ) \ . \end{array}$$

Preuve. Pour comparer i à j, soit on prend le pivot dans $\{i,j\}$ soit on le prend avant i ou après j. □

Une deuxième remarque (assez surprenante) est que $P(i,j,n)$ ne dépend pas de n.

Proposition 12

$$P(i,j,n) = \frac{2}{(j-i+1)} \ .$$

Preuve. Par récurrence sur n. Pour $n = 2$ on a bien $P(1,2,2) = \frac{2}{2-1+1} = 1$. Plus en général : $P(i,i+1,n) = 1$. Pour $n+1 > 2$ on a :

$$\begin{array}{lll} P(i,j,n+1) & = \frac{2}{n+1} + \frac{1}{n+1} (& \Sigma_{k=1,\ldots,(i-1)} \ P(i-k, j-k, n+1-k) \ + \\ & & \Sigma_{k=(j+1),\ldots,n+1} P(i,j,k-1) \ ) \\ & = \frac{2}{n+1} + \frac{1}{n+1} (& \Sigma_{k=1,\ldots,(i-1)} \ \frac{2}{j-i+1} \ + \\ & & \Sigma_{k=(j+1),\ldots,n+1} \ \frac{2}{j-i+1} \ ) \\ & = \frac{2}{n+1} + \frac{1}{n+1} & \frac{2(n-j+i)}{j-i+1} \\ & = \frac{2}{j-i+1} \ . & \end{array}$$

□

Proposition 13 *$E[X]$ est $O(nlog(n))$.*

Preuve. On calcule :

$$\begin{array}{ll} E[X] & = \Sigma_{i=1,\ldots,(n-1)} \Sigma_{j=i+1,\ldots,n} \frac{2}{(j-i+1)} \\ & = 2 \cdot (\Sigma_{i=1,\ldots,n-1} (\Sigma_{k=1,\ldots,(n-i)} \frac{1}{(k+1)})) \\ & \leq 2 \cdot (\Sigma_{i=1,\ldots,n-1} (\Sigma_{k=1,\ldots,n} \frac{1}{k})) \ . \end{array}$$

On approxime la somme par un intégral pour obtenir :

$$\Sigma_{x=2,\ldots,m}\frac{1}{x} < \int_1^m \frac{1}{x}\,\mathrm{d}x = \ln(m)\ .$$

Donc $\Sigma_{k=1,\ldots,n}\frac{1}{k} \leq 1+\ln(n)$ et :

$$E[X] \leq 2\cdot(n-1)(\ln(n)+1)$$

soit $E[X]$ est $O(n\ln(n))$. □

Exercice 23 *Le problème de la médiane est le suivant : on dispose d'un tableau* non-ordonné *de n éléments et on souhaite déterminer le k-*ème élément *du tableau trié où $1 \leq k \leq n$. On peut résoudre ce problème en $O(nlog(n))$ en triant le tableau et en retournant le k-ème élément du tableau trié. Il est facile de voir que pour $k = 1$ ou $k = n$ on a un algorithme en $O(n)$; il s'agit de trouver le minimum ou le maximum du tableau, respectivement. Proposez un algorithme probabiliste pour le problème de la médiane qui utilise la* fonction de partition *du tri rapide pour résoudre ce problème.*[3]

17.3 Test de primalité

Dans le cas du tri rapide le nombre maximum de comparaisons est borné par une valeur qui ne dépend pas de la suite de bits aléatoires ; l'algorithme *termine*. En général, on peut concevoir des algorithmes probabilistes où cette propriété n'est pas satisfaite. Dans ce cas on cherchera à montrer que l'algorithme *termine avec probabilité* 1. Certains algorithmes probabilistes (dits de *Montecarlo*) s'ils terminent peuvent fournir des *réponses incorrectes*. On cherche alors à borner la probabilité d'une réponse incorrecte et dans certains cas favorables on peut montrer qu'en itérant l'algorithme un certain nombre de fois sur la même entrée on obtient une réponse incorrecte avec une probabilité négligeable en pratique (par exemple une probabilité d'erreur inférieure à 2^{-100}). Parmi les algorithmes qui tombent dans cette catégorie, on étudie un test de primalité dans cette section et un test pour l'identité de deux polynômes dans la suivante.

Rappels d'arithmétique

Pour $n \geq 2$, on pose :

$$\begin{aligned}\mathbf{Z}_n &= \{0,1,\ldots,n-1\},\\ \mathbf{Z}_n^* &= \{a \in \mathbf{Z}_n \mid pgcd(a,n) = 1\}\ .\end{aligned}$$

L'ensemble $\mathbf{Z}_n$ avec les opérations d'addition et multiplication modulaire est un anneau et l'ensemble $\mathbf{Z}_n^*$ avec l'opération de multiplication modulaire est un groupe (le *groupe multiplicatif*). Si n est premier alors $\mathbf{Z}_n^* = \{1,\ldots,n-1\}$ et $\sharp\mathbf{Z}_n^* = (n-1)$. Le résultat suivant est connu comme *petit théorème de Fermat*.

Proposition 14 *Si n est premier et $a \in \mathbf{Z}_n^*$ alors* $(a^{(n-1)} \equiv 1) \mod n$.

3. Une variante de l'analyse du tri rapide permet de montrer qu'en *moyenne* l'algorithme qui en résulte a une complexité $O(n)$.

PREUVE. Soit $k = min\{i > 0 \mid (a^i \equiv 1) \mod n\}$ et soit :

$$A = \{a^0, a^1, \ldots, a^{k-1}\} ,$$

où il est entendu que les exposants sont modulo n. Si $k = (n-1)$ on a terminé. Sinon on, va montrer que $(n-1)$ est un multiple de k, disons $(n-1) = k \cdot m$, et par les propriétés de l'exposant on a :

$$a^{n-1} = (a^k)^m = 1^m = 1 .$$

On montre d'abord que $\sharp A = k$. En effet, si $0 \leq i < j \leq (k-1)$ alors $a^i \neq a^j$. Autrement, $a^{(j-i)} = 1$ et $(j-i) < k$.

Si $k < (n-1)$ alors on peut trouver $b_1 \in (\mathbf{Z}_n^* \backslash A)$. On considère l'ensemble :

$$A_1 = \{a^0 b_1, a^1 b_1, \ldots, a^{k-1} b_1\} .$$

À nouveau, $\sharp A_1 = k$. Car si $0 \leq i < j \leq (k-1)$ et $a^i b_1 = a^j b_1$ alors $a^i = a^j$. D'autre part, $A \cap A_1 = \emptyset$. Car si $a^i = a^j b_1$ alors $b_1 \in A$. Si $A \cup A_1 = \mathbf{Z}_n^*$ on a montré que $(n-1) = 2k$. Sinon on choisit $b_2 \in \mathbf{Z}_n^* \backslash (A \cup A_1)$ et on itère le même raisonnement. □

Test de Fermat

La proposition 14 suggère une méthode pour *tester* la primalité d'un nombre $n \geq 2$. Choisir $a \in \{2, \ldots, n-1\}$ et vérifier :

$$(a^{(n-1)} \equiv 1) \mod n . \tag{17.1}$$

Pour calculer l'exposant modulaire on utilise la méthode du *carré itéré* présentée dans le chapitre 6.

Soit n un nombre qui n'est pas premier. Le problème est maintenant d'estimer la probabilité qu'en choisissant un nombre $a \in \{2, \ldots, n-1\}$ on obtient :

$$(a^{(n-1)} \not\equiv 1) \mod n . \tag{17.2}$$

On appelle un tel nombre a un *témoin* de la non-primalité de n.

Proposition 15 *Soient $n \geq 2$ et $a \in \{2, \ldots, n-1\}$*

1. *Si $pgcd(a, n) \neq 1$ alors $(a^{(n-1)} \not\equiv 1) \mod n$.*
2. *Si n n'est pas premier et si n admet un témoin $a \in \mathbf{Z}_n^*$ alors au moins la moitié des éléments dans $\{2, \ldots, n-1\}$ sont des témoins de la non-primalité de n.*

PREUVE. (1) On remarque : (i) si $(a^{n-1} \equiv 1) \mod n$ alors $pgcd(a^{n-1}, n) = 1$ et (ii) si d divise a et n alors d divise a^{n-1} et n.

(2) D'abord on observe que si $a \in \mathbf{Z}_n^*$ est un témoin et $d \in \mathbf{Z}_n^*$ ne l'est pas alors $(ad) \in \mathbf{Z}_n^*$ est un témoin. En effet, on a :

$$((ad)^{n-1} \equiv a^{n-1} d^{n-1} \equiv a^{n-1} \not\equiv 1) \mod n .$$

Ensuite, on montre que si $a \in \mathbf{Z}_n^*$ est un témoin et $d, d' \in \mathbf{Z}_n^*$ ne le sont pas alors ad et ad' sont deux témoins différents. Supposons $1 \leq d < d' < n$. Alors $(ad \equiv ad') \mod n$ implique $\exists c \ a(d'-d) = cn$. Comme $1 \leq (d'-d) < n$, a doit contenir un facteur de n ce qui contredit $a \in (Z_n)^*$. Il suit que si $a \in (Z_n)^*$ est un témoin alors dans $(Z_n)^*$ il y a au moins autant de témoins que de non-témoins. Par ailleurs, par (1), tous les éléments dans $Z_n \backslash (Z_n)^*$ sont des témoins. □

Limitations du test de Fermat

Un nombre de Carmichael est un nombre n qui n'est pas premier et qui n'a pas de témoin de non-primalité qui est premier avec n. On sait qu'il y a une infinité de nombres de Carmichael mais qu'ils sont très rares. Le plus petit nombre de Carmichael est $561 = 3 \cdot 11 \cdot 17$ et parmi les premiers 10^{15} nombres environ 10^5 sont des nombres de Carmichael. On sait aussi qu'en *pratique* le test de Fermat a une chance raisonnable de tomber sur un témoin de non-primalité pour un nombre de Carmichael (à savoir sur un nombre qui n'est pas premier avec le nombre de Carmichael). On peut donc dire que le test de Fermat est un test simple et pratique avec une petite limitation théorique.

Avec un peu plus de travail, on peut concevoir des tests de primalité plus sophistiqués (par exemple, le test de Miller-Rabin [Mil76, Rab80]) qui sont encore plus efficaces que le test de Fermat et qui n'ont aucune difficulté théorique avec les nombres de Carmichael. Par ailleurs, depuis [AKS04], on connaît aussi un test de primalité *déterministe* et polynomial en temps mais en pratique les tests probabilistes sont plus efficaces.

Exercice 24 *Les tests de primalité sont aussi utilisés pour* générer *des grands nombres premiers (typiquement des nombres avec* 10^3 *chiffres). En effet, on sait que les nombres premiers ne sont pas rares : il y a environ* $\frac{n}{log(n)}$ *nombres premiers parmi les premiers n nombres. Il suffit donc de tirer un nombre (impair) au hasard un certain nombre de fois jusqu'à tomber sur un nombre qui passe le test de primalité. Programmez une fonction probabiliste qui trouve un nombre premier de* 512 *bits avec une probabilité d'erreur inférieure à* 2^{-100} *(en ignorant les difficultés liées aux nombres de Carmichael). Estimez le nombre moyen de tirages qu'il faut effectuer avant de tomber sur un nombre (probablement) premier.*

17.4 Identité de polynômes

Soient p et q deux *polynômes* (en plusieurs indéterminées). On cherche à déterminer s'ils sont *identiques*, ou de façon équivalente à savoir si le polynôme $p - q$ est zéro partout. Une façon de résoudre ce problème est d'écrire les polynômes p et q comme *somme de monômes* et d'en comparer les coefficients. Cette approche peut demander un nombre *exponentiel* de multiplications. Par exemple, considérez le polynôme :

$$\Pi_{i=1,\ldots,n}(x_i + x_{i+1}) \ .$$

Par contre, l'évaluation d'un polynôme sur un point demande un nombre de multiplications qui est *linéaire* dans la taille du polynôme. La stratégie pour déterminer si un polynôme est zéro partout consiste donc à l'évaluer sur un certain nombre de points choisis de façon aléatoire. On a donc besoin d'estimer la probabilité de tomber sur un zéro du polynôme. Le point de départ est un résultat standard sur les racines d'un polynômes.

Proposition 16 *Soit* $p(x)$ *un polynôme dans une indéterminée* x *et de degré* d*. Si* $p(x) \neq 0$ *alors* $p(x)$ *admet au plus* d *racines différentes.*

Preuve. On peut utiliser la proposition 7 qui passe par les matrices de Vandermonde. On présente ici une preuve alternative par récurrence sur d. Si $d = 0$ alors $p(x)$ est constant et si $p(x) \neq 0$ alors il a 0 racines. Si $d > 0$ et $p(a) = 0$ alors par la propriété de la division sur les polynômes ils existent uniques $p'(x)$ et $r(x)$ tels que $p(x) = p'(x)(x - a) + r$, le degré de $p'(x)$ est $d - 1$ et le degré de r est 0. De plus on doit avoir : $p(a) = r = 0$. Par hypothèse de récurrence, $p'(x)$ a au plus $d - 1$ racines et donc $p(x)$ a au plus d racines. □

Notation Soit X un ensemble fini. On utilise la notation $x \leftarrow X$ pour affecter à la variable x un élément de l'ensemble X avec probabilité uniforme.

Corollaire 1 *Soit $p(x) \neq 0$ un polynôme avec une indéterminée x et degré d sur un corps F. Soit $F' \subseteq F$ tel que $\sharp F' = f$. Alors :*

$$P(a \leftarrow F' : p(a) = 0) \leq \frac{d}{f} \ .$$

Preuve. Le polynôme a au plus d racines dans F et donc au plus d racines dans F'. □

Par exemple, si l'on prend $f = 2d$ la probabilité de tomber sur une racine du polynôme est au plus $1/2$. Si l'on répète le test 100 fois en choisissant des éléments $a_1, \ldots, a_{100}$ de F' de façon indépendante alors si $p(a_i) = 0$ pour $i = 1, \ldots, n$ on peut affirmer que $p = 0$ avec une probabilité d'erreur bornée par 2^{-100}. Ce résultat se généralise à des polynômes en plusieurs indéterminées (un résultat connu aussi comme *lemme de Schwartz-Zippel*).

Proposition 17 *Soit $p(x_1, \ldots, x_m) \neq 0$ un polynôme en m indéterminées $x_1, \ldots, x_m$ avec d comme degré maximal de chaque indéterminée. Le polynôme étant sur un corps F, soit $F' \subseteq F$ tel que $\sharp F' = f$. Alors :*

$$P(a_1, \ldots, a_m \leftarrow F' : p(a_1, \ldots, a_m) = 0) \leq \frac{m \cdot d}{f} \ .$$

Preuve. Par récurrence sur m. Pour $m = 1$ on applique le corollaire 1. Pour $m > 1$ on peut réécrire $p(x_1, \ldots, x_m)$ en factorisant la variable x_1 :

$$p(x_1, \ldots, x_m) = \Sigma_{i=0\ldots,d}\ x_1^i \cdot p_i(x_2, \ldots, x_m) \ . \tag{17.3}$$

Comme $p \neq 0$ il existe j tel que $p_j \neq 0$. Tirons $a_1, a_2, \ldots, a_m$ avec probabilité uniforme. On pose :

$$p'(x_1) = p(x_1, a_2, \ldots, a_m) \ .$$

Si $p(a_1, \ldots, a_m) = 0$ alors on a deux situations possibles :

1. $p_i(a_2, \ldots, a_m) = 0$ pour $i = 0, \ldots, d$.
2. $\exists\, i\ \ p_i(a_2, \ldots, a_m) \neq 0$ et $p'(a_1) = 0$ (avec $p'(x_1) \neq 0$ car $p_i(a_2, \ldots, a_m) \neq 0$).

La probabilité de la première situation est bornée par :

$$P(a_2, \ldots, a_m \leftarrow F' : p_j(a_2, \ldots, a_m) = 0) \leq \frac{d \cdot (m-1)}{f} \ ,$$

et la probabilité de la deuxième est bornée par :

$$P(a_1 \leftarrow F' : p'(a_1) = 0) \leq \frac{d}{f} \ ,$$

et on conclut en observant :

$$\frac{d \cdot (m-1)}{f} + \frac{d}{f} = \frac{d \cdot m}{f} \ .$$

□

Exemple 53 *Les polynômes ont une propriété remarquable : s'ils sont différents alors ils sont différents presque partout. Cette propriété est souvent utilisée pour* amplifier *les différences entre deux structures discrètes. On donne un petit exemple de cette application (par exemple, dans le cadre des codes correcteurs d'erreurs). Considérez les* expressions booléennes *suivantes :*

$$A = (\neg x_1 \cdot x_2 + x_1 \cdot \neg x_2) \cdot x_3 + x_1 \cdot x_2 \ , \quad B = (\neg x_1 \cdot x_2 + x_1) \cdot x_3 \ .$$

Dans une expression booléenne, les variables varient sur l'ensemble $\mathbf{2} = \{0, 1\}$*. Les symboles* $\neg$*,* $\cdot$ *et* $+$ *indiquent la négation, la conjonction et la disjonction logique, respectivement. On aimerait savoir si pour toute affectation de valeurs booléennes aux variables,* A *et* B *produisent toujours le même résultat.*

Si on échantillonne x_1, x_2, x_3 *dans* $\mathbf{2}$ *de façon uniforme, la probabilité de distinguer ces expressions est* $1/2^3$ *(il y a une seule affectation qui distingue les deux expressions). Il se trouve qu'on peut voir ces expressions comme des polynômes. Pour ce faire, on replace la négation* $\neg x$ *par* $(1 - x)$ *et la conjonction et la disjonction par le produit et la somme, respectivement. On obtient ainsi :*

$$p_A = ((1 - x_1) \cdot x_2 + x_1 \cdot (1 - x_2)) \cdot x_3 + x_1 \cdot x_2 \ , \quad p_B = ((1 - x_1) \cdot x_2 + x_1) \cdot x_3 \ .$$

Les expressions en question sont dans une classe d'expressions pour laquelle on peut montrer que deux expressions sont équivalentes ssi elles induisent le même polynôme. Ainsi, pour distinguer A *et* B*, on peut échantillonner* x_1, x_2, x_3 *dans* $\mathbf{Z}_7 = \{0, 1, \ldots, 6\}$ *et dans ce cas la probabilité de distinguer* p_A *de* p_B *est* $216/343$ *!*

Remarque 17 *C'est un problème ouvert important de savoir s'il existe un algorithme* déterministe *polynomial en temps pour décider de l'identité de deux polynômes à plusieurs variables.*

Chapitre 18

Arbres binaires de recherche

Soit A un ensemble fini d'éléments que l'on peut comparer avec un *ordre total*. On cherche une façon de représenter A qui nous permet d'effectuer (au moins) les *opérations* suivantes de façon efficace :

— *insertion* d'un élément dans A,
— *test d'appartenance*,
— *élimination* d'un élément de A.

Si l'on représente un ensemble avec n éléments comme une liste (ordonnée ou non-ordonnée) ces opérations coûtent $O(n)$. On va étudier une représentation basée sur des arbres binaires de recherche (*binary search trees* en anglais, *ABR* en abrégé) qui permet d'effectuer les opérations en $O(h)$ où h est la hauteur de l'arbre qui représente l'ensemble.

18.1 Opérations

Définition 15 (ABR) *Un ABR est un arbre dans le sens de la définition 5 et qui en plus satisfait la condition suivante : la valeur de chaque noeud est supérieure à la valeur de chaque noeud dans le sous-arbre gauche et est inférieure à la valeur de chaque noeud dans le sous-arbre droit.*

On fait l'hypothèse que que chaque noeud est représenté par une structure avec 3 champs :

val	valeur
left	pointeur fils gauche
right	pointeur fils droit

Un ABR vide nil est typiquement représenté par un pointeur NULL. On étudie un certain nombre d'opérations dont la programmation est directe si l'on utilise les appels récursifs. Avec l'exception de l'opération d'impression qui est linéaire dans la *taille* de l'ABR, toutes les autres opérations sont linéaires dans la *hauteur* de l'ABR.

Impression L'impression par ordre croissant d'un ABR correspond à une visite en profondeur d'abord et de gauche à droite de l'arbre. Récursivement :
— On imprime le sous-arbre gauche.
— On imprime le noeud.
— On imprime le sous-arbre droit.

Insertion Pour insérer un élément x on navigue dans l'ABR jusqu'à trouver :

— soit x : dans ce cas on ne fait rien.
— soit la feuille nil où il faut placer x.

Appartenance Pour déterminer si un élément x est dans l'ABR on navigue dans l'arbre jusqu'à trouver :
— soit x : dans ce cas on peut rendre un pointeur au noeud.
— soit nil : dans ce cas on peut rendre un pointeur NULL.

Minimum Pour trouver le minimum de l'ABR il suffit de suivre toujours le branchement gauche jusqu'à trouver un noeud dont le fils gauche est nil.

Élimination du minimum Pour éliminer l'élément minimum on commence par suivre le branchement gauche jusqu'à trouver un noeud dont le fils gauche est nil. Ensuite on remplace ce noeud par son fils droit (il peut être nil).

Élimination Pour éliminer un élément x dans l'ABR on navigue dans l'arbre jusqu'à trouver :
— soit nil et on ne fait rien.
— soit x et on distingue 3 *cas* :

1. x a *0 fils*. Le père de x va pointer vers NULL.
2. x a *1 fils*. Le père de x va pointer vers le fils de x.
3. x a *2 fils*. On transforme l'arbre comme suit :

$$(x, l, r) \to (\mathsf{min}(r), l, \mathsf{mindel}(r))$$

à savoir le minimum du sous-arbre droite remplace x et on élimine le minimum du sous-arbre droite alors que le sous-arbre gauche n'est pas modifié. Une solution symétrique où on modifie le sous-arbre gauche est possible. Il est aussi possible de combiner en une seule opération la recherche du minimum avec son élimination.

D'autres opérations comme la recherche de l'élément maximum et la recherche du successeur (ou du prédécesseur) d'un élément donné peuvent être réalisées en suivant les mêmes idées. On peut aussi se compliquer un peu la tâche en implémentant toutes les opérations sans appels récursifs et/ou en gérant explicitement la *récupération de la mémoire*.

18.2 Hauteur moyenne d'un arbre

Le *pire des cas* est quand un arbre est fortement *déséquilibré* (à la limite une liste). On va montrer qu'*en moyenne* la hauteur d'un arbre généré de façon aléatoire par une suite d'insertions est *logarithmique* dans sa taille. Malheureusement, l'analyse ne couvre pas la situation qu'on trouve en pratique où l'on mélange les opérations d'insertion et d'élimination. Pour cette raison, on trouve dans la littérature des représentations plus sophistiquées (arbres bicolores ou arbres AVL) qui implémentent les opérations d'insertion et d'élimination de façon à garder les arbre *équilibrés*. Dans le chapitre 19, on étudiera les *listes à enjambements* qui sont une alternative simple aux arbres équilibrés.

Définition 16 (somme hauteurs) *Si T est un* arbre binaire de recherche *(ABR) on dénote par $P(T)$ la somme des hauteurs de ses noeuds (la racine a hauteur 0 et $P(T) = 0$ si T est vide).*

Remarque 18 *Si T est une* liste *alors $P(T)$ est $O(n^2)$ et si T est un* arbre complet *alors $P(T)$ est $O(n \log n)$.*

Définition 17 (génération aléatoire) *Soit S_n l'ensemble des permutations sur l'ensemble $\{1, \ldots, n\}$ avec éléments π, π'. On suppose qu'un ABR avec n noeuds est* généré *de la façon suivante : on produit une permutation $\pi \in S_n$ avec probabilité uniforme et on insère dans l'arbre vide $\pi(1), \ldots, \pi(n)$. On dénote par T_π l'ABR obtenu.*

Définition 18 (moyenne somme hauteurs) *On dénote par $P(n)$ la moyenne des $P(T_\pi)$ (définition 16) :*

$$P(n) = \frac{1}{n!}(\Sigma_{\pi \in S_n} P(T_\pi)) \ .$$

Par exemple, pour $n = 3$ on a $3! = 6$ ABR possibles. Parmi ces ABR, il y en a 4 avec $P(T) = 3$ et 2 avec $P(T) = 2$. On a donc $P(3) = 8/3$. Notre objectif est de trouver une *borne supérieure* pour la fonction $P(n)$. Si T est un ABR *non-vide* alors on dénote par T_g et T_d les *sous-arbres* gauche et droite (qui peuvent être vides). On vérifie aisément la proposition suivante.

Proposition 18 *Si T est un ABR non-vide avece n noeuds alors :*

$$P(T) = P(T_g) + P(T_d) + (n-1) \ .$$

La proposition suivante nous donne une façon d'exprimer la fonction $P(n)$ par récurrence.

Proposition 19

$$P(n) = \begin{cases} 0 & \text{si } n = 1 \\ \frac{1}{n}(\Sigma_{i=1,\ldots,n}(P(i-1) + P(n-i) + (n-1))) & \text{si } n > 1 \ . \end{cases}$$

Preuve. Le cas $n = 1$ est clair. Si $n > 1$ on argumente comme suit. Si on tire π de façon uniforme on a pour $1 \leq i \leq n$:

$$P(\pi(1) = i) = \frac{1}{n} \ .$$

A noter que $\pi(1) = i$ veut dire que i est à la racine de l'arbre généré T. Ensuite l'arbre de gauche T_g (de droite T_d) résultera d'une permutation de $i-1$ éléments ($n-i$ éléments). On applique la proposition 18. □

On dérive que la hauteur moyenne est logarithmique.

Proposition 20 *La hauteur moyenne d'un noeud est $O(\log n)$.*

Preuve. Si $n > 1$ alors on dérive de la proposition 19 que :

$$P(n) = \frac{2}{n}(\Sigma_{k=1,\ldots,n-1} P(k)) + (n-1) \ .$$

On cherche une fonction $f(n)$ telle que $0 = P(1) \leq f(1)$ et

$$\frac{2}{n}(\Sigma_{k=1,\ldots,n-1} f(k)) + (n-1) \leq f(n) \ . \tag{18.1}$$

Si on prend $f(n) = 2n \log n$, on satisfait la première condition car $f(1) = 0$. Par ailleurs, on a :

$$\Sigma_{k=1,\ldots,n-1} f(k) \leq \int_1^n f(x)\mathrm{d}x \ .$$

En utilisant le fait que : $\int x \log x \mathrm{d}x = \frac{x^2}{4}(2 \log x - 1)$, on vérifie la deuxième condition (18.1). On peut donc montrer par récurrence que $P(n) \leq f(n) = 2n \log n$. Il suit que la hauteur de chaque noeud est en moyenne $O(\log n)$. □

Chapitre 19

Listes à enjambements (*skip lists*)

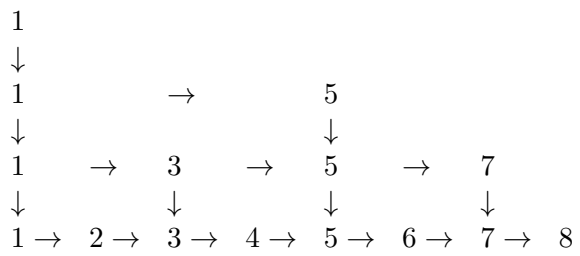
Les listes à enjambements (*skip lists*) sont une structure de données introduite par W. Pugh [Pug90] à base de listes doublement chaînées à plusieurs étages. Il s'agit d'une structure de données 'probabiliste' dans le sens que certaines décisions sur la configuration des listes sont prises de façon probabiliste. On obtient ainsi une mise en oeuvre relativement simple des opérations standards de recherche, insertion et élimination, tout en assurant une complexité en temps logarithmique avec une probabilité élevée.

19.1 Listes à enjambements

Considérons une liste ordonnée L_0 avec n éléments. Pour l'instant on se focalise sur la recherche d'un élément dans la liste. On sait que la complexité de cette opération est $O(n)$. Supposons maintenant qu'on ajoute une deuxième liste L_1 pour accélérer la recherche. Dans cette deuxième liste on va mémoriser une fraction des éléments de la liste L_0. L'idée est que la liste L_0 est une ligne locale qui dessert toutes les stations alors que la liste L_1 est une sorte de ligne rapide qui nous permet d'avancer rapidement jusqu'à un certain point où on peut être obligé d'emprunter la ligne locale L_0. Une configuration 'optimale' consiste à mettre dans la liste L_1 $\sqrt{n}$ éléments de la liste L_0 qui sont espacés de $\sqrt{n}$. Par exemple, si la liste L_0 contient les éléments $1, 2, \ldots, 15, 16$, on va insérer dans la liste L_1 les éléments $1, 5, 9, 13$. Dans la recherche d'un élément on va visiter au plus $\sqrt{n}$ noeuds dans la liste L_1 et au plus $\sqrt{n}$ noeuds dans la liste L_0. Donc, dans le pire des cas, on visite $2 \cdot \sqrt{n}$ noeuds et on a une complexité en $O(\sqrt{n})$.

On peut aussi ajouter une ligne 'TGV' L_2. Dans ce cas, L_1 va contenir $n^{2/3}$ éléments de L_0 espacés de $\sqrt[3]{n}$ et L_2 va contenir $\sqrt[3]{n}$ éléments de L_1 espacés de $\sqrt[3]{n}$. Par exemple, si L_0 contient les éléments $1, 2, \ldots, 27$, alors L_1 contient les éléments $1, 4, 7, 10, 13, 16, 19, 22, 25$, et L_2 contient les éléments $1, 10, 19$. La recherche va donc visiter au plus $3 \cdot \sqrt[3]{n}$ noeuds, soit $O(\sqrt[3]{n})$. En général, on peut imaginer de construire la liste L_{i+1} à partir de la liste L_i en sélectionnant un élément sur deux. Ce processus s'arrête forcement après $log_2(n)$ étapes et il permet la recherche d'un élément en temps $O(log(n))$ d'une façon qui rappelle la recherche dichotomique ou la recherche dans un arbre binaire ordonné. Par exemple, si L_0 contient les

éléments $1, 2, 3, 4, 5, 6, 7, 8$ alors on obtient :



19.2 Approche probabiliste

On va maintenant discuter la conception des opérations d'insertion et d'élimination. A priori ces opérations risquent de déséquilibrer les listes et ainsi d'augmenter la complexité. Il se trouve qu'une approche probabiliste à l'opération d'insertion permet une mise-en-oeuvre simple dont on peut garantir l'efficacité avec une probabilité élevée.

On fait l'hypothèse que les listes $L_0, \ldots, L_k$ sont ordonnées de façon croissante et qu'elles contiennent un noeud sentinelle avec une valeur non-standard $-\infty$. Le point d'entrée de la structure est le noeud $-\infty$ de la liste L_k (la plus haute). Initialement, la structure est donc composée d'une seule liste composée à son tour d'un noeud qui contient la valeur $-\infty$.

Chaque noeud contient 2 champs pointeurs right et down et un champ val qui contient sa valeur (par exemple un entier). Si le noeud se trouve dans la liste L_i alors le champ right pointe à l'élément suivant dans la liste L_i, s'il existe, et le champ down au noeud avec la *même valeur* dans la liste L_{i-1} si elle existe.

Recherche Un algorithme de *recherche* d'un noeud avec valeur x à partir du point d'entrée ℓ pourrait être le suivant :

1. Si $\ell = \mathsf{NULL}$: on rend NULL comme résultat.
2. Sinon, si $\ell \to \mathsf{val} = x$: on rend ℓ comme résultat.
3. Sinon, soient $\ell_r = (\ell \to \mathsf{right})$ et $\ell_d = (\ell \to \mathsf{down})$.
   3.1 si $\ell_r = \mathsf{NULL}$: $\ell = \ell_d$ et on itère.
   3.2 Sinon, si $x < (\ell_r \to \mathsf{val})$: $\ell = \ell_d$ et on itère.
   3.3 Sinon, si $x \geq (\ell_r \to \mathsf{val})$: $\ell = \ell_r$ et on itère.

Élimination Pour *éliminer* une valeur x de la structure on va effectuer une recherche jusqu'à trouver (si elle existe) l'occurrence de x dans la liste de niveau le plus élevé (sinon, il n'y a rien à faire). Pendant cette recherche on maintient un pointeur pred de façon telle que si on trouve la valeur x dans un noeud n alors pred pointe au prédécesseur de n. Ensuite, on élimine le noeud correspondant ainsi que tous les noeuds qui contiennent la même valeur jusqu'à la liste L_0. Pour ce faire, la première fois on utilise le pointeur pred et pour les niveaux inférieurs, on cherche le prédécesseur du noeud $(\mathsf{n} \to \mathsf{down})$ à partir du noeud $(\mathsf{pred} \to \mathsf{down})$. Il suit de la définition de la fonction d'insertion (à suivre), que le nombre moyen d'arêtes entre $(\mathsf{pred} \to \mathsf{down})$ et $(\mathsf{n} \to \mathsf{down})$ est 2.

Insértion Pour *insérer* une valeur x dans la structure il faut d'abord effectuer une recherche pour déterminer l'endroit où x doit être inséré dans la liste L_0 (si pendant la recherche on trouve x, il n'y a rien à faire). Pendant cette recherche on maintient dans une *pile* P les

derniers éléments visités dans chaque liste ; ces éléments sont les prédécesseurs potentiels des noeuds créés qui vont contenir la valeur x.

Après la phase de recherche, on va créer un noeud n qui contient la valeur x, on extrait le premier élément de la pile P et on l'utilise pour insérer n dans la liste L_0.

Ensuite on passe à la *phase probabiliste*. On joue à pile ou face et tant qu'on tire pile on effectue les opérations suivantes.

— Si la pile P est vide on ajoute un nouveau niveau à la structure avec un noeud qui contient la valeur non-standard $-\infty$. Ce noeud devient le nouveau point d'entrée de la structure et il est inséré dans la pile.

— On extrait le premier élément de la pile P et on l'utilise pour insérer un nouveau noeud qui contient x.

Exemple 54 *On considère l'insertion de la valeur* 7 *dans la liste suivante où on utilise des indices en exposant pour distinguer les occurrences de la même valeur (en pratique les exposants sont des pointeurs).*

$$\begin{array}{ccccccccc} -\infty^1 & & \rightarrow & & 10^2 & & & & \\ \downarrow & & & & \downarrow & & & & \\ -\infty^3 & \rightarrow & 5^4 & \rightarrow & 10^5 & \rightarrow & 15^6 \end{array}$$

A la fin de la phase de recherche, la pile P *correpond à* $(-\infty^1, 5^4)$*. On va donc insérer* 7 *entre* 5^4 *et* 10^5 *et* P *devient* $(-\infty^1)$*. Ensuite commence la phase probabiliste. Si on tire pile, on va insérer* 7 *entre* $-\infty^1$ *et* 10^2*. Et si on tire encore pile, on va ajouter un nouveau niveau et obtenir la structure suivante.*

$$\begin{array}{ccccccccccc} -\infty^9 & & \rightarrow & & 7^9 & & & & & & \\ \downarrow & & & & \downarrow & & & & & & \\ -\infty^1 & & \rightarrow & & 7^8 & \rightarrow & 10^2 & & & & \\ \downarrow & & & & \downarrow & & \downarrow & & & & \\ -\infty^3 & \rightarrow & 5^4 & \rightarrow & 7^7 & \rightarrow & 10^5 & \rightarrow & 15^6 \end{array}$$

19.3 Analyse

Chaque élément qui a été inséré dans la liste a été ajouté à la liste L_0 et ensuite il a été propagé aux listes supérieurs avec une probabilité fortement décroissante. Soit X_j, pour $j \in \{1, \ldots, n\}$ une v.a.d. qui indique la hauteur atteinte par le j-ème élément de la structure. On a :

$$P(X_j \geq k) \leq 2^{-k} \ .$$

Soit H une v.a.d. qui représente la hauteur de la structure (le nombre de listes). On a :

$$H = max\{X_j \mid j \in \{1, \ldots, n\}\} \ ,$$

et en utilisant la borne union on dérive :

$$P(H \geq k) \leq \Sigma_{j=1,\ldots,n} P(X_j \geq k) = n \cdot 2^{-k} \ .$$

Si l'on prend $k = 2 \cdot log_2(n)$ on obtient que :

$$P(H \geq 2 \cdot log_2(n)) \leq \frac{1}{n} \ ,$$

et plus en général si $k = c \cdot log_2(n)$ on a :

$$P(H \geq c \cdot log_2(n)) \leq \frac{1}{n^{(c-1)}} \ .$$

On peut conclure qu'avec une haute probabilité la structure aura une hauteur logarithmique dans le nombre d'éléments qu'elle contient.

Cherchons maintenant à évaluer le nombre de noeuds visités dans la recherche d'un élément. Une recherche peut être visualisée comme une suite de mouvements vers la droite (en suivant le pointeur right) ou vers le bas (en suivant le pointeur down). Dans le cas le plus défavorable, la recherche nous conduit jusqu'à la liste de base L_0. Considérons maintenant les noeuds visités en ordre inverse, à partir donc du dernier qui se trouve dans la liste L_0. Avec probabilité 1/2 un de ces noeuds, se trouvant, disons, dans la liste L_i, a un noeud avec la même valeur dans la liste L_{i+1} et il s'agit du noeud suivant dans le chemin inversé. On a donc un chemin (inversé) qui à chaque étape monte au niveau supérieur avec probabilité 1/2 et reste au même niveau avec probabilité 1/2. Par ailleurs, on sait qu'avec probabilité au moins $1 - 1/n$ on a au plus $2log_2(n)$ niveaux et qu'en moyenne un chemin qui monte $2log_2(n)$ niveaux a longueur $4log_2(n)$.

On esquisse un raffinement possible de cette analyse qui cherche à quantifier la probabilité qu'on s'écarte de la moyenne. Soit N la v.a.d. qui compte le nombre de fois qu'on reste au même niveau. On sait que N est la somme de v.a.d. de Bernoulli indépendantes et que dans une telle situation N est fortement concentrée autour de son espérance. Supposons qu'on effectue $m = 8log_2(n)$ tirages. On a donc :

$$N = \Sigma_{i=1,\ldots,8log_2(n)} X_i \ ,$$

où X_i est une v.a.d. de Bernoulli. L'espérance de N est $E[N] = 4log_2(n)$. La borne de Chernoff est une inégalité qui s'applique à la somme de v.a.d. de Bernoulli indépendantes. Intuitivement, la borne dit que la probabilité que la somme s'écarte de la moyenne diminue de façon exponentielle. Parmi les nombreuses formulations qu'on trouve dans la littérature, on utilise la suivante :

$$P(N \geq c_N E[N]) \leq e^{-kE[N]} \ , \tag{19.1}$$

où $k = c_N ln(c_N) - c_N + 1$. Si l'on prend $c_N = 3/2$ on a $k \approx 0,1$ et donc :

$$P(N \geq 6log_2(n)) \leq n^{-0,4} \ . \tag{19.2}$$

Cette borne implique que si on fait $8log_2(n)$ tirages avec probabilité au moins $(1 - n^{-0,4})$ on va remonter jusqu'à la liste sommitale. On peut conclure l'analyse en combinant les deux bornes.

$$\begin{aligned} P((H < 2log_2(n)) \cap (N < 6log_2(n))) &= 1 - P((H \geq 2log_2(n)) \cup (N \geq 6log_2(n))) \\ &\geq 1 - (P(H \geq 2log_2(n)) + P(N \geq 6log_2(n))) \\ &\geq 1 - (\tfrac{1}{n} + \tfrac{1}{n^{0,4}}) \ . \end{aligned}$$

En pratique, les listes à enjambements sont compétitives avec les arbres binaires équilibrés tout en ayant une mise-en-oeuvre plus simple. A noter, qu'il est aussi possible de concevoir des listes à enjambements *déterministes* [MPS92] qui garantissent une complexité des opérations considérées en temps $O(log(n))$ dans le pire des cas.

Chapitre 20

Tables de hachage

Après les arbres binaires et les listes à enjambements, les tables de hachage (*hash tables* en anglais) sont une troisième structure de données qui permet une mise en oeuvre efficace des opérations de recherche, insertion et élimination sur un ensemble fini d'éléments. Dans une table d'hachage, la recherche d'un élément commence par un accès direct à un tableau en utilisant une *fonction de hachage* et continue avec la visite d'une liste qui peut être représentée de façon explicite avec des pointeurs (*table avec chaînage*) ou de façon implicite avec une fonction de sondage (*table avec adressage ouvert*).

20.1 Fonctions de hachage

En général, une fonction de hachage est une fonction *facile à calculer* qui envoie un ensemble U de grande taille dans un ensemble $T = \{0, \ldots, m-1\}$ de taille beaucoup plus réduite. On appelle un élément $x \in U$ une *clé*.

Dans les applications aux *tables de hachage*, il s'agit par exemple d'envoyer une chaîne de caractères (le nom d'une personne) sur l'indice d'une table de taille m. La *propriété idéale* dans ce cas est : pour tout $x \in U$ et $i \in \{0, \ldots, m-1\}$,

$$P(h(x) = i) = \frac{1}{m} \ .$$

Dans ce cas, on dit que la fonction de hachage est *uniforme*.

Remarque 19 *Dans les applications à la* cryptographie*, on pose des conditions beaucoup plus sévères. Il doit être impossible (en pratique) de (i) trouver une* collision *: $x \neq y$ tel que $h(x) = h(y)$ et (ii) trouver une* image inverse *: pour y donné, trouver x tel que $h(x) = y$. Même si on a une fonction de hachage* uniforme*, il suffit de la calculer sur environ $\sqrt{m}$ valeurs pour avoir une probabilité d'environ $\frac{1}{2}$ de trouver une collision (c'est le* paradoxe des anniversaires *!). Dans les* applications cryptographiques*, il faut donc choisir un m assez grand pour qu'on ne puisse pas calculer $\sqrt{m}$ fois la fonction de hachage dans un temps raisonnable. Typiquement, $m = 2^{256}$ (*SHA-2*). Bien sûr, pour l'*application aux tables de hachage*, on a des valeurs beaucoup plus petites.*

20.2 Tables de hachage avec chaînage

Une table de hachage est une *structure de données* qui sert à représenter un ensemble $X \subseteq U$ avec les opérations standard (voir listes, arbres, listes à enjambements) :

Opérations	Description
hmem(x)	appartenance
hins(x)	insère un élément
hrem(x)	enlève un élément

Soient $n = \sharp X$ et m la *taille de la table*. Le *facteur de charge* est

$$\alpha = n/m$$

On suppose que l'accès à un élément de la table se fait en *temps constant*. L'*objectif* est de réaliser les opérations en $O(\alpha)$ en moyenne.

Remarque 20 *En programmation, on cherche souvent le* bon compromis *entre* temps d'exécution *et* espace mémoire*. Dans le cas des tables de hachage, on utilise* plus de mémoire *pour accélérer (en moyenne) le temps d'accès à une liste d'éléments (les* listes à enjambements *suivent une philosophie similaire ainsi que les* techniques de mémoïsation *qui sont discutées dans la section 22.1). Dans d'autres situations, on préfère, par exemple,* recalculer une valeur *plutôt que la garder en mémoire.*

Tables de hachage avec chaînage

Dans une table de hachage avec chaînage, la *fonction de hachage* nous donne une adresse de la table qui contient un *pointeur à une liste d'éléments*. Le coût du calcul dépend de la *longueur de la liste*. Il faut faire en sorte que les listes aient une *longueur comparable* (en moyenne). Si c'est le cas, le *coût* est proportionnel au facteur de charge α.

Une heuristique pour la fonction de hachage

Le but est d'avoir une fonction de hachage *uniforme*. A savoir, $h : U \to T$ telle que pour $k_1, k_2 \in U$ la *probabilité de collision est* $1/m$ *avec* $m = \sharp T$. Une *heuristique* possible est :

— Voir une clé $k \in U$ comme un entier.

— Choisir m *premier* et pas trop proche d'une puissance de 2 et définir :

$$h(k) = k \mod m \ .$$

Choix probabiliste de la fonction hachage

Un utilisateur malicieux pourrait dégrader les performances d'une table de hachage en proposant des données qui génèrent un grand nombre de collisions. On peut se défendre contre ce type d'attaque en choisissant la fonction de hachage de *façon aléatoire* (et en gardant ce choix secret). Cette stratégie rappelle celle adoptée dans le tri rapide (section 17.2) pour se défendre contre un choix défavorable des données à trier.

Avec un choix aléatoire, on peut alors garantir qu'*en moyenne* le hachage est *uniforme*, c'est à dire la probabilité d'une collision de deux clés est au plus $1/m$. Il se trouve qu'il suffit d'échantillonner les fonctions de hachage parmi certaines fonctions affines en arithmétique modulaire qui peuvent être représentées de façon compacte.

Construction

— Soient $k_1, k_2 \in U$ avec $k_1 \neq k_2$.

— On construit un *ensemble de fonctions de hachage* $\mathcal{H}$ tel qu'en tirant avec probabilité uniforme $h \in \mathcal{H}$ on a :

$$P(h(k_1) = h(k_2)) \leq 1/m \ . \tag{20.1}$$

— D'abord, on cherche p *premier* tel que $\sharp U \leq \sharp \mathbf{Z}_p$. On peut donc voir toute clé comme un entier modulo p et on prend les fonctions de la forme : [1]

$$x \mapsto ((ax + b) \mod p) \mod m \qquad a \in \mathbf{Z}_p^*, b \in \mathbf{Z}_p \ .$$

— Soit :

$$\mathcal{F} = \{f : \mathbf{Z}_p \to \mathbf{Z}_p \mid f(x) = (ax + b) \mod p, a \in (\mathbf{Z}_p)^*, b \in Z_p\} \ .$$

On a $\sharp\mathcal{F} = p(p-1)$.

— Soit :

$$\mathcal{P} = \{(x, y) \in \mathbf{Z}_p \times \mathbf{Z}_p \mid x \neq y\} \ .$$

On a aussi $\sharp\mathcal{P} = p(p-1)$.

— Soit $i : \mathcal{F} \to \mathcal{P}$:

$$i((a, b)) = ((ak_1 + b) \mod p, \ (ak_2 + b) \mod p) \ .$$

On vérifie que i est *injective*. Donc si on tire $f \in \mathcal{F}$ de façon uniforme on obtient un élément dans $\mathcal{P}$ avec une probabilité uniforme.

— Soit :

$$\mathcal{H} = \{((\_) \mod m) \circ f : \mathbf{Z}_p \to \mathbf{Z}_m \mid f \in \mathcal{F}\} \ .$$

— La *probabilité d'une collision* est donc la probabilité qu'en tirant deux points $x \neq y$ dans $\mathbf{Z}_p$ avec une probabilité uniforme on a :

$$(x \equiv y) \mod m \ .$$

— Si l'on fixe $x \in \mathbf{Z}_p$, le *nombre d'éléments* $z \in \mathbf{Z}_p$ tels que $x \neq z$ et $(x \equiv z) \mod m$ est au plus $\lceil p/m \rceil - 1$.

— On remarque (écrivez p comme $km + r$) :

$$\lceil p/m \rceil - 1 \leq \frac{(p + m - 1)}{m} - 1 = \frac{(p-1)}{m} \ .$$

Donc si on tire au hasard un élément différent de x, la probabilité d'une collision est au plus :

$$\frac{p-1}{m(p-1)} = \frac{1}{m} \ .$$

En d'autres termes, si on tire au hasard $(a, b) \in (\mathbf{Z}_p)^* \times \mathbf{Z}_p$:

$$P(((ak_1 + b) \mod p \equiv (ak_2 + b) \mod p) \mod m) \leq \frac{1}{m} \ .$$

1. On note au passage qu'en prenant les fonctions de la forme $x \mapsto (ax+b) \mod m$ avec $a \in \mathbf{Z}_m^*$ et $b \in \mathbf{Z}_m$ on n'obtient pas la propriété attendue (20.1) : si $(k_1 \equiv k_2) \mod m$ alors les valeurs hachées coïncident.

20.3 Tables de hachage avec adressage ouvert

On considère un deuxième schéma de mise en oeuvre d'une table de hachage.
— La *fonction de hachage* donne une adresse de la table.
— A partir de cette adresse une deuxième *fonction de sondage* (*probing* en anglais) donne une *suite d'adresses de la table* à visiter.
— La fonction de sondage doit permettre de visiter *toutes les adresses* de la table.

L'*élimination* d'un élément dans une table avec adressage ouvert est compliquée. Une solution populaire consiste à remplacer l'élément par une valeur spéciale DELETED. Une *insertion* peut avoir lieu dans une place marquée DELETED. Une *recherche* doit continuer après un élément DELETED. Les cellules DELETED entraînent une *dégradation des performances* et en général on évite l'approche avec adressage ouvert si l'utilisation de la structure comporte des éliminations.

Conception de la fonction de sondage

On discute deux approches à la conception de la fonction de sondage : le sondage *linéaire* et le *double hachage*.

Dans le *sondage linéaire*, on va parcourir :

$$h(k) \mod m, (h(k)+1) \mod m, \ldots, (h(k)+(m-1)) \mod m \ .$$

Cette approche a tendance à créer des *longues chaînes*.

Une approche plus sophistiquée utilise une technique de *double hachage*. A savoir, on introduit une deuxième fonction :

$$h_{aux} : U \to (\mathbf{Z}_m)^* \ ,$$

et on calcule pour $a = h_{aux}(k)$:

$$h(k), (h(k)+a) \mod m, \ldots, (h(k)+(m-1)a) \mod m \ .$$

Si m est *premier*, il suffit de prendre $m' = (m-1)$ et

$$a = h_{aux}(k) = (k \mod m') + 1 \in (\mathbf{Z}_m)^* \ ,$$

car $x \mapsto ax + b : \mathbf{Z}_m \to \mathbf{Z}_m$ est *injective* si $a \in (\mathbf{Z}_m)^*$.

Analyse de l'hachage ouvert

On suppose une fonction de *hachage uniforme* et un *facteur de charge* $\alpha = n/m < 1$. On cherche à déterminer le *nombre moyen* de sondages pour conclure qu'un élément *n'est pas* dans l'ensemble. Soit X la v.a.d. qui *compte le nombre de sondages*. On montre que :

$$E[X] \leq \frac{1}{1-\alpha} \ .$$

Soit A_i l'événement où l'on *sonde pour la i-ème fois une cellule occupée*. On a $(X \geq 1) = \Omega$ et pour $2 \leq i \leq n$:

$$(X \geq i) = A_1 \cap A_2 \cap \cdots \cap A_{i-1} \ .$$

En utilisant la *probabilité conditionnelle* :

$$\begin{aligned} P(X \geq i) &= P(A_1) \cdot P(A_2 \mid A_1) \cdot P(A_3 \mid A_1 \cap A_2) \cdots \\ &\quad \cdots P(A_{i-1} \mid A_1 \cap \cdots \cap A_{i-2}) \ . \end{aligned}$$

On dérive, en utilisant l'hypothèse d'*hachage uniforme* :

$$\begin{aligned} P(X \geq i) &= \tfrac{n}{m} \tfrac{n-1}{m-1} \cdots \tfrac{n-i+2}{m-i+2} \\ &\leq (\tfrac{n}{m})^{i-1} \\ &= \alpha^{i-1} \ . \end{aligned}$$

On a donc :

$$\begin{aligned} E[X] &\leq \Sigma_{i=1,\ldots,\infty} i \cdot P(X = i) \\ &= \Sigma_{i=1,\ldots,\infty} i \cdot (P(X \geq i) - P(X \geq i+1)) \\ &= \Sigma_{i=1,\ldots,\infty} P(X \geq i) \\ &\leq \Sigma_{i=1,\ldots,\infty} \alpha^{i-1} \\ &= \Sigma_{i=0,\ldots,\infty} \alpha^{i} \\ &= \tfrac{1}{1-\alpha} \ . \end{aligned}$$

Exemple 55 *Dans l'approche avec adressage ouvert, on* épargne la mémoire *pour les pointeurs mais la cardinalité de l'ensemble représenté est* bornée *par la taille de la table et* DELETED dégrade *les performances. Avec toutes ces réserves, considérons une situation où l'adressage ouvert se compare favorablement au chaînage.*

Supposons qu'une clé et un pointeur prennent le même espace *et que les problèmes associés aux bornes et aux* DELETED *ne se posent pas. Si une table de* hachage avec chaînage *a un facteur de charge* $\alpha = 2$ *on utilise* m *cellules pour la table et* $4m$ *cellules pour les listes. Avec l'*adressage ouvert*, on peut donc avoir un facteur de charge* $\alpha' = 2/5$ *et une recherche qui échoue effectue en moyenne* $\frac{1}{1-2/5} = 5/3 < 2$ *sondages.*

En conclusion, mentionnons 2 variations possibles sur le thème des tables de hachage : les tables *dynamiques* et les tables *parfaites*.

Dans les tables de hachage *dynamiques*, on prévoit la possibilité d'élargir ou réduire *dynamiquement* la taille de la table de hachage de façon à garder le *facteur de charge* dans un certain intervalle. De plus, dans certaines applications on souhaite élargir ou réduire de façon *incrémentale*. En d'autres termes, on répartit le travail de gestion des tables dynamiques sur toutes les opérations de façon à garantir la réactivité du système.

Parfois, on connaît à l'avance les n éléments qui peuvent être dans l'ensemble. On peut alors allouer une table de taille n et garantir un temps d'accès constant dans le *pire des cas* (plutôt qu'en moyenne). On parle dans ce cas de tables de hachage *parfaites*.

Chapitre 21

Algorithmes gloutons

Un algorithme *glouton* (*greedy* en anglais) est un algorithme qui cherche une solution à un problème en suivant un critère d'optimum local. En général cette approche peut être vue comme une *heuristique* mais dans certains cas elle permet de trouver la solution de façon *optimale* et *efficace*. En particulier, dans le cadre 'continu' de l'*optimisation convexe*, on sait qu'un optimum local coïncide toujours avec l'optimum global. Dans un cadre 'discret', on présente deux exemples de cette situation favorable qui concerne la recherche d'une sous-séquence contiguë maximale et la recherche d'une compression optimale. Le chapitre 24 proposera aussi deux autres exemples dans le cadre des graphes pondérés.

21.1 Sous-séquence contiguë maximale

On considère le problème de la sous-séquence contiguë maximale (abrégé en *SCM*).

Entrée Une séquence $x_1, \ldots, x_n$ dans $\mathbf{Z}$.

Sortie Un couple (i, j) tel que :

$$s_{i,j} = max\{s_{\ell,m} \mid 1 \leq \ell \leq m \leq n\} \ ,$$

où :

$$s_{\ell,m} = \Sigma_{k=\ell,\ldots,m} x_k \ .$$

Une interprétation possible du problème SCM est la suivante : on joue n tours et x_i représente le gain ou la perte au tour i ($i \in \{1, \ldots, n\}$). On cherche à déterminer la série de tours (=sous-séquence contiguë) dans laquelle on gagne le plus (ou on perd le moins...).

Remarque 21 *Dans la suite on se focalise sur le calcul de $s_{i,j}$ et on laisse comme exercice le problème de calculer le couple (i, j) associé.*

On fait l'hypothèse que la séquence est mémorisée dans un *tableau* ce qui permet d'accéder chaque élément du tableau en *temps constant*. Pour calculer le résultat il faut au moins lire chaque élément de la séquence. Donc on ne peut pas faire mieux que $O(n)$. On va *pratiquer* une approche 'directe' et une approche 'diviser pour régner' avant d'arriver à l'approche 'gloutonne'.

Approche directe

On calcule :

$$\begin{array}{lll} s_{1,1}, \quad s_{2,2}, \quad \cdots \quad , s_{n,n} & (n \text{ longueur } 1) \\ \quad s_{1,2}, \quad \cdots \quad , s_{n-1,n} & (n-1 \text{ longueur } 2) \\ \qquad \cdots & \cdots \\ \qquad s_{1,n} & (1 \text{ longueur } n) \end{array}$$

et on remarque que :

$$s_{i,j+1} = s_{i,j} + x_{j+1} \ .$$

Le *coût total* est donc :

$$n + (n-1) + \cdots + 2 + 1 = \frac{n(n+1)}{2} \ .$$

A savoir : $O(n^2)$ en *temps*.

Exercice 25 *Montrez qu'on peut calculer la SCM en utilisant une quantité linéaire de mémoire.*

Approche diviser pour régner

Que se passe-t-il si on cherche à *diviser le problème en* 2 comme dans la recherche dichotomique ou le tri par fusion ? On fixe un peu de notation.

- $SCM(i,j)$ est le problème de déterminer une *SCM* entre i et j.
- $SCMD(i,j)$ est le problème de déterminer une *SCM* entre i et j et qui *termine à* j (à *D*roite).
- $SCMG(i,j)$ est le problème de déterminer une *SCM* entre i et j et qui *commence à* i (à *G*auche).

Remarque 22 *Soit $m = (i+j)/2$. Pour calculer $SCM(i,j)$ (où $i < j$) on calcule :*

- $v_1 = SCM(i,m)$.
- $v_2 = SCM(m+1,j)$.
- $v_3 = SCMD(i,m)$ *et* $v_4 = SCMG(m+1,j)$.

et on prend :

$$max\{v_1, v_2, v_3 + v_4\} \ .$$

On remarque pour $i < j$:

$$SCMD(i,j) = max\{x_j, x_j + SCMD(i,j-1)\} \ .$$

Il en suit que le calcul de $SCMD(i,j)$ est $O(j-i)$. Et de même pour $SCMG$. On retrouve la récurrence du *tri par fusion* (chapitre 15) :

$$C(n) = 2 \cdot C(n/2) + n \ .$$

Soit un coût $O(n \log n)$. Est-ce possible de faire mieux ?

Approche gloutonne

On abrège :

$$m_i = SCMD(1, i) \qquad \text{pour } 1 \leq i \leq n \ .$$

La remarque suivante est attribuée à J. Kadane [Ben84] :

— Le max parmi $m_1, \ldots, m_n$ nous donne une solution à $SCM(1, n)$. En effet une SCM entre 1 et n va bien terminer à un i tel que $1 \leq i \leq n$ et la *même SCM* va être une solution pour le problème $SCMD(1, i)$.

— On sait déjà qu'on peut calculer m_{i+1} à partir de m_i en *temps constant* car :

$$m_{i+1} = max\{m_i + x_{i+1}, x_{i+1}\} \ .$$

On peut donc résoudre le problème en $O(n)$ ce qui est *optimal*.

Exercice 26 *Programmez les* 3 *approches et testez leur efficacité.*

21.2 Compression de Huffman

On considère un problème de *compression* de l'information. On fixe un *alphabet fini* $A = \{a_1, \ldots, a_m\}$ avec m symboles. On suppose que chaque symbole de l'alphabet paraît avec *probabilité* p_i, $i = 1, \ldots, m$. On cherche une fonction $C : A \to 2^*$ qui associe à chaque symbole un *code binaire* tel que :

— le codage est *décodable* : pour tout mot $b \in 2^*$ il existe au plus un mot $w \in A^*$ tel que $C(w) = b$.

— On *minimise la longueur moyenne* du codage d'un symbole, à savoir la quantité :

$$\Sigma_{i=1,\ldots,m}\ p_i \cdot |C(a_i)| \ .$$

On peut voir $b \in 2^*$ comme un chemin dans un arbre binaire et l'ensemble des codes $\{C(a_1), \ldots, C(a_n)\}$ comme le plus petit *arbre binaire* qui contient les chemins associés aux codes.

Définition 19 (propriété du préfixe) *Un codage a la* propriété du préfixe *(ou est préfixe) si deux codes différents ne sont jamais l'un le préfixe propre de l'autre.*

Dans la représentation à arbre d'un codage préfixe, les *codes sont exactement les feuilles de l'arbre*. Le *décodage d'un codage préfixe est simple* : on lit le code de gauche à droite et on décode dès qu'on reconnaît le code d'un symbole. Il se trouve que sans perte de généralité on peut se *restreindre aux codage préfixes* ! En d'autres termes, on peut toujours trouver un codage qui est optimal et préfixe.

Exercice 27 *Soit* $A = \{1, 2, 3, 4\}$. *On considère* 3 *candidats pour la fonction* C *:*

	P	C_1	C_2	C_3
1	0,5	0	0	0
2	0,3	1	10	01
3	0,1	00	110	011
4	0,1	01	111	111

Questions. (1) Quels codes sont décodables *? (2) Quels codes sont* préfixes *? (3) Quelle est la* longueur moyenne *du codage d'un symbole ?*

On reformule le problème de la façon suivante : construire un arbre binaire T avec m feuilles (=codes) et affecter les probabilités des m symboles aux m feuilles de façon à minimiser la longueur moyenne des chemins de la racine aux feuilles (qu'on dénote par $\ell(T)$).

Exercice 28 *Montrez que :*

1. *On peut supposer que l'arbre est* plein *(un noeud est une feuille ou a 2 fils).*
2. *Si la (distribution de) probabilité des symboles est* uniforme *alors on peut supposer que l'arbre optimal est* quasi-complet *(voir définition 10).*
3. *Il y a des distributions pour lesquelles la solution optimale est une* liste.

On va introduire deux transformations de l'arbre T.

Transformation 1 Soit T un *arbre* avec m feuilles avec probabilités $p_1 \leq p_2 \leq \cdots \leq p_n$. Soit T' l'arbre obtenu comme suit :

> on prend un noeud avec deux fils qui sont des feuilles de probabilité q_1 et q_2 ($q_1 \leq q_2$) qui est à *distance maximale* de la racine. On *"permute"* q_1 avec p_1 et p_2 avec q_2 (cas dégénérés laissés en exercice).

On *vérifie* que :

$$\ell(T') \leq \ell(T) \ .$$

Transformation 2 Prenez l'arbre T' de la transformation 1 et dérivez un arbre T'' en *remplaçant* le noeud avec feuilles p_1 et p_2 par un *seul noeud* qui est une feuille de probabilité $p_1 + p_2$. On *vérifie* que :

$$\ell(T') = \ell(T'') + (p_1 + p_2)$$

On utilise ces deux transformations pour montrer que la construction suivante produit un codage préfixe optimal [Huf52].

Construction de Huffman On associe aux probabilités $p_1, \ldots, p_m$, avec $p_1 \leq \cdots \leq p_m$, un *arbre* T_m avec m feuilles comme suit :

— Si $m = 1$ on a une feuille avec poids p_1.
— Si $m = 2$ on a 3 noeuds dont deux feuilles de poids p_1 et p_2.
— Si $m > 2$ on construit un arbre T_{m-1} pour les probabilités :

$$p_1 + p_2, p_3, \ldots, p_m \ ,$$

> ensuite on obtient l'arbre T_m en *remplaçant* la feuille de poids $p_1 + p_2$ avec un noeud avec deux feuilles de poids p_1 et p_2.

On a donc :

$$\ell(T_m) = \ell(T_{m-1}) + (p_1 + p_2) \ .$$

Remarque 23 *On reconnaît dans cette construction une stratégie gloutonne : pour résoudre le problème pour $p_1, \ldots, p_m$ avec $p_1 \leq \cdots \leq p_m$ on va résoudre le problème pour $p_1 + p_2, p_3, \ldots, p_m$ et ensuite expanser la feuille associée à $p_1 + p_2$.*

Proposition 21 *La construction de Huffman donne un* code préfixe optimal.

PREUVE. Par *récurrence* sur m. Les cas $m = 1$ et $m = 2$ sont clairs. Si $m > 2$ on sait par récurrence que l'arbre T_{m-1} pour $p_1 + p_2, p_3, \ldots, p_m$ est *optimal* et que :

$$\ell(T_m) = \ell(T_{m-1}) + (p_1 + p_2) \ .$$

Par *contradiction*, supposons T optimal pour $p_1, \ldots, p_m$ et $\ell(T) < \ell(T_m)$. On applique la *transformation 1* à T et on obtient un arbre T' avec $\ell(T') \leq \ell(T)$. Ensuite, on applique la *transformation 2* à T' et on obtient un arbre T'' avec $\ell(T') = \ell(T'') + (p_1 + p_2)$. On a donc :

$$\ell(T'') + (p_1 + p_2) = \ell(T') \leq \ell(T) < \ell(T_m) = \ell(T_{m-1}) + (p_1 + p_2) \ ,$$

qui *contredit l'optimalité* de T_{m-1} $(\ell(T'') < \ell(T_{m-1}))$! □

Mise en oeuvre

On considère des arbres où chaque noeud contient la *somme des probabilités* des feuilles accessibles depuis le noeud (donc la racine de l'arbre contient la somme des probabilités des feuilles de l'arbre). Initialement, on a m *arbres* constitués d'une seule feuille avec probabilités $p_1, \ldots, p_m$. On maintient les arbres dans un *min-tas* (chapitre 14), ordonnés d'après les valeurs des racines. A chaque étape, on *extrait* les deux arbres plus petits t_1 et t_2 du tas et on y *insère* un nouveau arbre obtenu en ajoutant un *noeud* qui pointe à t_1 et t_2 et dont la probabilité est la somme des probabilités de t_1 et t_2. On répète cette opération $m - 1$ *fois* pour obtenir l'arbre T_m.

Exercice 29 *Déterminez la complexité asymptotique de la fonction qui construit l'arbre T_m.*

Remarque 24 *La construction de Huffman s'applique aussi dans les cas où :*

1. *on associe aux symboles des* poids *(des nombres non-négatifs) plutôt que des probabilités.*
2. *on utilise pour le codage au lieu d'un alphabet binaire un alphabet avec $k > 2$ symboles.*

Chapitre 22

Programmation dynamique

La *programmation dynamique* est une branche de l'*optimisation combinatoire* popularisée par R. Bellman [Bel54]. [1] En *programmation dynamique*, on déduit la solution optimale d'un problème en combinant les solutions optimales d'une série de *sous problèmes* et ces sous-problèmes sont en *nombre raisonnable* (polynomial). [2]

22.1 Techniques de programmation

On considère la situation suivante : le calcul d'une fonction f dans un point x demande le calcul préalable de f dans Q(|x|) points, Q polynôme. On distingue 3 approches.

Calcul descendant (top-down)

On définit une fonction récursive. Disons :

```
f(x){int r; P; return r}
```

où l'on suppose que P ne contient *pas de* return et n'a *pas d'effet de bord* visible. Le *coût* est souvent *exponentiel* car on recalcule f plusieurs fois sur les mêmes points. Un exemple typique est la programmation récursive de la fonction de Fibonacci :

```
int fibo(int n){
int r;
if (n==0){r=0;}
  else {if (n==1){r=1;}
     else {r=fibo(n-2)+fibo(n-1);}} ;
return r;}
```

Calcul descendant avec mémoïsation

On *transforme* la fonction récursive en utilisant une *table de hachage* T qui *mémoïse* le graphe de la fonction.

```
int f(x){int r; r=value(T,x);
     if (undefined(r)){P; insert(T,x,r);}
     return r;}
```

1. La terminologie répond à une logique 'commerciale' plutôt que 'scientifique'...
2. Le terme *programmation* est utilisé ici comme synonyme de *planification*.

Chaque point est calculé *une fois*. La table T contient à la fois la clé (l'entrée) et la valeur de la fonction qui est récupérée avec la fonction value. Le prédicat undefined nous dit si la fonction a été déjà calculée sur l'entrée x et à défaut la fonction insert insère la valeur calculée. Il est possible de remplacer la table de hachage par une autre structure de données. Par exemple, par un arbre binaire de recherche ou une liste à enjambements.

Exercice 30 *1. Appliquer la transformation au calcul de la suite de Fibonacci.*

2. Vous disposez d'un générateur aléatoire dans $\mathbf{2} = \{0, 1\}$. *Pour effectuer une simulation vous avez besoins de tirer des fonctions dans* $[\mathbf{2}^{128} \to \mathbf{2}^{128}]$ *avec probabilité uniforme. Comment faire ?*

3. Quid si l'on veut simuler une permutation sur $\mathbf{2}^{128}$ *?*

Calcul ascendant (bottom-up)

On *réorganise* le calcul de façon à que le calcul de f(x) soit précédé par le calcul de tous les f(y) où y est *'plus petit'* que x. Ce calcul est typiquement stocké dans une *table*.

22.2 Calcul de la plus longue sous-séquence commune

Soient α, β des *mots* (=séquences finies) sur un alphabet. On dénote par $|\alpha|$ la longueur de la séquence et par ϵ la séquence de longueur 0.

Définition 20 (sous-séquence) α *est un* sous-séquence *de* β *si l'on obtient* α *de* β *en supprimant un certain nombre d'éléments de la séquence.*

Définition 21 (*lcs) On dénote par* $lcs(\alpha, \beta)$ une *plus* plus longue sous-séquence commune *(*longest common subsequence*, en anglais, abrégé en lcs) des mots* α *et* β.

Remarque 25 *La lcs n'est pas forcement unique. Par exemple, considérez les mots ABC et ACB et en général le nombre de lcs peut être exponentiel. Notre but sera juste de calculer* une *lcs.*

Exemple 56

$$\begin{array}{ll} \alpha & = ACCGGTCGAGTGCGCGGAAGCCGGCCGAA \\ \beta & = GTCGTTCGGAATGCCGTTGCTCTGTAAA \\ lcs(\alpha, \beta) & = GTCGTCGGAAGCCGGCCGAA \end{array}$$

On peut voir la lcs comme une mesure de la similarité *de deux séquences (par exemple, deux séquences d'ADN). Nombreuses variations existent : problème de l'*alignement de séquences*,* distance d'édition*,...*

Définition 22 (*llcs) On définit la* longueur *de la lcs :* $llcs(\alpha, \beta) = |lcs(\alpha, \beta)|$.

Proposition 22 *La longueur de la plus longue sous-séquence commune satisfait les propriétés suivantes :*

$$\begin{array}{ll} llcs(\epsilon, \alpha) & = llcs(\alpha, \epsilon) = 0 \\ llcs(a\alpha, a\beta) & = 1 + llcs(\alpha, \beta) \\ llcs(a\alpha, b\beta) & = \mathsf{max}(llcs(a\alpha, \beta), llcs(\alpha, b\beta)) \ . \end{array}$$

On peut utiliser la fonction *llcs* pour calculer la fonction *lcs*. En particulier, si *llcs* a été pré-calculée, le calcul de *lcs* est linéaire en $max(|\alpha|, |\beta|)$.

Proposition 23

$$\begin{array}{ll} lcs(\epsilon, \alpha) & = lcs(\alpha, \epsilon) = \epsilon \\ lcs(a\alpha, a\beta) & = a \cdot lcs(\alpha, \beta) \\ lcs(a\alpha, b\beta) & = \begin{cases} lcs(a\alpha, \beta) & si\ llcs(a\alpha, \beta) \geq llcs(\alpha, b\beta) \\ lcs(\alpha, b\beta) & autrement. \end{cases} \end{array}$$

On se focalise maintenant sur le calcul de la fonction *llcs* et on considère les 3 méthodes évoquées dans la section 22.1. Supposons $|\alpha| = n$ et $|\beta| = m$. La proposition 22 nous dit qu'il faut calculer *llcs* sur $O(nm)$ points.

— Un calcul *descendant récursif* va appeler la fonction *llcs* un nombre exponentiel de fois. Pour vous en convaincre, considérez la recurrence suivante.

$$\begin{array}{ll} C(n, 0) & = C(0, n) = 1 \\ C(n+1, m+1) & = C(n, m+1) + C(n+1, m) \geq 2 \cdot C(n, m)\ . \end{array}$$

— Un calcul *descendant récursif va mémoïser* $O(nm)$ valeurs avant de rendre un résultat. Ensuite le calcul de *lcs* se fait en $O(max(m, n))$.

— Un calcul ascendant peut calculer *llcs par diagonal*. Par exemple si $n = m = 3$:

$$\begin{array}{lll} (1,3) & (2,3) & (3,3) \\ (1,2) & (2,2) & (3,2) \\ (1,1) & (2,1) & (3,1)\ . \end{array}$$

On peut calculer dans l'ordre :

$$(3,3), (3,2), (2,3), (1,3), (2,2), (3,1), (2,1), (1,2), (1,1)\ .$$

A nouveau le calcul de *lcs* se fait en $O(max(m, n))$.

Remarque 26 *Le calcul ascendant calcule* toutes *les cellules du tableau alors que le calcul descendant avec mémoïsation en calcule un* sous-ensemble. *Le cas le plus favorable pour le calcul descendant est si* $\alpha = \beta$ *et le pire est si* α *et* β *n'ont pas de caractères communs. En général, le calcul descendant avec mémoïsation est excellent pour un* prototypage efficace *alors que le calcul ascendant est utilisé (si besoin) pour une* optimisation *plus poussée.*

22.3 Algorithme CYK

Une *grammaire* est une façon de spécifier un *langage formel*, à savoir un ensemble de mots sur un alphabet. Les grammaires sont classifiées selon la *forme des règles* utilisées. Dans les *grammaires algébriques* (*context-free* en anglais ou *hors-context* en franglais), les règles ont la forme :

$$A \to A_1 \cdots A_n$$

avec l'*interprétation* suivante : toute occurrence du symbole A peut être remplacée par les symboles $A_1, \ldots, A_n$. Des *sous-classes des grammaires algébriques* (par exemple *LR(1)*) sont utilisées pour spécifier la *syntaxe* des langages de programmation et des outils automatiques

(par exemple Yacc) construisent un *programme d'analyse syntaxique* (le *parseur* en franglais) à partir de la grammaire.

On va considérer les grammaires en *forme normale de Chomsky* (FNC). Toute grammaire algébrique peut être transformée en une grammaire en FNC équivalente (à quelques détails près).

Définition 23 (forme normale de Chomsky) *Une* grammaire *en forme normale de Chomsky (FNC) est spécifiée par :*

- *Un ensemble fini* $\mathcal{N}$ *de* symboles non-terminaux *avec un* symbole initial $S \in \mathcal{N}$.
- *Un ensemble fini* Σ *de* symboles terminaux.
- *Une ensemble fini de* règles $\mathcal{R}$ *qui ont la forme :*

$$A \to a \quad ou \quad A \to BC$$

avec $A, B, C \in \mathcal{N}$ *et* $a \in \Sigma$.

Exemple 57 *Voici un exemple de grammaire FNC qui décrit les suites de parenthèses 'bien formées'.*

$$\begin{array}{lll} \mathcal{N} & = \{S, L, R, A\} & \textit{Non-terminaux} \\ \Sigma & = \{(,)\} & \textit{Terminaux (ou alphabet)} \\ S & \to LR & \textit{Règles} \\ S & \to SS & \\ S & \to LA & \\ A & \to SR & \\ L & \to (& \\ R & \to) & \end{array}$$

Par exemple, le mot $()(())$ *est généré de la façon suivante :*

$$\begin{array}{llllll} S & \to SS & \to LRS & \to (RS & \to ()S & \to ()LA \\ & \to ()(A & \to ()(SR & \to ()(LRR & \to ()((RR & \to ()(()R & \to ()(()) \ . \end{array}$$

Problème On considère le problème de la *reconnaissance de mots* par une grammaire. A savoir, pour toute grammaire G en FNC on cherche un algorithme qui prend en entrée un mot w sur l'alphabet Σ et décide s'il est possible de générer w à partir du symbole initial S de la grammaire.

Notation Comme dans la section précédente on dénote par $|w|$ la *longueur* du mot w. On dénote aussi par $w[i, j]$ le *sous-mot* de w compris entre les positions i et j ($1 \leq i \leq j \leq |w|$). On écrit $G(A, i, j)$ ssi le symbole A de la grammaire G peut générer $w[i, j]$. Avec cette notation, le *problème à résoudre* est équivalent à savoir si $G(S, 1, |w|)$. La proposition suivante nous donne une stratégie pour calculer $G(A, i, j)$.

Proposition 24 *Pour toute grammaire* G *en FNC et* w *mot.*

- $G(A, i, i)$ *ssi* $A \to w[i, i]$ *est un règle dans* $\mathcal{R}$.
- *Si* $i < j$,

$$G(A, i, j) = \bigvee_{\substack{A \to BC \in \mathcal{R} \\ k = i, \ldots, j-1}} \left(\ G(B, i, k) \ \wedge \ G(C, k+1, j) \ \right)$$

On considère maintenant l'application des 3 stratégie évoquées dans la section 22.1.

Calcul descendant (top-down)

On définit une fonction récursive d'après la proposition 24. On commence par appeler $G(S, 1, n)$. Le coût est exponentiel.

Calcul descendant avec mémoïsation

On définit une fonction récursive d'après la proposition 24 qui utilise en plus une *table de hachage* T (par exemple).

— A chaque *appel* de $G(A, i, j)$ on regarde d'abord si le résultat est déjà dans T.

— Sinon, à chaque *retour* de $G(A, i, j)$ on *mémoïse* le résultat dans T.

Le coût est cubique si l'accès à la table est en $O(1)$. On a $O(n^2)$ points à calculer et le travail pour chaque point est $O(n)$. [3]

Calcul ascendant (bottom-up)

En supposant $\mathcal{N} = \{A_1, \ldots, A_m\}$, $S = A_1$ et $|w| = n$, on ordonne le calcul comme suit :

$$\begin{gathered} G(A_1,1,1),\ldots\ldots,G(A_1,n,n),\ldots,G(A_m,1,1),\ldots\ldots G(A_m,n,n) \\ G(A_1,1,2),\ldots,G(A_1,n-1,n),\ldots,G(A_m,1,2),\ldots G(A_m,n-1,n) \\ \cdots,\ldots,\cdots,\ldots,\cdots,\ldots,\cdots \\ G(A_1,1,n) \end{gathered}$$

Le coût est aussi cubique. L'algorithme est connu comme algorithme CYK en référence aux noms des concepteurs (Cocke, Younger et Kasami).

Exercice 31 *Un* graphe dirigé étiqueté avec racine *est un tuple* (N, A, r, L) *où :*

— N *est l'ensemble (fini) des* noeuds*,*

— $r \in N$ *est la* racine*,*

— $A \subseteq N \times N$ *est l'ensemble des* arêtes*,*

— $L : A \to \Sigma$ *étiquette chaque arête un symbole d'un* alphabet Σ. [4]

Proposez un algorithme qui pour chaque mot fini $w = a_1 \ldots a_n$ *sur* Σ *détermine s'il y a un chemin dans le graphe qui commence par la racine et passe par des arêtes étiquetées par* $a_1, \ldots, a_n$*.*

Exercice 32 *Soient* A_i *des* matrices *de dimension* $d_{i-1} \times d_i$ *pour* $i = 1, \ldots n$*. On suppose que le* produit *de deux matrices de dimension* $x \times y$ *et* $y \times z$ *prend* $O(xyz)$*. Comment peut-on déterminer la* façon optimale d'associer *les matrices pour calculer* $A_1 \cdots A_n$ *? Par exemple, si* $n = 3$ *le nombre de multiplications est :*

$$\begin{array}{ll} \textit{Association à gauche :} & d_0d_1d_2 + d_0d_2d_3 \\ \textit{Association à droite :} & d_0d_1d_3 + d_1d_2d_3 \end{array}$$

Pour $d_0 = 10$*,* $d_1 = 1$*,* $d_2 = 10$*,* $d_3 = 1$*, associer à gauche coûte* 10 fois plus cher *que associer à droite.*

---

3. Pour l'analyse syntaxique des langages de programmation, on utilise des grammaires algébriques particulières (*LR(1)*) qui admettent un algorithme de reconnaissance en $O(n)$.

4. Il s'agit d'un cas particulier d'*automate fini non-déterministe* (AFN) dans lequel chaque état est accepteur.

Chapitre 23

Graphes

Les graphes sont des structures omniprésentes en informatique dont les listes et les arbres sont des cas particuliers. Dans ce chapitre, on introduit les méthodes principales pour représenter les graphes finis et pour les visiter.

23.1 Représentation

Définition 24 (graphe dirigé) *Un* graphe dirigé *est un couple* $G = (N, A)$ *où* N *est l'ensemble des* noeuds *et* $A \subseteq N \times N$ *est l'ensemble des* arêtes.

Convention On s'intéresse aux graphes finis et on fixe $n = \sharp N$ pour la cardinalité des noeuds et $m = \sharp A$ pour la cardinalité des arêtes. Dans un graphe dirigé, on a :

$$0 \leq m \leq n^2 \ .$$

Si m est linéaire en n on dira que le graphe est *creux* (*sparse* en anglais) et si m s'approche de n^2 on dira qu'il est *dense*.

Variantes On trouve dans la littérature une grande variété de définitions dont voici certaines.

- Graphes non-dirigés : dans ce cas les arêtes ne sont pas dirigées. On a donc : $0 \leq m \leq \frac{n(n-1)}{2}$.
- Graphes étiquetés : les noeuds ou les arêtes ont des valeurs associés (voir, par exemple, les tas du chapitre 14, les BDD du chapitre 17 et les ABR du chapitre 18).
- Multi-graphes : plusieurs arêtes entre deux noeuds permises.
- Hyper-graphes : une arête peut connecter plus que 2 noeuds.

Terminologie Voici des terminologies souvent utilisées.

- Deux noeuds sont *adjacents* s'il y a une arête qui les connecte.
- Le *degré* d'un noeud est le nombre de noeuds qui lui sont adjacents. Pour un graphe dirigé on distingue le *degré entrant* et le *degré sortant*.
- Un *chemin* dans un graphe est une suite de noeuds $i_1, \ldots, i_k$ tel que $(i_j, i_{j+1}) \in A$ pour $j = 1, \ldots, k-1$. On dit que $k-1$ est la *longueur* du chemin.
- Un chemin est *simple* s'il n'y a pas de répétition de noeuds. Donc dans un chemin simple on a au plus n noeuds.

— Un *circuit* est un chemin de longueur positive dont le premier et dernier noeud sont identiques (pour un graphe non-dirigé il faut préciser qu'on ne peut pas utiliser la même arête 2 fois).
— Un graphe *acyclique* est un graphe sans circuits.
— S'il y à un chemin de i à j on dit que i est *connecté* à j. Dans le cas des graphes dirigés, on dit que deux noeuds sont *fortement connectés* si i est connecté à j et j à i.
— La relation de connexité forte est une relation d'équivalence et on appelle *composantes fortement connexes* ses classes d'équivalence.

Les arbres sont un cas particulier de graphes. On se place dans le cadre des graphes *non-dirigés*.

Proposition 25 *Soit $G = (N, A)$ un graphe non-dirigé. Les conditions suivantes sont équivalentes.*
— *G est connecté et acyclique.*
— *G est connecté et a $n - 1$ arêtes.*
— *il existe un chemin unique qui connecte chaque couple de noeuds.*

On peut prendre une de ces 3 conditions comme définition d'arbre. On remarquera que ces arbres diffèrent des arbres utilisés dans les chapitres 14 et 18. En effet, dans les arbres dont il est question ici, il n'y a pas de racine, les arêtes sont non-dirigées et non-ordonnées (on ne distingue pas entre arête gauche et arête droite) et le nombre de noeuds adjacents n'est pas borné.

Soit $G = (N, A)$ un graphe dirigé avec $N = \{1, \ldots, n\}$. Les deux représentations principales qu'on va considérer sont les *matrices d'adjacence* et les *listes d'adjacence*.

Matrice d'adjacence Une matrice M $n \times n$ de booléens telle que :

$$M[i, j] = 1 \text{ ssi } (i, j) \in A \ .$$

Liste d'adjacence Un tableau T de taille n tel que l'entrée $T[i]$ pointe à une liste qui contient exactement les noeuds j tels que $(i, j) \in A$

On utilisera surtout les listes d'adjacence dont la *taille* est $O(n + m)$ par opposition au $O(n^2)$ d'une matrice d'adjacence ce qui est avantageux dans les graphes creux.

Remarque 27 *Si le graphe est non-dirigé alors la matrice d'adjacence est symétrique et en pratique il suffit de manipuler le triangle supérieur (ou inférieur) de la matrice. La représentation d'un graphe non-dirigé avec une liste d'adjacence peut poser des problèmes d'efficacité dans certains cas. Par exemple, supposons que la liste $T[i]$ contient le noeud j et qu'on souhaite éliminer l'arête $\{i, j\}$ du graphe. Dans ce cas, on doit aussi éliminer i de la liste $T[j]$ et pour réaliser cette opération en temps constant l'introduction de pointeurs additionnels peut être nécessaire. Par exemple, on peut faire en sorte que chaque élément j dans la liste d'adjacence du noeud i pointe vers l'élément i dans la liste d'adjacence du noeud j.*

Les deux exercices qui suivent utilisent la représentation par matrice d'adjacence.

Exercice 33 *Un* puit *(*sink *en anglais) dans un graphe dirigé est un noeud avec degré entrant $n-1$ et degré sortant 0. On suppose que le graphe est représenté par une matrice d'adjacence M et que le temps d'accès à un élément de la matrice est $O(1)$. Soient $i, j \in N$ avec $i \neq j$. On remarque la propriété suivante : si i est un puit alors $M[j, i] = 1$ et $M[i, j] = 0$. Proposez un algorithme en $O(n)$ (on a donc pas le droit de regarder toutes les arêtes !) qui décide s'il y a un noeud puit dans le graphe et dans ce cas donne sa position.*

Exercice 34 *Soit M la* matrice d'adjacence *d'un graphe. Soit :*

$$M^0 = I \quad M^{k+1} = M^k M \ .$$

1. *Montrez que $M^k[i,j]$ est égal au* nombre de chemins *entre i et j de longueur k.*
2. *Quid si on travaille avec des matrices dans $\{0,1\}$ avec comme addition et multiplication la disjonction et la conjonction logique, respectivement ?*

23.2 Visite d'un graphe

On introduit un algorithme pour visiter un graphe à partir d'un noeud désigné comme racine. On fait l'hypothèse que pour chaque noeud i on dispose d'un bit de marquage mark[i] qui est initialement à 0.

Entrée Un graphe et un noeud racine $r \in N$.

Algorithme

```
W = {r};
while(W ≠ ∅){
i = remove(W);
if(!mark[i]){
   mark[i] = 1;
   ∀j if((i,j) ∈ A && !mark[j]){insert(j,W);} } }
```

Analysons cet algorithme. Pour l'instant on suppose que W est un *multi-ensemble* et que remove enlève un élément du multi-ensemble. En spécialisant la structure W on pourra mettre en oeuvre différentes stratégie de visite (en largeur, en profondeur,...).

— Chaque fois qu'on insère un élément dans W on a une arête (i,j) tel que le noeud i vient d'être marqué et le noeud j n'est pas marqué.

Le nombre d'insertions est borné par m (nombre d'arêtes) !

— Chaque noeud inséré dans W est accessible depuis la racine. Donc chaque noeud marqué est accessible depuis la racine.

— Chaque noeud accessible depuis la racine est marqué. En effet soit $(r,i_1),\ldots,(i_k,j)$ un chemin de longueur minimal vers un noeud qui n'est pas marqué par l'algorithme. Mais alors i_k est accessible avec un chemin plus court et il est marqué. Donc j sera inséré dans W et il sera marqué car l'algorithme termine avec W vide. Contradiction.

Stratégies de visite Les *stratégies de visite* dépendent de la mise-en-oeuvre de remove et insert. Les 2 stratégies principales sont :

En largeur (*breadth-first*)	W est une *queue* (*first-in first out*)
En profondeur (*depth-first*)	W est une *pile* (*last-in first-out*)

Chaque stratégie a des applications intéressantes (voir suite). On rappelle que si W est une *queue* ou une *pile* les opérations remove et insert coûtent $O(1)$.

23.3 Visite en largeur et distance

On peut utiliser la recherche en largeur pour calculer la longueur du chemin le plus court entre le noeud racine et les autres noeuds (qu'on abrège en *distance*). Pour ce faire, on *initialise un tableau* d comme suit :

$$d(i) = \begin{cases} 0 & \text{si } i = r \\ +\infty & \text{autrement} \end{cases}$$

L'algorithme de recherche est *modifié* comme suit (où W est une *queue*) :

```
W = {r};
while(W ≠ ∅){
i = remove(W);
if(!mark[i]){
   mark[i] = 1;
   ∀j if((i,j) ∈ A && !mark[j]){
      if(d[j] == +∞){d[j] = d[i] + 1;}
      insert(j,W);} } }
```

Propriété L'algorithme est $O(n+m)$ car on examine chaque arête au plus une fois et à la fin de l'algorithme $d[i]$ est la *distance* de r à i ($+\infty$ si i n'est pas accessible depuis r). Le fait qu'on utilise une *queue* assure qu'un noeud 'proche' de r est toujours traité avant un noeud 'éloigné de r.

23.4 Visite en profondeur et tri topologique

Dans ce cas W est une *pile*. Alternativement, on peut obtenir le même effet en utilisant la *pile implicite* qui gère les appels récursifs et on obtient le programme suivant.

```
void dfs(i){
if(!mark[i]){
   mark[i] = 1;
   ∀j if((i,j) ∈ A && !mark[j]){dfs(j);} }}
```

Remarque 28 *Il est possible d'effectuer une visite en profondeur* sans pile et sans récursion. *Il suffit de réserver dans chaque noeud un nombre de bits* logarithmique dans le degré du noeud. *Ensuite on utilise une technique d'*inversion de pointeurs *:*

— *si on descend en profondeur, on inverse le pointeur pour se souvenir d'où on vient.*

— *si on remonte, on remet le pointeur à sa place.*

Ce type d'algorithme (connu comme algorithme de Schorr-Waite*) est utilisé lorsque la mémoire est précieuse. E.g., dans un ramasse miettes (*garbage collector*, en anglais).*

On considère maintenant une application de la visite en profondeur au problème dit du *tri topologique.*

Définition 25 (tri topologique) *Un* tri topologique *d'un graphe* dirigé $G = (N, A)$ *avec* n *noeuds est une fonction* $\ell : N \to \{1, \ldots, n\}$ *telle que :*

$$(i, j) \in A \text{ implique } \ell(i) < \ell(j) \ .$$

Un tri topologique est une façon de *linéariser* l'ordre partiel induit par les arêtes. La linéarisation :

— est *unique* ssi le graphe est une *liste*.

— *existe* ssi le graphe est *acyclique*.

Pour calculer un tri topologique d'un graphe acyclique il suffit d'enrichir la version récursive de la fonction dfs comme suit.

— On ajoute un *tableau* ℓ et un *compteur* count qui est initialisé à n.

— La fonction dfs est modifiée pour qu'avant le retour d'un appel dfs(i) on enregistre dans ℓ[i] la position du noeud i dans le tri en on décrémente count (si au lieu de la récursion on utilise une pile, la même idée s'applique).

```
void dfs(i){
if(!mark[i]){
   mark[i] = 1;
   ∀j if((i,j) ∈ A && !mark[j]){dfs(j);}
   ℓ[i] = count; count − −; }}
```

— Comme le graphe peut être *disconnecté*, pour avoir un tri complet il faut dans le pire des cas appeler dfs sur tous les noeuds.

```
void dfs_loop(){
int i = 1;
while (count > 0 && i <= n){dfs(i); i = i + 1; }}
```

— Si l'on ne sait pas à l'avance si le graphe est acyclique on peut quand même calculer le tableau ℓ et ensuite *vérifier la condition* :

$$(i,j) \in A \text{ implique } \ell[i] < \ell[j]$$

Elle sera satisfaite ssi le graphe est acyclique. On a donc un algorithme $O(n)$ pour savoir si un graphe est acyclique.

Les exercices suivants explorent des notions classiques de théorie des graphes.

Exercice 35 *Soit $G = (N, E)$ un* graphe non-dirigé. *G est* k-coloriable *s'il y a une fonction $c : N \to \{1, \ldots, k\}$ telle que si les noeuds i et j sont adjacents alors $c(i) \neq c(j)$. Il est* facile *de décider si un graphe est* 1*-coloriable et* difficile *(*NP*-complet) de décider s'il est k-coloriable pour $k \geq 3$. Programmez une fonction (efficace !) qui décide si un graphe est* 2-coloriable.

Exercice 36 *Un* circuit simple *dans un graphe non-dirigé est un chemin qui revient au point de départ sans jamais passer deux fois par la même arête. Un* circuit Eulerien *est un circuit simple qui passe par chaque arête du graphe.*

— *Montrez qu'un graphe* non-dirigé connecté *a un* circuit Eulerien *ssi chaque noeud a* degré pair.

— *Programmez un* algorithme *qui construit un circuit Eulerien pour un graphe connecté G avec noeuds de degré pair comme suit.*

1. *Il construit au hasard un* circuit simple γ.
2. *Tant que* γ n'est pas un circuit Eulerien *:*
   (a) *Il cherche un* noeud i *dans* γ *avec une arête* $\{i, j\}$ *pas encore dans* γ.

(b) Il construit un circuit *simple γ' à partir de $\{i, j\}$ avec les arêtes qui ne sont pas dans γ.*

(c) Il combine γ et γ' pour obtenir un nouveau circuit simple *γ.*

Chapitre 24

Graphes pondérés

Soit $G = (N, A)$ un graphe *non-dirigé* $G = (N, A)$ *connecté* et où les arêtes ont un *poids non-négatif* :

$$w : A \to \mathbf{R}^+ \ .$$

Définition 26 (ARM) *Un arbre de recouvrement minimum (ARM) pour un graphe G est un arbre qui est un sous-graphe de G qui contient tous les noeuds de G et dont la somme des poids des arêtes est minimum.*

Définition 27 (PCC) *Un arbre des plus courts chemins dans un graphe G depuis un noeud désigné comme source (PCC) est un arbre qui est un sous-graphe de G qui contient tous les noeuds de G et dont les chemins du noeud source aux autres noeuds sont les plus courts.*

Il n'y a pas de perte de généralité à supposer que la solution aux problèmes ARM et PCC sont des arbres. En effet si on a un graphe qui est une solution optimale on peut toujours élaguer certaines arêtes jusqu'à obtenir un arbre.

Notez aussi que l'ARM et le PCC peuvent différer. Par exemple, considérez le graphe :

$$n_1 \overset{2}{-} n_2 \overset{2}{-} n_3 \quad n_1 \overset{3}{-} n_3 \ .$$

Alors, l'ARM est $\{\{n_1, n_2\}, \{n_2, n_3\}\}$ mais le PCC depuis 1 est $\{\{n_1, n_2\}, \{n_1, n_3\}\}$.

Dans ce chapitre, on va introduire des algorithmes efficaces (quasi-linéaires) pour résoudre ces problèmes qui utilisent une *stratégie gloutonne* (chapitre 21) et la structure de données *tas* (chapitre 14).

24.1 Algorithme de Prim pour le recouvrement minimum

On présente l'algorithme de Prim [Pri57] pour calculer l'ARM d'un graphe.

Initialisation On partitionne N en N_1, N_2 avec $\sharp N_1 = 1$. L'arbre T est vide.

On itère $n-1$ fois

— Parmi les arêtes $\{i, j\}$ avec $i \in N_1$ et $j \in N_2$, soit $\{i_0, j_0\}$ une qui *minimise* :

$$w(\{i, j\}) \ .$$

— On pose :

$$N_1 = N_1 \cup \{j_0\},\ N_2 = N_2 \backslash \{j_0\},\ T = T \cup \{\{i_0, j_0\}\} \ .$$

Argument Voici l'argument pour prouver que la stratégie gloutonne calcule bien l'ARM.

- D'abord on remarque que si on a un *arbre* et on ajoute une arête on a un *circuit* et si on enlève une arête quelconque du circuit on a à nouveau un *arbre*.
- L'algorithme de Prim construit un *arbre* en ajoutant $n-1$ arêtes, disons $a_1, \ldots, a_{n-1}$.
- Soit a_i la *première arête* qui est incompatible avec un ARM. Disons que a_i connecte les noeuds $j \in N_1$ et $k \in N_2$.
- Soit T un ARM qui contient les arêtes $a_1, \ldots, a_{i-1}$.
- On obtient une *contradiction* en montrant que T peut être transformé en un ARM qui contient les arêtes $a_1, \ldots, a_i$.
- Ajoutons l'arête a_i à T. On a donc un *circuit*.
- Dans le circuit il doit y avoir au moins *une autre arête* a qui connecte un noeud dans N_1 avec un noeud dans N_2.
- Par définition de l'algorithme de Prim, le *poids* de a_i est inférieur ou égal au poids de a.
- Donc si on enlève l'arête a on obtient à nouveau un arbre et même un *ARM*. L'arête a_i n'est donc pas la première qui pose problème. Contradiction.

24.2 Algorithme de Dijkstra pour les plus courts chemins

On présente l'algorithme de Dijkstra [Dij59] pour calculer le PCC depuis un noeud racine.

Initialisation On partitionne N en N_1, N_2 avec $\sharp N_1 = 1$. L'arbre T est vide. On suppose que N_1 contient le *noeud source* s. L associe le *poids du chemin* de la source à un noeud dans N_1. Au début on a $L(s) = 0$.

On itère $n-1$ fois

- Parmi les arêtes $\{i, j\}$ avec $i \in N_1$ et $j \in N_2$, soit $\{i_0, j_0\}$ une qui *minimise* :

$$L(i) + w(\{i, j\}) \ .$$

- On pose :

$$\begin{array}{ll} N_1 = N_1 \cup \{j_0\}, & N_2 = N_2 \backslash \{j_0\}, \\ T = T \cup \{\{i_0, j_0\}\}, & L(j_0) = L(i_0) + w(\{i_0, j_0\}) \ . \end{array}$$

Argument Voici l'argument pour prouver que la stratégie gloutonne calcule bien l'arbre PCC.

- Chaque fois qu'on calcule $L(j_0) = L(i_0) + w(\{i_0, j_0\})$, $L(j_0)$ est bien le *poids du plus court chemin* de la racine à j_0.
- En effet un plus court chemin γ de la racine à j_0 doit comprendre une arête qui connecte un noeud $k \in N_1$ avec un noeud $k' \in N_2$.
- On a :

$$\begin{array}{ll} w(\gamma) \geq L(k) + w(\{k, k'\}) & \text{(par hyp. de récurrence)} \\ L(k) + w(\{k, k'\}) \geq L(i_0) + w(\{i_0, j_0\}) & \text{(par construction)} \end{array}$$

Remarque 29 *La* visite en largeur *étudiée dans la section 23.2 nous donne déjà les plus courts chemins quand le poids de chaque arête est* 1. *On pourrait imaginer la transformation*

suivante d'un graphe pondéré avec des nombres naturels en un graphe ordinaire : on transforme une arête de poids n en n arêtes de poids 1 *en introduisant des noeuds intermédiaires. Le problème avec cette transformation est qu'elle est* exponentielle *dans le nombre de bits nécessaires à représenter les poids.*

Remarque 30 *L'algorithme de Prim pour le calcul de l'ARM s'applique aussi en présence de poids* négatifs*. Par contre, la stratégie gloutonne pour le calcul des PCC est* myope*. Exemple :*

$$n_1 \overset{5}{-} n_2 \overset{-3}{-} n_3 \qquad n_1 \overset{3}{-} n_3 \ .$$

24.3 Une autre application de la structure tas (cas de Dijkstra)

On va analyser la complexité de l'algorithme de Dijkstra dans le cas où on utilise la structure tas (chapitre 14)

- On maintient un *tas*. Chaque élément du tas est composé de : (i) un *noeud*, (ii) une estimation de sa *distance* de la racine, (iii) une estimation du noeud prédécesseur.
- Notez qu'à partir d'une table de prédécesseurs il est facile de construire les listes d'adjacence qui représentent l'arbre des PCC.
- Les éléments du tas sont ordonnés par ordre croissant par rapport à la distance (on a donc un *min-tas*).
- On maintient aussi un *tableau* T avec autant d'éléments que de noeuds. Chaque élément contient un pointeur à la position du noeud dans le tas (donc *chaque modification* de la position d'un élément dans le tas doit être enregistrée dans le tableau).

Remarque 31 *Le tas contient des* noeuds *pas des* arêtes *!*

Complexité de l'algorithme de Dijkstra Avec ces *structures de données* on peut en $O(\log n)$:

- Extraire le *min du tas*.
- *Mettre à jour* (diminuer) l'estimation de la distance d'un élément du tas et de son prédécesseur. Ceci est possible car le tableau T nous donne un *accès direct* à la position de l'élément dans le tas et ensuite il suffit de permuter avec le père autant que nécessaire.

Avec un peu de travail, on obtient une *complexité* $O(m \log n)$.

Remarque 32 *Des arguments similaires s'appliquent à l'*algorithme de Prim*.*

Chapitre 25

Flot maximum et coupe minimale

On peut voir un graphe comme un réseau de transport et les pondérations des arêtes comme une mesure de la capacité de transport de chaque arête. Si on fixe un noeud 'source' et un noeud 'destination' une question naturelle est celle de maximiser la quantité que l'on peut transporter de l'origine à la destination. En termes plus techniques, on considère le problème de maximiser le *flot* (définition à suivre) dans un graphe dont les arêtes ont des capacités bornées (problème MaxFlot).

Il se trouve que ce problème admet un problème *dual* qui consiste à rechercher une *coupe* minimale du graphe. Cette notion de *coupe* minimale est aussi facile à motiver. Par exemple, si toutes les capacités des arêtes sont identiques, une coupe minimale du graphe consiste à déterminer le nombre minimum d'arêtes qu'il faut couper pour disconnecter le noeud source du noeud destination. On a ici un premier exemple de *dualité* en optimisation combinatoire. Typiquement cette dualité prend la forme suivante : un problème de maximisation admet un problème dual de minimisation tel que les solutions des deux problèmes, si elles existent, alors elles coïncident. En particulier, dans ce chapitre on montrera que le flot maximum coïncide avec la coupe minimale [FF56].

Ce résultat de dualité couplé avec la notion de *chemin augmentant* est la base pour la conception d'algorithmes efficaces (polynomiaux) pour le problème MaxFlot. Le problème MaxFlot est aussi intéressant pour d'autres raisons.

— Le problème peut être formalisé comme un problème de *programmation linéaire* (chapitre 26) à savoir un problème de maximisation d'une fonction linéaire sujette à des contraintes linéaires.

— Quand les capacités sont des entiers, le problème MaxFlot peut aussi être vu comme un problème de *programmation entière*, à savoir un problème de programmation linéaire où en plus on demande à que les solutions soient des entiers. Dans ce contexte, le problème MaxFlot a la propriété remarquable que le maximum du problème de programmation linéaire coïncide avec le maximum du problème de programmation entière.

25.1 Flots et coupes

Définition 28 (capacité) *Soit N un ensemble fini (de noeuds). Une capacité est une fonction $c : N^2 \to \mathbf{R}^+$ qui associe un nombre non négatif à chaque couple de noeuds.*

Une capacité est une fonction. Par extension, on parlera aussi de capacité d'une arête et

dans ce cas il s'agira d'un nombre réél positif. Si on prend comme ensemble des arêtes :

$$A = \{(i,j) \in N^2 \mid c(i,j) > 0\} \ ,$$

on obtient un graphe dirigé et pondéré (voir chapitre 24). [1] Par exemple, si une arête modélise un tuyau alors la capacité peut correspondre au nombre de litres par second qui peuvent transiter dans le tuyau. Pour d'autres exemples, on peut s'inspirer des réseaux électriques ou routiers.

On distingue deux noeuds différents s (source) et d (destination) et on s'intéresse à la question de trouver le 'flot' maximal de s à d que le réseau peut supporter.

Définition 29 (flot) *Un flot est une fonction $f : N^2 \to \mathbf{R}$ qui satisfait les 3 conditions suivantes :* [2]

symétrie $\forall i,j \in N \ (f(i,j) = -f(j,i))$.

conservation [3] $\forall i \in N \backslash \{s,d\} \ \Sigma_{j \in N} \ f(i,j) = 0$.

capacité $\forall i,j \in N \ (f(i,j) \leq c(i,j))$.

L'intuition est que $f(i,j)$ décrit la quantité de flot *net* qui peut aller de i à j. La première condition exprime le fait que le flot de i à j doit être l'opposé du flot de j à i. Cette condition implique qu'on a toujours $f(i,i) = 0$. La deuxième condition est un principe de conservation du flot : dans tous les noeuds sauf s et d le flot 'entrant' doit être égal au flot 'sortant'. Enfin la troisième condition impose un flot compatible avec la capacité de l'arête. En particulier : (i) si $c(i,j) = c(j,i) = 0$ alors on doit avoir $f(i,j) = f(j,i) = 0$ et (ii) si $f(i,j) < 0$ alors $|f(i,j)| \leq c(i,j)$ (le flot négatif est aussi borné).

Exemple 58 *Supposons un graphe avec deux noeuds et les capacités (non nulles) suivantes : $c(s,d) = 4$ et $c(d,s) = 2$. Alors tout flot doit satisfaire $f(d,d) = f(s,s) = 0$. Par ailleurs, si on pose $f(s,d) = 2$ alors on doit avoir $f(d,s) = -2$ ce qui est compatible avec les capacités. On peut remarquer qu'il existe plusieurs façons de 'réaliser' concrètement ce flot* net. *A savoir, s envoie x à d pour $2 \leq x \leq 4$ et d envoie $(x-2)$ à s.*

Définition 30 (coupe) *Une coupe est une partition de l'ensemble de noeuds N en deux ensembles A et B tels que $s \in A$ et $d \in B$.*

Définition 31 (capacité d'une coupe) *La capacité $c(A,B)$ d'une coupe (A,B) est :*

$$c(A,B) = \Sigma_{i \in A, j \in B} \ c(i,j)$$

De façon similaire, si f est un flot on pose $f(A,B) = \Sigma_{i \in A, j \in B} \ f(i,j)$. Il se trouve que cette quantité depend du flot mais pas de la coupe.

Proposition 26 *Soit f un flot. Alors :*

1. *la valeur $f(A,B)$ est constante (ne depend pas du choix de la coupe).*

1. On remarquera qu'on retient dans A seulement les couples avec capacité strictement positive.
2. On retrouve ces mêmes conditions dans d'autres contextes. Par exemple, les *lois de Kirchoff* jouent un rôle similaire dans l'étude de circuits électriques.
3. Dans l'analyse de circuits électriques, cette loi est connue comme loi de Kirchoff.

2. $f(A,B) \leq c(A,B)$.

PREUVE. (1) On pose :

$$|f| = f(\{s\}, N\backslash\{s\}) \ .$$

On montre que pour toute coupe (A,B), $f(A,B) = |f|$. On procède par récurrence sur $\sharp A$. Le cas $\sharp A = 1$ suit de la définition de $|f|$. Sinon soit (A,B) une coupe avec $A = A' \cup \{i\}$ et $i \neq s$. Par hypothèse de récurrence, on sait :

$$|f| = f(A', B \cup \{i\}) = f(A', B) + f(A', \{i\}) \ .$$

D'autre part, $f(A,B) = f(A',B) + f(\{i\},B)$. Par symétrie, $f(A',\{i\}) = -f(\{i\},A')$ et par conservation, $f(\{i\},A') + f(\{i\},B) = 0$. Donc $f(A',\{i\}) = f(\{i\},B)$ et on peut conclure que $|f| = f(A,B)$.
(2) Par la condition sur la capacité. □

On peut donc définir la valeur $|f|$ d'un flot comme le flot $f(A,B)$ par rapport à une coupe (A,B) quelconque.

$$|f| = f(A,B) \quad (A,B) \text{ coupe} \tag{25.1}$$

Définition 32 (problème flot maximum) *Le problème du flot maximum est le problème de déterminer pour toute capacité donnée $c : N^2 \to \mathbf{R}^+$, un flot de valeur maximale.*

Définition 33 (problème coupe minimale) *Le problème de la coupe minimale est le problème de trouver pour toute capacité donnée $c : N^2 \to \mathbf{R}^+$, une coupe de capacité minimale.*

25.2 Chemin augmentant et graphe résiduel

Soit f un flot pour le graphe dirigé G construit à partir d'un ensemble de noeuds N et la capacité c. On définit la *capacité résiduelle* comme la fonction $r = c - f$ (qui est bien une capacité d'après la définition 28). La capacité résiduelle sur une arête (i,j) représente donc l'augmentation maximale du flot $f(i,j)$. On définit aussi le *graphe résiduel* comme le graphe G_f construit à partir du même ensemble de noeuds et la capacité résiduelle r. Dans le graphe G_f l'ensemble des arêtes est :

$$A_f = \{(i,j) \mid r(i,j) > 0\} \ .$$

Notez que le graphe G_f peut avoir une arête que le graphe initial G n'a pas. Par exemple, on pourrait avoir $c(i,j) = 3$, $f(i,j) = 1$ et $c(j,i) = 0$. Dans ce cas, $r(i,j) = (3-1) = 2$ et $r(j,i) = (0-(-1)) = 1$.

Définition 34 (chemin augmentant) *Un chemin augmentant est un chemin simple dans le graphe résiduel G_f qui va du noeud source s au noeud destination d.*

Définition 35 (obstruction) *L'obstruction d'un chemin augmentant est la plus petite capacité qu'on trouve sur le chemin.*

Proposition 27 *Soit f un flot pour le graphe G et soit G_f le graphe résiduel.*

1. *La fonction f' est un flot pour G_f ssi $f + f'$ est un flot pour G.*

2. *Si $f + f'$ est un flot maximum pour G alors f' est un flot maximum pour G_f.*
3. *Si f, f' sont des flots alors $|f \pm f'| = |f| \pm |f'|$.*
4. *Si f est un flot et f^m est un flot maximum pour G alors la valeur d'un flot maximum dans G_f est $|f^m| - |f|$.*

PREUVE. Par manipulation élémentaire des propriétés de symétrie, conservation et capacité d'un flot (définition 29). □

Proposition 28 *Soit f un flot pour le graphe G. Alors les propriétés suivantes sont équivalentes.*

1. *Il y a une coupe (A, B) telle que $|f| = c(A, B)$.*
2. *f est un flot maximum.*
3. *Il n'y a a pas de chemin augmentant dans le graphe résiduel G_f.*

PREUVE. (1) $\Rightarrow$ (2) Par la proposition 26, on sait que si f' est un flot alors :

$$|f'| \leq c(A, B) = |f| \ .$$

Le flot f est donc maximum. Notez aussi que la coupe (A, B) doit être minimum car pour toute coupe (A', B') on a :

$$c(A, B) = |f| \leq c(A', B') \ .$$

(2) $\Rightarrow$ (3) Par contradiction, on suppose disposer d'un chemin augmentant dans G_f. Soit $d > 0$ la capacité plus petite dans ce chemin. On peut alors construire un flot f' dans G_f en posant :

$$f'(i, j) = \begin{cases} d & \text{si } (i, j) \text{ est dans le chemin} \\ -d & \text{si } (j, i) \text{ est dans le chemin} \\ 0 & \text{autrement} \end{cases}$$

Par la proposition 27, on dérive que $f + f'$ est un flot dans G et $|f + f'| = |f| + d > |f|$. Donc f n'est pas un flot maximum.
(3) $\Rightarrow$ (1) S'il n'y a pas de chemin augmentant dans G_f alors on peut construire une coupe (A, B) où A est l'ensemble des noeuds accessibles depuis s et $B = N \backslash A$. Dans cette coupe il n'y a pas d'arête de A dans B ce qui veut dire que pour $i \in A$ et $j \in B$ on a $r(i, j) = c(i, j) - f(i, j) = 0$. Donc $c(i, j) = f(i, j)$ et $f(A, B) = c(A, B)$. □

La proposition 28 implique que si les capacités sont des *entiers* alors la valeur du flot maximum est un entier car il est égal à la capacité d'une coupe minimale. La proposition 28 est aussi la base pour la conception d'un algorithme qui calcule le flot maximum en itérant la construction du graphe résiduel et la recherche d'un chemin augmentant.

Cet algorithme est correct et efficace (polynomial) à condition d'éviter certains écueils. Clairement on dispose d'algorithmes efficaces pour calculer un chemin augmentant. Une première stratégie pourrait donc consister à démarrer avec un flot nul est à itérer la recherche d'un chemin augmentant jusqu'à ce qu'il n'y en ait plus. Si les capacités sont des entiers cette stratégie termine avec le flot maximum. Cependant on peut trouver des suites de chemins augmentants dont la longueur est exponentielle dans le nombre de bits nécessaires à représenter les capacités.

Exemple 59 *Soit* $N = \{s, 1, 2, d\}$ *avec* $c(s,1) = c(s,2) = c(1,d) = c(2,d) = M$, $c(1,2) = 1$ *et* $c(i,j) = 0$ *autrement. Si on prend comme chemin augmentant* $s \xrightarrow{M} 1 \xrightarrow{1} 2 \xrightarrow{M} d$, *on obtient comme capacité residuelle :* $c(s,1) = c(2,d) = M - 1$, $c(s,2) = c(1,d) = M$, $c(2,1) = 1$ *et* $c(i,j) = 0$ *autrement. Maintenant on prend comme chemin augmentant :* $s \xrightarrow{M} 2 \xrightarrow{1} 1 \xrightarrow{M} d$. *En continuant de la sorte, on peut construire* M *graphes residuels avant de converger.*

Si les capacités sont des nombres irrationnels, on peut construire un exemple encore plus patologique. Dans cet exemple on produit une suite infinie de chemins augmentants, les augmentations suivent une loi géométrique et leur somme converge vers une valeur finie qui peut être aussi éloignée que l'on le souhaite du flot maximum. Cependant, ce contre-exemple a un impact limité car en pratique les capacités sont très souvent exprimées par des entiers ou des rationnels et dans ce dernier cas on peut toujours se ramener à un problème avec capacités entières en multipliant toutes les capacités par le plus petit dénominateur commun.

Avec des capacités entières, une stratégie simple (il y en a d'autres) qui permet d'obtenir une complexité polynomiale consiste à choisir toujours un chemin augmentant de longueur minimale [EK72]. Un tel chemin peut être calculé en effectuant une visite en largeur du graphe residuel (section 23.3) et ce calcul se fait en temps linéaire dans le nombre d'arêtes. Les meilleurs algorithmes permettent d'avoir une complexité en $O(n^3)$.

Chapitre 26

Programmation linéaire

Un problème de *programmation linéaire* est un problème d'optimisation d'une fonction linéaire sur un ensemble décrit par un ensemble d'inégalités linéaires (ou de façon équivalente sur un polyèdre convexe). [1]

26.1 Optimisation convexe

Un problème de programmation linéaire est un exemple particulier de problème d'optimisation *convexe*.

Définition 36 (ensemble convexe) *Un ensemble $S \subseteq \mathbf{R}^n$ est* convexe *si pour tout $x, y \in S$ et $\lambda \in [0,1]$ on a :*

$$\lambda x + (1-\lambda)y \in S \ .$$

Le point $\lambda x + (1-\lambda)y = \lambda(x-y) + y$ se trouve sur le segment déterminé par les points x et y. Ainsi un ensemble S est convexe si tout segment qui connecte deux points de l'ensemble est contenu dans S.

Exercice 37 *Montrez que l'intersection d'ensembles convexes est convexe.*

Définition 37 (fonction convexe) *Soient S un ensemble convexe dans $\mathbf{R}^n$ et $f : S \to \mathbf{R}$ une fonction. On dit que f est convexe si pour tout $x, y \in S$ et $\lambda \in [0,1]$ on a :*

$$f(\lambda x + (1-\lambda)y) \leq \lambda f(x) + (1-\lambda) f(y) \ .$$

On dit aussi que f est concave si $-f$ est convexe.

Dans une fonction convexe, le segment qui connecte deux points du graphe de la fonction domine la fonction.

Exemple 60 *Si $g_i : \mathbf{R}^n \to \mathbf{R}$ sont des fonctions convexes pour $i = 1, \ldots, m$ alors l'ensemble suivant est convexe :*

$$\{x \in \mathbf{R}^n \mid g_i(x) \leq 0, i = 1, \ldots, m\} \ .$$

1. Comme dans le cas de la *programmation* dynamique (chapitre 22), il faut comprendre *programmation* comme synonyme de *planification*.

Définition 38 (problème d'optimisation convexe) *Soient S un ensemble convexe et $f : S \to \mathbf{R}$ une fonction convexe. Le problème d'optimisation associé est le problème de trouver (s'il existe) un $x \in S$ qui minimise la fonction f :*

$$min\ \{f(x) \mid x \in S\}\ .$$

Un élément dans l'ensemble convexe S est dit admissible.

Une propriété remarquable d'un problème d'optimisation convexe est que si un élément est le minimum dans son *voisinage immédiat* alors il est aussi le minimum de tout l'ensemble S. Une situation idéale pour appliquer une stratégie gloutonne (voir chapitre 21).

Dans un problème d'optimisation convexe on peut adopter la stratégie suivante : on démarre avec un élément admissible $x_0 \in S$ et à chaque étape on vérifie si dans le voisinage immédiat de x_i il y a un élément admissible $x_{i+1} \in S$ tel que $f(x_{i+1}) < f(x_i)$. Si un tel élément n'existe pas on sait que x_i est une solution minimale, et sinon on continue la recherche à partir de x_{i+1}.

Pour formaliser la notion de voisinage immédiat on rappelle des notions standards de topologie.

Définition 39 (norme et distance euclidienne) *Si $x \in \mathbf{R}^n$ on dénote par $\|x\|$ sa norme euclidienne :*

$$\|x\| = \sqrt{\Sigma_{i=1,\ldots,n} x_i^2}\ . \tag{26.1}$$

On dérive de la norme la distance euclidienne :

$$\|x - y\| = \sqrt{\Sigma_{i=1,\ldots,n} (x_i - y_i)^2}\ .$$

Définition 40 (boule) *Soient S un ensemble convexe, $x \in S$ et $\epsilon > 0$. On définit la boule de centre x et rayon ϵ par :*

$$B(x, S, \epsilon) = \{y \in S \mid \|y - x\| \leq \epsilon\}\ .$$

Définition 41 (minimum local) *Soit S un ensemble convexe et $f : S \to \mathbf{R}$ une fonction convexe. On dit que $x \in S$ est un minimum local s'il existe $\epsilon > 0$ tel que pour tout $y \in B(x, S, \epsilon)$ on a $f(x) \leq f(y)$.*

Proposition 29 *Soient S un ensemble convexe, $f : S \to \mathbf{R}$ une fonction convexe et $x \in S$ un minimum local. Alors x est aussi un minimum de la fonction f sur S.*

Preuve. Soient $y \in S$ un autre élément de S. On veut montrer $f(y) \geq f(x)$. On sait que pour tout $\lambda \in [0, 1]$:

$$z = \lambda x + (1 - \lambda) y \in S\ .$$

En particulier, on peut toujours prendre $\lambda \in]0, 1[$ pour que :

$$\|z - x\| \leq \epsilon\ .$$

On peut cerner $f(z)$ de la façon suivante :

$$f(x) \leq f(z) = f(\lambda x + (1 - \lambda) y) \leq \lambda f(x) + (1 - \lambda) f(y)\ .$$

L'inégalité gauche utilise l'hypothèse que x est un minimum local et l'inégalité droite l'hypothèse que f est convexe. Comme $(1-\lambda) > 0$, on dérive :

$$f(x) = \frac{(1-\lambda)f(x)}{(1-\lambda)} \leq f(y) \ .$$

□

Remarque 33 *La proposition 29 est fausse si on remplace la recherche d'un minimum avec la recherche d'un maximum. Dans une fonction convexe, on peut avoir un maximum local qui n'est pas un maximum global. Si l'on s'intéresse à la propriété duale pour le maximum il faut prendre une fonction concave.*

26.2 Optimisation linéaire et problème dual

Définition 42 (fonctions affines et linéaires) *Une fonction affine sur* $\mathbf{R}^n$ *est une fonction de la forme :*

$$f(x_1, \ldots, x_n) = \Sigma_{i=1,\ldots,n}\ a_i \cdot x_i + b \ .$$

Si $b = 0$ *on dit aussi que la fonction est linéaire.*

Définition 43 (programmation linéaire) *Un problème de* programmation linéaire *est un problème d'optimisation convexe (définition 38) où l'ensemble convexe* S *est spécifié par un ensemble fini d'inégalités (voir exemple 60) :*

$$g_i(x) \leq 0 \quad g_i \text{ fonction affine}, i = 1, \ldots, m,$$

et la fonction f *à optimiser est une fonction linéaire.*

Remarque 34 *Une fonction affine est à la fois convexe et concave. Donc pour une fonction affine sur un ensemble convexe* S *un minimum (maximum) local est aussi un minimum (maximum) sur* S*.*

Il est possible de donner plusieurs formulations équivalentes d'un problème de programmation linéaire. Pour fixer les idées, on suppose que le problème est le suivant :

$$max\{c^T x \mid Ax \leq b, x \geq 0\} \ , \tag{26.2}$$

où $c, x, 0 \in \mathbf{R}^n$, $A \in \mathbf{R}[m, n]$ et $b \in \mathbf{R}^m$. Ce problème peut être transformé dans des problèmes équivalents en utilisant les transformations suivantes :

- La maximisation de la fonction $c^T x$ est équivalente à la minimisation de la fonction $(-c)^T x$.
- Une contrainte de la forme $a^T x \geq b$ est équivalente à la contrainte $(-a)^T x \leq -b$.
- Une contrainte de la forme $a^T x = b$ est équivalente aux contraintes $a^T x \leq b$ et $(-a)^T x \leq -b$.
- Une contrainte de la forme $a^T x \leq b$ est équivalente aux contraintes $a^T x + y = b$ et $y \geq 0$, où y est une nouvelle variable.
- Une contrainte de la forme $a^T x = b$ (qui ne suppose pas $x \geq 0$) est équivalente aux contraintes $a^T(x' - x'') = b$ et $x', x'' \geq 0$ où x, x' sont des nouvelles variables.

Exercice 38 *Formulez le problème du flot maximum comme un problème de programmation linéaire.*

Il se trouve que tout problème de programmation linéaire a un *problème dual.*[2] Cette propriété joue un rôle important dans la conception d'algorithmes pour la programmation linéaire et par ailleurs elle donne une façon systématique de reformuler un problème en passant par son problème dual. Souvent, la formulation duale donne un point de vue nouveau sur le problème. Dans ce contexte, l'intérêt de la formulation (26.2) de la programmation linéaire est que la forme du problème dual est facilement mémorisable.

Définition 44 (problème dual) *Le problème* dual *d'un problème de la forme (26.2) est :*

$$min\ \{b^T y \mid A^T y \geq c, y \geq 0\}\ . \tag{26.3}$$

Dans ce contexte, on appelle le problème (26.2) primal.

De façon symétrique, on définit le dual d'un problème de la forme :

$$min\{c^T x \mid Ax \geq b, x \geq 0\}\ ,$$

comme $max\{b^T y \mid A^T y \leq c, y \geq 0\}$. En utilisant le fait que $(A^T)^T = A$ on vérifie que le dual du dual est identique au problème de départ. On a donc la correspondance suivante :

Primal	Dual
$max\ c^T x$	$min\ b^T y$
$Ax \leq b$	$A^T y \geq c$
$x \geq 0$	$y \geq 0$.

Proposition 30 *Soient x admissible pour le problème primal (26.2) et y admissible pour le problème dual (26.3). Alors :*

$$c^T x \leq b^T y\ .$$

Preuve. On observe : $c^T x \leq (A^T y)^T x = y^T (Ax) \leq y^T b = b^T y$. A noter qu'on utilise l'hypothèse que $x, y \geq 0$. □

En d'autres termes, tout élément admissible du problème primal (dual) donne une borne inférieur (supérieure) pour la solution (si elle existe) du problème dual (primal).

Un problème primal (dual) peut :

1. avoir une solution admissible et optimale,
2. avoir une solution admissible mais pas de maximum (minimum) et
3. ne pas avoir de solution admissible.

La proposition 30 montre que si le primal (dual) est dans le cas 2. alors le dual (primal) est forcement dans le cas 3. Par ailleurs, la proposition 30 nous donne aussi un critère *suffisant* pour l'optimalité. Si x est admissible pour le primal, y est admissible pour le dual et $b^T y = c^T x$ alors x est optimal pour le primal et y est optimal pour le dual. Un corollaire de l'algorithme du simplexe qu'on discutera dans le prochain chapitre 27 est que si on a une solution optimale

2. Le concept de problème dual n'est pas propre à la programmation linéaire ; on retrouve ce concept aussi dans des problèmes d'optimisation convexe plus généraux que la programmation linéaire.

pour le primal (dual) alors on a aussi une solution optimale pour le dual (primal) et les fonctions coïncident sur ces solutions optimales. Donc il est *impossible* que le primal (dual) ait une solution optimale et le dual (primal) en ait pas et il est aussi *impossible* que la fonction de coût sur la solution optimale du primal soit strictement inférieure à la fonction de coût sur la solution optimale du dual. Une dernière situation possible est que primal et dual soient dans le cas 3. En résumant on a 4 cas *possibles* (sur 9) : $(1,1)$, $(2,3)$, $(3,2)$ et $(3,3)$. De plus, dans le cas $(1,1)$ les fonctions de coût sur les solutions optimales coïncident.

Chapitre 27

Algorithme du simplexe

L'algorithme du simplexe est un cas particulier de la méthode itérative de recherche d'un optimum local qu'on a esquissé dans la section 26.1. A chaque itération, on se trouve à un sommet du polyèdre qui contient les solutions admissibles et on détermine s'il est possible de se deplacer vers un sommet adjacent en améliorant la fonction objectif.

27.1 Formulation avec variables écart

L'intuition géométrique qu'on vient d'évoquer est très seduisante mais pour avoir un algorithme il faut lui donner un contenu algébrique et vérifier au passage que l'intuition en dimension 2 est valide en toute dimension finie. Pour ce faire, on introduit une formulation alternative de la programmation linéaire.

Un problème de la forme (26.2) est équivalent au problème

$$max\{c^T x \mid Ax + x' = b, x \geq 0, x' \geq 0\},$$

qui se réécrit aussi en :

$$max\{c_0 + c^T x \mid x' = b - Ax, x \geq 0, x' \geq 0\} \ . \tag{27.1}$$

Dans cette nouvelle forme :

— pour transformer l'inégalité en égalité, on introduit des *variables écart* (*slack variables*) x'.

— on réécrit la fonction de coût comme une fonction affine avec une constante additive c_0 qui dans notre cas vaut 0. A noter que les coefficients multiplicatifs pour les variables écart sont implicitement 0.

Pour l'instant, on va supposer que $b \geq 0$. Ainsi $(x, x') = (0, b)$ est une solution *admissible* du problème. Dans la forme (27.1), on appelle les variables x' *basiques* et les variables x *non-basiques*.

Problème initiale

On va introduire une méthode pour permuter une variable basique avec une variable non-basique. Pour formuler cette méthode il convient de réécrire la forme (27.1) de la façon suivante

où l'on suppose que N est l'ensemble des indices des variables non-basiques et B est l'ensemble des indices des variables basiques ; initialement, $N = \{1, \ldots, n\}$ et $B = \{n+1, \ldots, n+m\}$.

$$\begin{array}{ll} max\ c_0 + \Sigma_{i \in N} c_i x_i & \\ x_i = b_i - \Sigma_{j \in N} a_{i,j} x_j & i \in B \\ x_k \geq 0 & k \in B \cup N\ . \end{array} \tag{27.2}$$

Dans cette formulation, on suppose que $b_i \geq 0$ pour $i \in B$ ce qui est équivalent à dire que :

$$x_k = \begin{cases} b_k & \text{si } k \in B \\ 0 & \text{si } k \in N\ , \end{cases} \tag{27.3}$$

est admissible. On montrera dans la section 27.4 comment arriver à cette formulation ou conclure qu'une solution admissible n'existe pas.

Exemple 61

$max\ 3x_1 + x_2 + 2x_3$	$max\ 3x_1 + x_2 + 2x_3$
$x_1 + x_2 + 3x_3 \leq 30$	$x_4 = 30 - x_1 - x_2 - 3x_3$
$2x_1 + 2x_2 + 5x_3 \leq 24$	$x_5 = 24 - 2x_1 - 2x_2 - 5x_3$
$4x_1 + x_2 + 2x_3 \leq 36$	$x_6 = 36 - 4x_1 + x_2 + 2x_3$
$x_1, x_2, x_3 \geq 0$	$x_1, x_2, x_3, x_4, x_5, x_6 \geq 0$
Problème initial.	*Problème dérivé ; x_4, x_5, x_6 variables écart (basiques).*

Sélection du pivot

On détermine un indice 'entrant' $e \in N$ tel que :

$$c_e > 0\ . \tag{27.4}$$

Si aucun indice satisfait cette condition on montrera dans la section 27.3 que la solution (27.3) est optimale. L'intuition est que actuellement $x_e = 0$ et si on augmente x_e on a la possibilité d'augmenter la fonction objectif.

On détermine un indice 'sortant' $s \in B$ tel que :

$$\frac{b_s}{a_{s,e}} = min\ \{\frac{b_i}{a_{i,e}} \mid i \in B, a_{i,e} > 0\}\ . \tag{27.5}$$

L'intuition est qu'en augmentant x_e on risque de rendre les variables basiques négatives. On va selectionner s parmi les premières variables basiques qui vont tomber à 0. On remarque que si un tel indice n'existe pas on peut augmenter la valeur de la variable x_e de 0 à $+\infty$ tout en gardant $x_i \geq 0$ pour $i \in B$. Comme $c_e > 0$, il en suit que la fonction de coût va aussi à $+\infty$ et donc le problème est non-borné (le *max* n'existe pas). Les nouveaux ensembles de variables basiques et non-basiques sont donc :

$$\overline{B} = B \backslash \{s\} \cup \{e\}\ , \quad \overline{N} = N \backslash \{e\} \cup \{s\}\ .$$

Exemple 62 *En continuant l'exemple 61, on peut prendre comme variable entrante x_1 et comme variable sortante x_6.*

Mise à jour des coefficients

Une fois qu'on a déterminé la variable entrante et celle sortante on va effectuer une série de manipulations pour mettre les contraintes dans la forme (27.2) par rapport aux ensembles $\overline{B}$ et $\overline{N}$. A partir de l'équation :

$$x_s = b_s - \Sigma_{j \in N \backslash \{e\}} a_{s,j} x_j - a_{s,e} x_e \tag{27.6}$$

en divisant par $a_{s,e} > 0$, on peut exprimer la variable entrante en fonction de la sortante et des autres variables non-basiques :

$$x_e = \frac{b_s}{a_{s,e}} - \Sigma_{j \in N \backslash \{e\}} \frac{a_{s,j}}{a_{s,e}} x_j - \frac{1}{a_{s,e}} x_s \ . \tag{27.7}$$

En d'autres termes, les coefficients pour la variable entrante sont les suivants :

$$\overline{b}_e = \tfrac{b_s}{a_{s,e}} \ , \quad \overline{a}_{e,j} = \tfrac{a_{s,j}}{a_{s,e}} \quad (j \in N \backslash \{e\}), \qquad \overline{a}_{e,s} = \tfrac{1}{a_{s,e}} \ .$$

Il s'agit maintenant de remplacer toute occurrence de x_e dans les équations pour les autres variables basiques $i \in B \backslash \{s\}$ par l'expression à droite de l'équation (27.7). Un simple calcul permet de dériver :

$$\overline{b}_i = b_i - a_{i,e} \overline{b}_e \ , \quad \overline{a}_{i,j} = a_{i,j} - a_{i,e} \overline{a}_{e,j} \quad (j \in N \backslash \{e\}) \ , \qquad \overline{a}_{i,s} = -a_{i,e} \overline{a}_{e,s} \ .$$

On remarquera que le choix de la variable entrante assure que $\overline{b}_i \geq 0$ pour $i \in \overline{N}$. Enfin on effectue la même opération de remplacement pour la fonction objectif :

$$\overline{c}_0 = c_0 + c_e \overline{b}_e \ , \quad \overline{c}_j = c_j - c_e \overline{a}_{e,j} \quad (j \in N \backslash \{s\}), \qquad \overline{c}_s = -\overline{c}_e \overline{a}_{e,s} \ .$$

Exemple 63 *En continuant l'exemple 62, on obtient la forme écart suivante :*

$$\begin{aligned} max \ & 27 + \tfrac{x_2}{4} + \tfrac{x_3}{2} - \tfrac{3x_6}{4} \\ x_1 &= 9 - \tfrac{x_2}{4} - \tfrac{x_3}{2} - \tfrac{x_6}{4} \\ x_4 &= 21 - \tfrac{3x_2}{4} - \tfrac{5x_3}{2} - \tfrac{x_6}{4} \\ x_5 &= 6 - \tfrac{3x_2}{2} - 4x_3 + \tfrac{x_6}{2} \ . \end{aligned}$$

Remarque 35 *Comme dans l'élimination de Gauss pour la solution d'un système d'équations linéaires, les transformations décrites modifient la forme du problème sans affecter l'ensemble des solutions. La forme écart obtenue admet le mêmes solutions admissibles que celle de départ et pour toute solution x la valeur de la fonction de côut sur x coïncide avec celle de départ.*

27.2 Complexité

On note que les opérations effectuées ne modifient pas l'ensemble des solutions du problème initial et que le coût d'une mise à jour est $O(n \cdot m)$. On a donc obtenu une nouvelle forme écart et une nouvelle solution admissible avec une valeur de la fonction objectif qui est $\overline{c}_0 \geq c_0$. On a $\overline{c_0} = c_0$ si et seulement si $b_s = 0$. Il y a des situations dans lesquelles l'opération de pivot n'augmente pas la fonction objectif. Pour s'assurer de la *terminaison* de la méthode il faut remarquer que :

1. Chaque forme écart est déterminée par un choix de m variables basiques parmi $n+m$. On a donc au plus $\binom{n+m}{m}$ formes écart.
2. Il est possible de donner un critère de sélection (critère de Bland) de la variable entrante et sortante qui assure qu'on ne boucle jamais sur une suite de formes écart. Le critère consiste à ordonner de façon totale les indices des variables et à choisir toujours la variable entrante et sortante avec le plus petit indice parmi celles qui satisfont les conditions de sélection.

Une forme écart correspond à un sommet d'un polytope et l'opération de pivot correspond à un déplacement d'un sommet à un sommet adjacent. [1]

Le nombre de sommets d'un polytope croit de façon exponentiel dans la dimension m. Par exemple, un hypercube de dimension m a 2^m sommets. On peut effectivement construire des problème de programmation linéaire pour lesquels le nombre de sommets traversés par l'algorithme du simplexe est exponentiel. Ces problèmes sont assez artificiels et en pratique le nombre d'itérations de l'algorithme du simplexe croit plutôt de façon *linéaire* dans la dimension du problème.

Si les coefficients initiaux sont rationnels alors il est possible d'effectuer tous les calculs en restant dans l'ensemble des nombres rationnels. Il est aussi possible de montrer que la croissance des nombres rationnels calculés est modérée (polynomial dans la taille initiale).

L'algorithme du simplexe est donc un algorithme exponentiel dans le pire des cas qui est efficace en pratique. Des analyses de complexité comme l'analyse lissée (*smoothed analysis*) essayent d'expliquer ce phénomène [ST04]. L'algorithme du simplexe a été mis au point autour de 1950 par George Dantzig [Dan48]. Le premier algorithme polynomial pour la programmation linéaire a été proposé par [Kha79] en utilisant une méthode de l'ellipsoïde complètement différente de celle du simplexe. En pratique, cet algorithme n'est pas compétitif avec l'algorithme du simplexe. Un deuxième algorithme polynomial dit des *points intérieurs* a été proposé par [Kar84]. A ce jour, les mises en oeuvre de cette méthode sont compétitives avec l'algorithme du simplexe.

La plupart des systèmes disponibles pour la solution de problèmes de programmation linéaire utilisent des nombres flottants et donc des erreurs d'approximation peuvent se produire pendant le calcul. Dans ce cas, un calcul des erreurs est nécessaire pour estimer la fiabilité de la solution. Un nombre important de problèmes pratiques de programmation linéaire produisent des matrices creuses et dans ces cas des techniques spécifiques de mise en oeuvre peuvent améliorer l'efficacité de façon significative. Autour de 2015, les meilleurs systèmes non-commerciaux peuvent traiter des problèmes avec environ $10^5 - 10^6$ contraintes et autant de variables.

27.3 Condition d'optimalité et dualité

On peut montrer que si dans la forme écart (27.2), $c_j \leq 0$ pour tout $j \in N$ alors la solution admissible x associée à la forme écart est optimale. Une façon élégante de montrer cette propriété est d'utiliser la dualité. La forme écart (27.2) est équivalente à :

$$\begin{array}{ll} max\ \Sigma_{j=1,\ldots,n} c_j x_j & \\ \Sigma_{j=1,\ldots,n} a_{i,j} x_j \leq b_i & i = 1,\ldots,m \\ x_j \geq 0 & j = 1,\ldots,n\ , \end{array} \tag{27.8}$$

1. Modulo certains cas dégénérés où on reste dans le même sommet !

et par la définition 44, sa forme duale est :

$$\begin{array}{ll} min\ \Sigma_{i=1,\ldots,m} b_i y_i & \\ \Sigma_{i=1,\ldots,m} a_{j,i} y_i \geq c_j & j = 1,\ldots,n \\ y_i \geq 0 & i = 1,\ldots,m\ . \end{array} \tag{27.9}$$

On peut déterminer la solution optimale de la forme duale à partir de la forme écart terminale. Plus précisément, soient N' l'ensemble des indices dans la forme écart terminale et soient c'_j pour $j \in N'$ les coefficients (négatifs) de la fonction objectif de cette forme terminale. On rappelle que par convention l'ensemble initiale B des indices basiques est $\{n+1,\ldots,n+m\}$. Il y a donc une correspondance bijective entre les variables de la forme duale $y_1,\ldots,y_m$ et les variables basiques $x_{n+1},\ldots,x_{n+m}$. On peut montrer le fait suivant (une preuve est dans [CLRS09]).

Fait 2 *Une solution admissible et optimale pour la forme duale est :*

$$y_i = \begin{cases} -c'_{n+i} & si\ n+i \in N' \\ 0 & autrement. \end{cases} \tag{27.10}$$

Donc un coefficient c_j d'une variable x_j non-basique de la forme écart terminale ($j \in N'$) est retenu si et seulement si x_j est une variable basique de la forme écart intiale. Pour montrer l'optimalité, on utilise la proposition 30. à savoir, si $(x_1,\ldots,x_n)$ est la solution optimale pour le problème primal qui est dérivée de la forme écart terminale alors on montre que :

$$\Sigma_{j=1\ldots,n} c_j x_j = \Sigma_{i=1,\ldots,m} b_i y_i\ .$$

Exemple 64 *En continuant l'exemple 63, on peut arriver à la forme écart suivante où les coefficients de la fonction objectif sont tous négatifs.*

$$\begin{array}{l} max\ 28 - \frac{x_3}{6} - \frac{x_5}{6} - \frac{2x_6}{3} \\ x_1 = 8 + \frac{x_3}{6} + \frac{x_5}{6} - \frac{x_6}{3} \\ x_2 = 4 - \frac{8x_3}{3} - \frac{2x_5}{3} + \frac{x_6}{3} \\ x_4 = 18 - \frac{x_3}{2} + \frac{x_5}{2}\ . \end{array}$$

Une solution optimal pour le problème primal est donc $(x_1, x_2, x_3) = (8, 4, 0)$ *avec objectif égal à* 28 *et une solution optimale pour le problème dual est* $(y_1, y_2, y_3) = (0, \frac{1}{6}, \frac{2}{3})$ *avec objectif aussi égal à* 28*.*

27.4 Solution admissible initiale

Considérons un problème de la forme (26.2) :

$$max\{c^T x \mid Ax \leq b, x \geq 0\}\ , \tag{27.11}$$

Si $b \geq 0$ on peut prendre $x = 0$ comme solution admissible. Sinon, on peut générer le problème auxiliaire suivant où z est une nouvelle variable et $\mathbf{1}$ un vecteur composé de 1 :

$$max\{-z \mid (Ax - z\mathbf{1}) \leq b, x, z \geq 0\}\ . \tag{27.12}$$

Si on pose $x = 0$ et on affecte à chaque composante de z la valeur $max\{|b_i| \mid i = 1,\ldots,m\}$ on obtient une solution admissible de ce problème.

Proposition 31 *Le problème 27.11 a une solution admissible si et seulement si le problème auxiliaire 27.12 a une solution optimale avec valeur de la fonction objectif égal à* 0.

PREUVE. ($\Rightarrow$) Si x est admissible pour (27.11) alors $(x, 0)$ est admissible pour (27.12) et comme la valeur objectif est 0 il doit s'agir d'une solution optimale.
($\Leftarrow$) Si la valeur de la fonction objectif du problème auxiliaire est 0 on doit avoir $z = 0$ et donc on a une solution admissible pour le problème (27.11). □

Annexe A

Problèmes

On présente quelques problèmes un peu plus longs qui sont tirés de sujets d'examen.

A.1 Chiffrement par permutation

On suppose $2 \leq m \leq n$. Pour chiffrer un texte composé de n caractères avec une permutation sur l'ensemble $\{0, \ldots, m-1\}$ on procède de la façon suivante.

Phase de bourrage On complète le texte à chiffrer de façon à que sa longueur soit un multiple de m. Si n est un multiple de m on ajoute au texte les caractères $XY \cdots Y$ où Y est répété $m-1$ fois. Sinon, on ajoute au texte les caractères $XY \cdots Y$ où Y est répété $m-r-1$ fois et $r = n \mod m$ (notez que dans ce cas $0 < r < m$ et $m-r-1 \geq 0$). Par exemple, si $n = 4$, $m = 3$ et le texte est $ABCD$ alors on est dans le deuxième cas avec $r = 1$ et on ajoute le texte XY pour obtenir $ABCDXY$.

Phase de chiffrement Après le bourrage, la longueur du texte à chiffrer est un multiple de m. On applique la permutation au texte par blocs de m caractères pour obtenir un texte chiffré qui a autant de caractères que le texte après bourrage. En continuant l'exemple précédent, si la permutation est $1, 2, 0$ on obtient comme texte chiffré $CABYDX$.

Déchiffrement Le déchiffrement d'un texte obtenu de cette façon est calculé en appliquant la permutation inverse au texte chiffré par blocs de m caractères et ensuite en éliminant la partie terminale du texte de la forme $XY \cdots Y$. Dans notre exemple, la permutation inverse est $2, 0, 1$ et si on l'applique à $CABYDX$ par blocs de 3 on obtient $ABCDXY$ et après élimination de XY on revient au texte d'origine $ABCD$.

1. Programmez une fonction d'en tête `int lonbourrage(int n, int m)` qui calcule la longueur d'un texte composé de n caractères après bourrage relativement à une permutation sur $\{0, \ldots, m-1\}$.
2. Programmez une fonction d'en tête `void bourrage(int n, char t[n], int l, char bt[l], int m)` qui prend en argument un tableau `t[n]` qui contient le texte et un tableau `bt[l]` non-initialisé dont la longueur `l` est exactement celle prévue par la fonction `longbourrage` relativement à `m`. La fonction écrit dans le tableau `bt[l]` le texte obtenu de `t[n]` après bourrage.
3. Programmez une fonction d'en tête `void chif(int l, char bt[l], int m, int perm[m])` qui prend en argument un texte après bourrage (par rapport à `m`) représenté par le tableau `bt[l]` et une permutation représentée par le tableau `perm[m]` et écrit le chiffrement du texte dans le tableau `bt[l]`.
4. Programmez une fonction d'en tête `void invperm(int m, int perm[m])` qui calcule la permutation inverse de celle reçue en entrée dans le tableau `perm[m]` et écrit le résultat dans le tableau `perm[m]`.
5. Programmez une fonction d'en tête `int dechif(int l, char t[l], int m, int perm[m])` qui prend en argument un texte après chiffrement représenté par le tableau `t[l]` et la permutation utilisée pour le chiffrer représentée par le tableau `perm[m]` et écrit dans le tableau `t[l]` le texte après déchiffrement. La fonction `dechif` rend aussi comme résultat l'indice du dernier caractère significatif dans le tableau.

A.2 Chaînes additives

Une *chaîne additive* est une séquence $x_0, \ldots, x_k$ telle que $x_0 = 1$ et chaque élément x_i de la séquence avec $i \geq 1$ est égal à la somme de deux nombres qui le précèdent dans la séquence :

$$\forall\, i \in \{1, \ldots, k\}\ \ \exists\, j, \ell < i\ \ x_i = x_j + x_\ell$$

Par exemple, $1, 2, 4, 5, 9$ est une chaîne additive car $x_1 = x_0 + x_0$, $x_2 = x_1 + x_1$, $x_3 = x_0 + x_2$ et $x_4 = x_2 + x_3$.

1. Écrire une fonction lire qui prend en entrée un entier k et un tableau t d'entiers (avec au moins $k+1$ entiers) et qui effectue l'opération suivante : lit $k+1$ entiers de la console et les mémorise dans le tableau t aux positions $0, 1, \ldots, k$.
2. Écrire une fonction verifie_aux qui prend en entrée un entier i positif et un tableau t d'entiers (avec au moins $i+1$ entiers aux positions $0, 1, \ldots, i$) et qui rend la valeur 1 si

   $$\exists\, j, \ell\ \ (\ 0 \leq j \leq \ell < i\ \ \text{et}\ \ \mathsf{t}[i] = \mathsf{t}[j] + \mathsf{t}[\ell]\ )$$

   et 0 autrement. Estimez la complexité asymptotique de verifie_aux en fonction de i.
3. Écrire une fonction verifie qui prend en entrée un entier k (positif ou nul) et un tableau t d'entiers (avec au moins $k+1$ entiers aux positions $0, 1, \ldots, k$) et qui rend la valeur 1 si t[0], ..., t[k] est une chaîne additive et 0 autrement. Estimez la complexité asymptotique de verifie en fonction de k. Vous devez utiliser la fonction verifie_aux.
4. Une *chaîne additive pour un entier* n est une chaîne additive dont le dernier élément est n. On dénote par $|n|$ le nombre de bits nécessaires à représenter le nombre n en base 2. Montrez que tout entier $n \geq 1$ admet une chaîne additive de longueur inférieure à $2 \cdot |n|$. Calculez une chaîne additive pour $n = 25$.
5. Une chaîne additive $x_0, \ldots, x_k$ est *croissante* si on a $x_i < x_{i+1}$ pour $i = 0, \ldots, k-1$. Une chaîne additive pour n est *optimale* s'il n'existe pas une chaîne additive plus courte pour n. Montrez que tout entier $n \geq 1$ admet une chaîne additive croissante et optimale de longueur inférieure à $2 \cdot |n|$.
6. Calculez la longueur d'une chaîne additive optimale pour $n \in \{1, \ldots, 10\}$. Vous devez expliquer la méthode utilisée et répondre à la question suivante : quel est le plus petit nombre $n \geq 1$ qui a deux chaînes additives croissantes et optimales différentes ?
7. Écrire une fonction chaîne qui prend en entrée un entier $n \geq 1$ et un tableau t qui contient au moins $2 \cdot |n|$ éléments et qui mémorise aux positions $t[0], \ldots, t[k]$ ($k \leq 2 \cdot |n|$) une chaîne additive croissante pour n. En plus du code, vous devez fournir : (i) une description de l'exécution de l'algorithme pour $n = 25$, et (ii) une estimation de sa complexité asymptotique en fonction de $|n|$.
8. Soient a, n, m entiers avec $2 \leq a, n \leq m$ et soit $x_0, \ldots, x_k$ une chaîne additive pour n. Montrez qu'on peut calculer l'exposant modulaire $(a^n) \mod m$ en effectuant k multiplications modulo m.
9. La séquence de Fibonacci (en supposant $F(0) = 1$) est un cas particulier de chaîne additive. On sait que $F(15) = 987$. Combien de multiplications modulo m faut-il pour calculer $(a^{F(15)}) \mod m$ avec la méthode du carré itéré ? Peut-on faire mieux ?
10. Écrire une fonction opt qui prend en entrée un entier $n \geq 1$ et retourne la longueur d'une chaîne additive optimale pour n. En plus du code, vous devez donner la trace des appels de fonction à partir de l'appel opt(4).
11. Écrire une fonction opt_print qui prend en entrée un entier $m \geq 1$ et imprime la longueur d'une chaîne additive optimale pour les nombres compris entre 1 et m. Par exemple, si $m = 4$, la sortie aura la forme :

```
opt(1) = 1
opt(2) = 2
opt(3) = 3
opt(4) = 3
```

A.3 Affectation stable

Soient E un ensemble d'étudiants et T un ensemble de tuteurs. On suppose que ces ensembles sont finis et ont la même cardinalité $n \geq 1$. Chaque étudiant classe (strictement) les tuteurs et chaque tuteur classe

(strictement) les étudiants. On écrit $t >_e t'$ si l'étudiant e préfère strictement le tuteur t au tuteur t' et on écrit $e >_t e'$ si le tuteur t préfère strictement l'étudiant e à l'étudiant e'. Une affectation *complète* a est une fonction bijective des étudiants aux tuteurs : $a : E \to T$. Une affectation complète a est *stable* si pour tout couple $(e, t) \in E \times T$ tel que $a(e) = t'$ et $a(e') = t$ on a : (i) si $t >_e t'$ alors $e \not>_t e'$ et (ii) si $e >_t e'$ alors $t \not>_e t'$.

On représente les préférences des étudiants par une matrice pe de dimension $n \times n$ et les préférences des tuteurs par une matrice pt de dimension $n \times n$ aussi. On suppose $E = T = \{0, \ldots, n-1\}$ et on indique les préférences avec un nombre compris entre 0 et $n-1$ avec la convention que 0 est le premier choix et $n-1$ le dernier (l'ordre est donc inversé par rapport à l'ordre usuel sur les nombres naturels). Ainsi on a $pe[e][t] = r$ si et seulement si l'étudiant e place le tuteur t au rang r. Et de même $pt[t][e] = r$ si et seulement si le tuteur t place l'étudiant e au rang r. On représente une affectation complète par un tableau de n entiers différents qui varient dans T.

Par exemple, supposons $E = T = \{0, 1, 2\}$ avec les préférences suivantes (dans cet exemple les matrices des préférences coïncident !).

$$pe = pt = \begin{bmatrix} 1 & 0 & 2 \\ 2 & 1 & 0 \\ 0 & 2 & 1 \end{bmatrix}$$

L'affectation où chaque étudiant a son premier choix ($a[0] = 1, a[1] = 2, a[2] = 0$) est stable. Par ailleurs, si on prend le premier choix des tuteurs on a aussi une affectation stable. Il y a aussi une troisième affectation stable où chaque étudiant et chaque tuteur a son deuxième choix. Les autres trois affectations possibles ne sont pas stables.

1. Programmez une fonction d'en tête :

   `void imprimer(int n, int a[n])`

   qui imprime à l'écran l'affectation complète `a`.
2. Programmez une fonction d'en tête :

   `void pref2rang(int n, int pe[n][n], int er[n][n])`

   qui prend en entrée la matrice préférence des étudiants et initialise la matrice `er` de façon telle que `er[e][r] = t` si et seulement si `pe[e][t] = r`. En d'autres termes, `er[e][r]` est le tuteur qui se retrouve au rang `r` dans le classement de l'étudiant `e`.
3. Programmez une fonction d'en tête :

   `short verif_cmp(int n, int a[n])`

   qui retourne `1` si l'affectation représentée par le tableau `a` est complète, et `0` autrement.
4. Programmez une fonction d'en tête :

   `short verif_stable(int n, int pe[n][n], int pt[n][n], int a[n])`

   qui retourne `1` si l'affectation représentée par le tableau `a` est complète et stable par rapport aux matrices `pe` et `pt` et `0` autrement.
5. Programmez une fonction d'en tête

   `void gen_stable(int n, int pe[n][n], int pt[n][n])`

   qui énumère les affectations complètes jusqu'à en trouver une qui est stable et dans ce cas elle l'imprime à l'écran.

A.4 Remplissages de grilles

On considère une grille 5×5 où chaque *position* est déterminée par un nombre compris entre 0 et 24 selon le schéma suivant :

0	1	2	3	4
5	6	7	8	9
10	11	12	13	14
15	16	17	18	19
20	21	22	23	24

Par exemple, l'angle SW de la grille correspond à la position 20. A partir de chaque position, on peut aller vers N,S,W,E en sautant 2 positions ou vers NE, NW, SE, SW en sautant 1 position. Bien sûr ces déplacements sont possibles seulement si on ne sort pas de la grille. Ainsi de la position 12 (le centre de la grille) on a 4 déplacements possibles (à savoir $0, 4, 20, 24$) et dans toutes les autres positions on en a que 3 (par exemple de 5 on peut aller dans $8, 17, 20$). Une *trajectoire* à partir de la position p est une suite de déplacements qui commence à la position p, qui ne passe jamais deux fois par la même position et qui termine à une position où on ne peut plus se déplacer sans aller dans une position déjà visitée. La *longueur* d'une trajectoire est le nombre de positions visitées et clairement ce nombre ne peut pas excéder 25. On *représente une trajectoire* par un tableau d'entiers `int t[25]` avec la convention que les positions visitées sont dans l'ordre $t[0], t[1], t[2], \ldots$ et que si la trajectoire comporte i positions avec $i < 25$ alors $t[i] = -1$ (et donc les valeurs après le premier -1 ne sont pas significatives). Par exemple, le tableau de 25 entiers suivant représente une trajectoire de longueur 13 qui commence à la position 0 et termine à la position 23 :

```
t_0={0,12,20,5,8,16,1,13,10,22,14,11,23,-1,0,37,-2,3,41,5,6,77,8,9,10}
```

1. Écrire 2 fonctions `ligne` et `colonne` qui prennent en entrée une position et donnent comme résultat la ligne et la colonne qui correspondent à la position, respectivement. Par convention on compte les lignes et les colonnes à partir de 0.
2. Écrire une fonction `depl` qui prend en entrée une position p et un tableau `int d[5]` et écrit dans le tableau `d` les positions vers lesquelles on peut se déplacer à partir de la position p en ajoutant une valeur -1 à la fin. Par exemple, si $p = 5$ alors on écrira les valeurs $8, 17, 20, -1$ dans $d[0], d[1], d[2], d[3]$, respectivement.
3. On représente les positions déjà visitées par un tableau `short v[25]` tel que $v[i]$ vaut 0 si la position n'a pas été visitée et 1 autrement. écrire une fonction `depl_adm` qui prend en entrée une position p, les positions déjà visitées (représentées par un tableau de `short`) et un tableau `int d[5]` et écrit dans le tableau `d` les positions vers lesquelles on peut se déplacer à partir de la position p et qu'on a pas déjà visité en ajoutant une valeur -1 à la fin. Par exemple, si $p = 5$, $v[8] = 1$, $v[17] = 0$ et $v[20] = 1$ alors on écrira les valeurs $17, -1$ dans $d[0], d[1]$, respectivement.
4. Écrire une fonction `verifie` qui prend en entrée un tableau d'entiers de 25 éléments et qui rend 1 si le tableau représente une trajectoire et 0 autrement.
5. Pouvez-vous estimer le nombre de trajectoires possibles à partir d'une position initiale donnée ? Pensez-vous que ce nombre est bien plus petit que $25! = 25 \cdot 24 \cdots 2$?
6. Écrire une fonction `imprime` qui prend en entrée une trajectoire et l'imprime comme une grille 5×5. Par exemple, la trajectoire t_0 ci-dessus doit être imprimée de la façon suivante :

1	7			
4			5	
9	12	2	8	11
	6			
3		10	13	

7. Écrire un fonction `gen` qui prend en entrée une position initiale et génère une *trajectoire aléatoire* à partir de cette position en utilisant la stratégie suivante : elle calcule les déplacements possibles à partir de la dernière position et il en sélectionne un avec probabilité uniforme. Vous ferez l'hypothèse qu'un appel à `rand()%m` vous donne un entier dans $0, \ldots, m-1$ avec une probabilité uniforme.
8. Écrire un fonction `echantillon` qui prend en entrée une position initiale et un entier n et qui calcule n trajectoires en utilisant la stratégie aléatoire décrite au point précédent. A la fin du calcul, la fonction imprime la *longueur moyenne* des n trajectoires ainsi que *une plus longue* et *une plus courte* trajectoire parmi celles calculées.
9. Écrire une fonction `max` qui prend en entrée une position initiale, énumère les trajectoires valides à partir de cette position et imprime une *trajectoire de longueur maximale*. En particulier, si votre fonction trouve une trajectoire de longueur 25 elle doit l'imprimer et terminer.
10. Écrire une fonction `C min` qui prend en entrée une position initiale, énumère les trajectoires valides à partir de cette position et imprime une *trajectoire de longueur minimale*. Votre fonction devrait écarter rapidement les trajectoires qui sont au moins aussi longues que celles déjà trouvées.

A.5 Tournoi à élimination directe

La *configuration initiale* d'un tournoi à élimination directe est décrite par un tableau t de type int t[n] où $n = 2^k$, $k \geq 1$ et chaque cellule contient le nom d'un joueur (dans notre cas un entier). Un tel tournoi se joue en k tours et au tour i, pour $i = 1, \ldots, k$, on joue 2^{k-i} parties. Vous disposez d'une fonction play d'en-tête short play(int x, int y) qui renvoie 0 si x gagne et 1 si y gagne. On écrit $t[i] \leftrightarrow t[j]$ si $t[i]$ joue contre $t[j]$ avec $i < j$ et dans ce cas on suppose que le gagnant est mémorisé dans $t[i]$. Ainsi la structure des parties d'un tournoi est la suivante :

$$
\begin{array}{ll}
t[0] \leftrightarrow t[1], t[2] \leftrightarrow t[3], t[4] \leftrightarrow t[5], \ldots, t[n-2] \leftrightarrow t[n-1] & \text{(tour 1)} \\
t[0] \leftrightarrow t[2], t[4] \leftrightarrow t[6], \ldots, t[n-4] \leftrightarrow t[n-2] & \text{(tour 2)} \\
t[0] \leftrightarrow t[4], \ldots, t[n-8] \leftrightarrow t[n-4] & \text{(tour 3)} \\
\ldots & \\
t[0] \leftrightarrow t[n/2] & \text{(tour } k\text{)}
\end{array}
$$

1. Programmez en C une fonction tournoi d'en-tête void tournoi(int k, int n, int t[n]) qui simule le tournoi en se basant sur les hypothèses décrites ci-dessus. À la fin du calcul $t[0]$ est donc le nom du gagnant du tournoi.
2. En supposant que le coût d'un appel à la fonction play est $O(1)$ en temps, déterminez la complexité asymptotique en temps de la fonction tournoi.
3. On suppose maintenant que les identités des joueurs correspondent aux entiers $\{0, \ldots, n-1\}$ et qu'on dispose d'un tableau score de type float score[n] qui associe à chaque joueur son score. Programmez en C une fonction ranking d'en-tête void ranking(int n, float score[n], int position[n]). La fonction reçoit (i) le nombre de joueurs n, (ii) le tableau avec leur score score et (iii) un tableau vide position, et écrit dans ce dernier les noms des joueurs d'après leur score. Ainsi à la fin du calcul le tableau position représente une permutation sur $\{0, 1, \ldots, n-1\}$ telle que :

   $$\text{score[position[0]]} \geq \text{score[position[1]]} \geq \cdots \geq \text{score[position[n} - 1\text{]]} \ .$$

   Par exemple, pour $n = 4$ voici un tableau score possible et le contenu du tableau position à la fin du calcul :

	0	1	2	3
score	5, 4	2, 7	3, 1	7, 8
position	3	0	2	1

   Vous pouvez allouer des tableaux auxiliaires d'un type approprié et utiliser une fonction auxiliaire de tri sur ces tableaux sans la programmer.
4. On suppose maintenant que le tableau position de type int position[n] contient les noms des joueurs ordonnés d'après leur score. Programmez une fonction affect d'en-tête void affect(int k, int n, int position[n], int t[n]) qui affecte les joueurs au tableau t qui représente la configuration initiale du tournoi en respectant la règle suivante pour $i = 1, \ldots, k$:

   Règle : les premiers 2^i joueurs ne peuvent pas se rencontrer avant le tour $k - i + 1$,

   ce qui revient à dire que les premiers 2 joueurs ne peuvent pas se rencontrer avant la finale (tour k), les premiers 4 avant les demi-finales (tour $k-1$) et ainsi de suite. Par exemple, en supposant $position[i] = i$ pour $i = 0, 1, \ldots, 15$ (ce qui revient à dire que 0 est le nom du joueur avec le meilleur score et 15 le nom du joueur avec le pire score) on peut avoir l'affectation suivante pour le tableau t qui représente la configuration initiale du tournoi :

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$t[i]$	0	8	4	9	2	10	5	11	1	12	6	13	3	14	7	15

5. L'affectation de la question 4 est déterminée par le score des joueurs, ce qui est ennuyeux, car tant que le score des joueurs ne change pas on joue toujours le même tournoi. On souhaite introduire une composante aléatoire dans cette affectation tout en respectant la règle de la question 4. Vous disposez maintenant d'une fonction perm d'en-tête void perm(int i, int j, int r[]). Si $i \leq j$ et i, j sont dans le domaine de définition du tableau r alors la fonction perm permute les éléments r[i], r[i + 1], ..., r[j] avec

une probabilité uniforme. Programmez une variante de la fonction affect, disons affect_alea, qui a le même en-tête, respecte toujours la règle de la question 4, mais utilise la fonction perm pour introduire une composante aléatoire dans la génération du tableau initial. Par rapport à l'exemple de la question 4, la fonction affect_alea doit pouvoir générer aussi le tableau :

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$t[i]$	0	10	5	9	3	8	7	11	1	15	4	13	2	14	6	12

A.6 Motifs et empreintes

Un *texte* est une suite de $n \geq 1$ caractères représentés par un tableau de valeurs de type char. Un *motif* est aussi un suite de $m \leq n$ caractères représentés par un tableau de valeurs de type char. Une *occurrence* d'un motif dans un texte est un nombre naturel qui indique une position dans le texte à partir de laquelle les caractères du texte coïncident avec ceux du motif. Par exemple, les occurrences du motif *bra* dans le texte *abracadabra* sont exactement 1 et 8. On notera que ici $m = 3$, $n = 11$ et qu'on compte les positions de gauche à droite à partir de 0. Les occurrences d'un motif peuvent se 'superposer'. Par exemple, les occurrences du motif *bb* dans *abbba* sont 1 et 2.

1. Programmez une fonction d'en tête
   `short check(int m, char motif[m], int n, char texte[n], int pos)`
   qui prend en argument un motif, un texte et une position pos et qui rend 1 si pos est une occurrence du motif dans le texte et 0 autrement.
2. Programmez une fonction d'en tête
   `void occurrences(int m, char motif[m], int n, char texte[n])`
   qui prend en argument un motif et un texte et imprime sur la sortie standard toutes les occurrences du motif dans le texte.
3. Analysez la complexité asymptotique de votre programme en fonction de m et n.

L'entier $M =$ 2147483647 est l'entier le plus grand que l'on peut représenter en C avec le type int (sur 32 bits). Le plus grand entier p tel que $p^2 \leq M$ est 46340. Par ailleurs on pose $B = 256$.

4. Programmez les fonctions suivantes :
   4.1 Une fonction injective d'en tête int ci(char c) qui prend en argument une valeur de type char et rend comme résultat un entier dans l'intervalle $[0, 255]$.
   4.2 Une fonction d'en tête int mod(int x, int p) qui prend en argument un entier x et un entier positif p et rend comme résultat $x \mod p$, à savoir l'unique entier r dans l'intervalle $[0, p-1]$ tel que pour un q nombre entier, $x = q \cdot p + r$. NB Dans le langage C, la fonction % sur les entiers peut rendre comme résultat un entier négatif.
   4.3 Une fonction d'en tête int puis(int m, int p) qui prend en argument les entiers m et p et rend comme résultat $B^{m-1} \mod p$.

Soit $w \equiv x_0 \cdots x_{m-1}$ une suite de m caractères. Si x est un caractère soit $ci(x)$ l'entier qui lui est associé dans l'intervalle $[0, 255]$ (voir question 4.1). L'empreinte $e(w)$ de la suite est un entier modulo p défini par :

$$e(w) = (\Sigma_{i=1,\ldots,m} B^{m-i} \cdot ci(x_{i-1})) \mod p \,. \tag{A.1}$$

Par exemple, si $m = 3$ et $w = x_0x_1x_2$ on a :

$$e(w) = (B^2 \cdot ci(x_0) + B \cdot ci(x_1) + ci(x_2)) \mod p \,.$$

5. Est-ce possible d'avoir deux suites de longueur m qui ont la même empreinte ?
6. Programmez une fonction d'en tête : int emp(int m, int k, char t[], int p) qui prend en argument un tableau de char défini aux positions $t[k], \ldots, t[k+m-1]$ ainsi que le module p et rend comme résultat l'empreinte de la suite $t[k], \ldots, t[k+m-1]$. Le calcul doit être organisé de façon à éviter tout débordement.
7. Soit e l'empreinte de la suite $t[k], \ldots, t[k+m-1]$ et soit $b = B^{(m-1)} \mod p$. On suppose que les opérations de multiplication, addition et calcul de l'opposé modulo p prennent un temps constant $O(1)$. Supposons que l'on souhaite calculer l'empreinte e' de la suite $t[k+1], \ldots, t[k+m]$. Expliquez comment effectuer ce calcul en $O(1)$ à partir des entiers e, b, p, $t[k]$, $t[k+m]$.
8. Programmez une fonction d'en tête :
   `void empreintes(int n, char t[n], int m)`
   qui prend en entrée un texte de n char et un entier $m \leq n$ et imprime sur la sortie standard les empreintes des suites $t[i] \cdots t[i+m-1]$ pour $i = 0, \ldots, n-m$. Votre fonction doit optimiser le temps de calcul en utilisant la méthode évoquée à la question 7.

9. Programmez une fonction d'en tête
   `void occurrences_emp(int m, char motif[m], int n, char texte[n])`
   qui prend en argument un motif et un texte et imprime sur la sortie standard toutes les occurrences du motif dans le texte. Votre fonction doit utiliser la notion d'empreinte pour optimiser le temps de calcul. En particulier dans la situation où l'empreinte du motif est différente de toutes les empreintes des suites $t[i]\cdots t[i+m-1]$ pour $i = 0, \ldots, n-m$, la complexité de la fonction doit être $O(n)$.
10. Supposons maintenant que le texte contient $n-m$ occurrences du motif et que $m = n/2$. Quelle est complexité asymptotique (en fonction de n) de la fonction occurrence_emp dans ce cas ?

A.7 Majorité

On cherche à déterminer si parmi les n entiers d'un tableau il y en a un qui paraît $k > n/2$ fois. On appelle un tel élément *majoritaire*. On commence avec un algorithme *probabiliste*.

1. Programmez une fonction check d'en tête :

   $$\mathsf{short\ check(int\ n,\ int\ t[n],\ int\ m)}$$

   qui retourne 1 si m paraît plus que $n/2$ fois dans le tableau t et 0 autrement.
2. On suppose que le tableau t contient un élément majoritaire m. Estimez la probabilité qu'avec ℓ tirages dans le tableau t (indépendants et avec probabilité uniforme) on ne tire jamais m.
3. On suppose la déclaration de type : struct result{short maj; int m}. Programmez une fonction (probabiliste !) pmajority d'en tête :

   $$\mathsf{struct\ result\ pmajority(int\ n,\ int\ t[n])}$$

   de complexité en temps $O(n)$ telle que la fonction rend la structure $\{1, m\}$ si le tableau t contient un élément majoritaire m et la structure $\{0, -1\}$ sinon. Dans ce dernier cas, le tableau peut quand même contenir un élément majoritaire avec une probabilité inférieure à $(1/2^{30})$.

On cherche maintenant à concevoir un algorithme *déterministe* pour le même problème. Soit $t_1, \ldots, t_n$ une séquence de n entiers. On définit les séquences $c_0, c_1, \ldots, c_n$ et $v_1, \ldots, v_n$ par $c_0 = 0$ et

$$(c_{i+1}, v_{i+1}) = \begin{cases} (1, t_{i+1}) & \text{si } c_i = 0 \\ (c_i + 1, v_i) & \text{si } c_i > 0, t_{i+1} = v_i \\ (c_i - 1, v_i) & \text{si } c_i > 0, t_{i+1} \neq v_i \end{cases}$$

4. Montrez que si m est majoritaire dans $t_1, \ldots, t_n$ et $c_i > 0$ pour $i = 1, \ldots, n-1$ alors $t_1 = v_1 = \cdots = v_n = m$ et $c_n > 0$.
5. Montrez que si m est majoritaire dans $t_1, \ldots, t_n$ alors $c_n > 0$ et $v_n = m$.
6. Programmez une fonction dmajority d'en tête :

   $$\mathsf{struct\ result\ dmajority(int\ n,\ int\ t[n])}$$

   de complexité en temps $O(n)$ telle que la fonction rend la structure $\{1, m\}$ si le tableau t contient un élément majoritaire m et la structure $\{0, -1\}$ sinon.

A.8 Un tas en dimension 2

Soit T un tableau $m \times n$ qui contient des entiers ou un symbole spécial ∞ qui est plus grand que n'importe quel entier. On dit que le tableau est *bien formé* si chaque ligne lue de gauche à droite et chaque colonne lue du haut vers le bas donne une suite croissante (mais pas forcement strictement croissante). Un tableau bien formé peut donc contenir r entiers pour $0 \leq r \leq mn$. Voici un exemple de tableau bien formé 4×4 qui contient 8 entiers :

2	3	5	14
4	8	16	∞
12	∞	∞	∞
∞	∞	∞	∞

1. Proposez des conditions pour vérifier en $O(1)$ si un tableau bien formé est : (i) *vide*, (ii) *plein*.

2. Proposez un algorithme en $O(m+n)$ pour *extraire un élément minimum* d'un tableau bien formé non-vide. Vous devez illustrer votre algorithme en utilisant le tableau ci-dessus et expliquer pourquoi votre algorithme est bien en $O(m+n)$.
3. *Programmez* la fonction C qui correspond à l'algorithme d'extraction. Vous ferez l'hypothèse que le symbole ∞ est représenté par la constante `INT_MAX` et que le tableau contient des entiers strictement plus petits que `INT_MAX`.
4. Proposez un algorithme en $O(m+n)$ pour *insérer un entier* dans un tableau bien formé non-plein.
5. *Programmez* la fonction C qui correspond à l'algorithme d'insertion (mêmes hypothèses que pour l'extraction).
6. Supposons maintenant $n = m$. Proposez un algorithme pour *trier* n^2 entiers en $O(n^3)$ qui utilise les fonctions d'extraction et d'insertion aux points 3. et 4. (bien sûr, une solution qui fait appel à un des algorithmes de tri étudiés dans le cours n'est pas valide !). Comparez la complexité asymptotique dans le pire de cas de votre algorithme à celles du tri par insertion et du tri par fusion.

A.9 Recherche des deux points les plus rapprochés

On s'intéresse au problème suivant : on reçoit en entrée un tableau p qui contient n points distincts dans $\mathbf{R}^2$ et on souhaite calculer la distance euclidienne minimale entre deux points. On suppose : (i) que les points sont des valeurs de type `struct point {double x; double y;}`, (ii) qu'on peut ignorer les erreurs d'approximation dûs au calcul sur les flottants des opérations arithmétiques et de l'opération d'extraction de la racine carrée et (iii) que les dites opérations sont effectuées en temps constant.

1. Programmez une fonction d'en tête :
   `double dp(int n, struct point p[n])`
   qui prend en argument un entier $n \geq 2$ et un tableau de n points distincts et retourne comme résultat la distance euclidienne minimale entre deux points distincts. Votre fonction devrait avoir une complexité asymptotique en temps en $O(n^2)$.
2. Dans la suite, on suppose disposer d'une fonction d'en tête :
   `void trifusion(int n,struct point t[n],short coord)`
   qui prend en argument un tableau t avec n points et le trie par ordre croissant par rapport à la première composante (si coord est 1) ou la deuxième composante (si coord est 2). On développe maintenant une approche *diviser pour régner*. Supposons que xord est un tableau de points et $i < j$ deux nombres naturels tels que $\mathsf{xord}[i].x \leq \cdots \leq \mathsf{xord}[j].x$ (première composante croissante). On dénote par $dp(i,j)$ la distance minimale entre deux points dans l'ensemble $P_{i,j} = \{\mathsf{xord}[i], \ldots, \mathsf{xord}[j]\}$. Si $j - i$ est petit on peut appliquer l'algorithme de la question 1. Sinon, soient $m = (i+j)/2$, $xm = \mathsf{xord}[m].x$ et

   $$d = min\{dp(i,m), dp(m+1,j)\} \ .$$

   La valeur d est donc une *borne supérieure* à $dp(i,j)$. Pour calculer $dp(i,j)$ il reste à déterminer la distance minimale entre un point dans $\{\mathsf{xord}[i], \ldots, \mathsf{xord}[m]\}$ et un point dans $\{\mathsf{xord}[m+1], \ldots, \mathsf{xord}[j]\}$. Soit $B(xm,d,i,j)$ l'ensemble des points dans $P_{i,j}$ tels que la distance de leur abscisse de xm est au plus d.[1] Programmez une fonction points_bande d'en tête :
   `struct bande {int low; int high;};`
   `struct bande points_bande(int i, int j, struct point xord[], double d)`
   qui calcule l'ensemble $B(xm,d,i,j)$. Plus précisement, points_bande retourne une valeur b de type struct bande telle que $B(xm,d,i,j) = \{\mathsf{xord}[b.low], \ldots, \mathsf{xord}[b.high]\}$. Analysez la complexité asymptotique de la fonction points_bande.
3. Soit p un point dans $B(xm,d,i,j)$. Bornez le nombre de points qu'on peut trouver dans $B(xm,d,i,j)$ dont l'ordonnée est à une distance au plus d de l'ordonnée de p. Question auxiliaire/suggestion : combien de points à une distance au moins d peut-on mettre dans un carré dont le côté mesure d ?
4. Programmez une fonction d'en tête :
   `double dpbande(struct bande b, struct point xord[], double d)`
   qui calcule la distance minimale entre deux points distincts (s'ils existent) dans $B(xm,d,i,j)$. Analysez la complexité asymptotique de la fonction dpbande.

1. Les points dans $B(xm,d,i,j)$ sont donc dans une bande de largeur $2 \cdot d$ centrée autour de la droite composée des points dont l'abscisse est xm.

5. Soit C une fonction sur les nombres naturels qui satisfait la récurrence :

$$C(0) = 1, \quad C(n) = 2 \cdot C(n/2) + n \cdot log_2(n) \quad (\text{si } n \geq 1).$$

   Trouvez :
   — Un nombre naturel minimal k tel que $f(n) = n^k$ et $C(n)$ est $O(f)$.
   — Un nombre naturel minimal k tel que $g(n) = n \cdot (log_2(n))^k$ et $C(n)$ est $O(g)$.
6. D'après ce que vous avez appris, pensez-vous qu'une approche *diviser pour régner* permet d'améliorer la complexité $O(n^2)$ de la question 1 ?

A.10 Arbres binaires de recherche

On se place dans le cadre des arbres binaires de recherche (ABR) utilisés pour représenter des ensembles ordonnés finis. On suppose le type struct node et la fonction allocat_node suivants :

```
struct node {int val; struct node * left; struct node * right;};
struct node *allocate_node(int v){
   struct node *p=(struct node *)(malloc(sizeof(struct node)));
   (p->val)=v; (p->left)=NULL; (p->right)=NULL;  return p;}
```

1. Écrire une fonction tree2tab d'en tête :

$$\text{int tree2tab(struct node} * \text{ tree, int n, int tab[n])}$$

   La fonction prend en entrée un pointeur tree à un ABR et un tableau tab (non-initialisé) de taille n. Ensuite la fonction écrit les entiers dans l'ABR dans le tableau tab par *ordre croissant* et rend comme résultat le nombre d'entiers dans l'ABR. Vous pouvez faire l'hypothèse que le nombre d'entiers dans l'ABR est au plus n.
2. Par convention, soit -1 la hauteur d'un ABR vide. *Définition ABR équilibré* : un ABR vide est équilibré et un ABR non-vide est équilibré si les sous-arbres gauche et droit de la racine sont équilibrés et ont une hauteur qui diffère au plus de 1.
   Écrire une fonction tab2tree d'en tête :

$$\text{struct node} * \text{ tab2tree(int i, int j, int tab[])}$$

   La fonction prend en entrée un tableau tab et deux indices i et j tels que : (i) i $\leq$ j, (ii) tab[i], ..., tab[j] sont définis et (iii) tab[i] $< \cdots <$ tab[j] Ensuite la fonction construit un ABR *équilibré* qui contient les entiers tab[i],...,tab[j] et rend un pointeur à la racine de l'ABR. Vous devez expliquer pourquoi l'ABR calculé est équilibré.
3. Bonus Proposez un algorithme en $O(n)$ (en temps) pour calculer un ABR équilibré qui résulte de la fusion de deux ABR (pas forcement équilibrés) de taille n.

A.11 Calcul du centre d'un arbre

Un *arbre* est un graphe, non-dirigé, acyclique et connecté. Soit T un arbre avec $N = \{0, \ldots, n-1\}$ comme ensemble des noeuds. On suppose $n \geq 2$. La *distance* $d(i, j)$ entre deux noeuds $i, j \in N$ est le nombre d'arêtes qu'il faut traverser pour aller de i à j (rappel : dans un arbre le chemin existe et est unique). Si i est un noeud, son *dégré* $deg(i)$ est le nombre de noeuds qui lui sont adjacents. L'*excentricité* d'un noeud $i \in N$ dans T est :

$$ex_T(i) = max\{d(i, j) \mid j \in N\} \ .$$

Le *centre* d'un arbre T est l'ensemble de noeuds :

$$C_T = \{i \in N \mid ex_T(i) \text{ est minimale}\} \ .$$

Les *feuilles* d'un arbre T sont les éléments de l'ensemble :

$$L_T = \{i \in N \mid deg(i) \leq 1\} \ .$$

1. Montrez les propriétés suivantes :

A Si tous les noeuds de l'arbre T sont des feuilles alors tous les noeuds sont dans le centre C_T.

B Sinon, soit T' l'arbre obtenu en éliminant de T tous les noeuds qui sont des feuilles (et les arêtes relatives). Alors le centre de l'arbre T coïncide avec le centre de l'arbre T'.

2. Dérivez des propriétés A et B un algorithme qui prend en entrée un arbre T représenté par une table de listes d'adjacence et retourne comme résultat une liste qui contient (exactement une fois) les noeuds dans le centre de l'arbre. Illustrez le calcul de votre algorithme sur un arbre avec une petite dizaine de noeuds.
3. Programmez l'algorithme comme une fonction center d'en tête :

$$\mathsf{struct\ node\ *\ center(int\ n, struct\ node\ *\ tadj[n])}$$

La fonction center prend en entrée une table de listes d'adjacence qui représente l'arbre et retourne le pointeur à la liste des noeuds qui se trouvent dans le centre de l'arbre. On admet les définitions suivantes.

```
struct node {int val; struct node * next;};
struct node *allocate_node(int v){
    struct node *p=(struct node *)(malloc(sizeof(struct node)));
    (p->val)=v; (p->next)=NULL; return p;};
struct node * insert(int i, struct node * list){
    struct node * q = allocate_node(i); (q->next)=list;   return q;};
```

4. Analysez la complexité asymptotique de votre mise-en-oeuvre en fonction de n (le nombre de noeuds).

A.12 Optimisation de requêtes

Une requête est un intervalle $[a, b]$ où a, b sont des entiers et $a \leq b$. Deux requêtes $[a_1, b_1]$ et $[a_2, b_2]$ sont en *conflit* si $[a_1, b_1] \cap [a_2, b_2] \neq \emptyset$. On dit qu'un ensemble R de requêtes est *cohérent* s'il ne contient pas deux requêtes en conflit.

1. Programmez une fonction C d'en tête short coh(int a1, int b1, int a2, int b2) qui renvoie 1 si $[a_1, b_1] \cap [a_2, b_2] = \emptyset$ et 0 autrement.

On cherche maintenant à concevoir un algorithme qui reçoit en entrée n requêtes $R = \{[a_i, b_i] \mid i \in \{1, \ldots, n\}\}$ et calcule un sous-ensemble $R' \subseteq R$ *cohérent* et de *cardinalité maximale*. Dans ce cas, on dit que R' est une solution optimale.

2. On considère la stratégie suivante : on ordonne de façon croissante les requêtes d'après leur *taille* :

$$[a_1, b_1], \ldots, [a_n, b_n] \text{ avec } (b_1 - a_1) \leq \cdots \leq (b_n - a_n)$$

et on pose :

$$\begin{array}{lll} R'_0 & = \emptyset & \\ R'_{i+1} & = \begin{cases} R'_i \cup \{[a_{i+1}, b_{i+1}]\} & \text{si } R'_i \cup \{[a_{i+1}, b_{i+1}]\} \text{ est cohérent} \\ R'_i & \text{autrement} \end{cases} \\ R' & = R'_n & \end{array}$$

Montrez que R' n'est pas toujours une solution optimale.

3. On considère la stratégie suivante : on ordonne de façon croissante les requêtes d'après leur *deuxième composante* :

$$[a_1, b_1], \ldots, [a_n, b_n] \text{ avec } b_1 \leq \cdots \leq b_n$$

et on pose :

$$\begin{array}{lll} R'_0 & = \emptyset & \\ R'_{i+1} & = \begin{cases} R'_i \cup \{[a_{i+1}, b_{i+1}]\} & \text{si } R'_i \cup \{[a_{i+1}, b_{i+1}]\} \text{ est cohérent} \\ R'_i & \text{autrement} \end{cases} \\ R' & = R'_n & \end{array}$$

3.1 Montrez qu'il y a toujours une solution optimale qui contient $[a_1, b_1]$.

3.2 Montrez que R' est toujours une solution optimale.

3.3. Estimez la complexité asymptotique en temps d'une fonction qui calcule R' à partir d'un tableau de requêtes ordonnées d'après leur deuxième composante.

On *généralise* le problème en supposant qu'une requête est maintenant un couple $([a,b],w)$ où w est le *poids de la requête* ($w > 0$) et que l'objectif est de rendre un sous-ensemble R' des requêtes R qui est *cohérent* et qui *maximise* la somme des poids des requêtes qui le composent.

4. Montrez que dans ce cas les stratégies proposées aux points 2. et 3. ne donnent pas toujours une solution optimale.
5. Programmez une fonction C d'en tête `int sup(int n, int t[n], int x)` qui prend en entrée un tableau t de n entiers ordonnés de façon croissante et un entier x et rend le plus grand indice j tel que $t[j] < x$ et -1 si un tel indice n'existe pas. Est-il possible de résoudre ce problème en temps $O(log(n))$? Expliquez.
6. On suppose avoir ordonné les requêtes par :

$$([a_1,b_1],w_1),\ldots,([a_n,b_n],w_n) \qquad b_1 < \cdots < b_n$$

On définit pour $j = 1,\ldots,n$:

$$pred(j) = max\{i \mid 1 \leq i < j, b_i < a_j\}$$

où par convention $max(\emptyset) = 0$. Proposez un algorithme pour calculer $pred(j)$ pour $j = 1,\ldots,n$. Est-il possible d'effectuer ce calcul en temps $O(nlog(n))$? Expliquez.

7. Soit $R_j = \{([a_i,b_i],w_i) \mid 1 \leq i \leq j\}$ pour $j = 1,\ldots,n$. Soit O_j le poids maximal d'une solution du problème R_j où par convention on pose : $O_0 = 0$. Montrez :

$$O_{j+1} = max(w_{j+1} + O_{pred(j+1)}, O_j)$$

8. Proposez un algorithme en temps $O(nlog(n))$ pour résoudre le problème généralisé (avec poids).

A.13 Plus longue sous-séquence croissante

Soit $x_0,\ldots,x_{n-1}$ une séquence de n entiers. Les *sous-séquences croissantes* ont la forme $x_{i_1},\ldots,x_{i_k}$ où :

$$0 \leq i_1 < \cdots < i_k \leq (n-1) \text{ et } \; x_{i_1} \leq \cdots \leq x_{i_k} \; .$$

On cherche à programmer un algorithme qui prend en entrée une séquence représentée par un tableau d'entiers et qui imprime à l'écran une plus longue sous-séquence croissante.

1. Soit $ls(i)$ pour $i = 0,\ldots,n-1$ la longueur de la plus longue sous-séquence croissante qui termine avec x_i. Clairement $ls(0) = 1$. Montrez qu'on peut calculer $ls(i+1)$ en fonction de $ls(0),\ldots,ls(i)$ et estimez en fonction de n la complexité asymptotique du temps de calcul nécessaire au calcul de $ls(i)$ pour $i = 1,\ldots,n-1$.
2. Programmez une fonction `lsf` d'en tête : `void lsf(int n, int x[n], int ls[n])`, qui prend en entrée une séquence de longueur `n` représentée par le tableau `x` et écrit dans le tableau `ls` les valeurs $ls(0),\ldots,ls(n-1)$.
3. Programmez une fonction `plsc` d'en tête : `void plsc(int n, int x[n])`, qui prend en entrée une séquence de longueur `n` représentée par le tableau `x` et imprime sur la sortie standard (écran) une plus longue sous-séquence.

A.14 Distance d'édition

Soit Σ un ensemble fini avec éléments $a,b,\ldots$ qu'on appelle *caractères*. Un mot α est une suite finie de caractères On dénote par ϵ la suite vide et par $|\alpha|$ le nombre de caractères qui composent la suite α. Par convention, le premier caractère de la suite est en position 1, le deuxième en position 2, ... On considère les *opérations* suivantes sur un mot α :

`rem(i)` si $1 \leq i \leq |\alpha|$, on efface le i-ème caractère avec un coût 2.

`ins(i,a)` si $1 \leq i \leq |\alpha|+1$ on déplace les caractères des positions $i,\ldots,|\alpha|$ aux positions $i+1,\ldots,|\alpha|+1$ et on insère le caractère `a` à la position i avec un coût 2.

`rpl(i,a)` Si $1 \leq i \leq |\alpha|$ on remplace le caractère en position i par le caractère `a` avec un coût 3.

Si o est une opération on dénote par $p(o)$ la position sur laquelle l'opération opère et par $C(o)$ son coût. Par exemple, $p(\mathsf{rpl}(5,a)) = 5$ et $C(\mathsf{rpl}(5,a)) = 3$. Notez que si la position n'est pas dans les bornes indiquées l'opération n'a pas d'effet sur le mot et son coût est 0. Soient α et β deux mots. On définit :

$d(\alpha, \beta)$ comme le coût minimal d'une suite d'opérations qui permet de transformer le mot α en le mot β.

1. Montrer que d est bien une *distance*. En particulier, pour tout α, β, γ mots : (i) $d(\alpha, \beta)$ est un nombre naturel, (ii) $d(\alpha, \beta) = d(\beta, \alpha)$, (iii) $d(\alpha, \beta) = 0$ ssi $\alpha = \beta$ et (iv) $d(\alpha, \beta) \leq d(\alpha, \gamma) + d(\gamma, \beta)$.
2. Soit $\sigma = o_1, \ldots, o_n$ une suite d'opérations et soit $\sigma_i = o_1, \ldots, o_i$ pour $i = 0, \ldots, n$. Soit α_i le mot obtenu en appliquant la séquence σ_i au mot α. On définit $C(\sigma_i) = \Sigma_{j=1,\ldots,i} C(o_j)$. Montrez que si $C(\sigma_{i+1}) = d(\alpha, \alpha_{i+1})$ alors $C(\sigma_i) = d(\alpha, \alpha_i)$.
3. On dit qu'une suite d'opérations $o_1, \ldots, o_m$ est *standard* si $p(o_1) \leq \cdots \leq p(o_m)$. Montrez que pour tout mot α et pour toute suite d'opérations $o_1, \ldots, o_m$ qui aboutit au mot β avec un coût c on peut trouver une suite d'opérations standard $o'_1, \ldots, o'_n$ qui aboutit aussi au mot β avec $n \leq m$ et avec un coût $c' \leq c$ (il suffit donc d'éditer de gauche à droite).
4. On suppose les propriétés suivantes avec $a \neq b$:

$$\begin{array}{ll} d(\epsilon, \alpha) & = 2|\alpha| \\ d(\alpha a, \beta a) & = d(\alpha, \beta) \\ d(\alpha a, \beta b) & = \min\{2 + d(\alpha, \beta b), 2 + d(\alpha a, \beta), 3 + d(\alpha, \beta)\} \end{array}$$

On suppose aussi avoir mémorisé les mots α et β dans deux tableaux de `char`, `a` et `b` de taille `m` et `n` respectivement. Écrire une fonction `C distance` qui calcule $d(\alpha, \beta)$. Votre programme doit être assez efficace pour traiter des mots avec au moins 10^3 caractères.

A.15 Clôture transitive

Soit $G = (N, A)$ un graphe dirigé avec $N = \{1, \ldots, n\}$ et $A \subseteq N \times N$. On suppose que l'ensemble des arêtes est représenté par une matrice d'adjacence, qu'on dénote aussi par A, de dimension $n \times n$ et à valeurs dans $\{0, 1\}$. Le problème qu'on considère dans la suite est le calcul de la matrice A^+ qui représente la *clôture transitive* de A. On utilisera aussi A^* pour (la matrice qui représente) la *clôture réflexive et transitive*.

On dénote par $A[i, j]$ l'élément de la matrice A à la ligne i et à la colonne j. Par ailleurs, on étend les opérations de disjonction logique '$+$' et conjonction logique '$\cdot$' aux matrices comme suit :

$$\begin{array}{ll} (B + C)[i, j] & = B[i, j] + C[i, j] \\ (B \cdot C)[i, j] & = \Sigma_{k=1,\ldots,n} B[i, k] \cdot C[k, j] \end{array}$$

1. On définit la séquence :

$$A_1 = A \qquad A_{k+1} = A_k \cdot A$$

Montrez que $A_k[i, j] = 1$ si et seulement si il y a un chemin de i à j dont la longueur est exactement k.

2. Dérivez un algorithme en $O(n^d)$ pour calculer A^+ et donnez votre estimation de d.
3. On définit maintenant une nouvelle séquence où I est la matrice identité :

$$A_0 = A + I \qquad A_{k+1} = A_k \cdot A_k$$

Montrez que $A_k[i, j] = 1$ si et seulement si il y a un chemin de i à j dont la longueur est au plus 2^k.

4. Dérivez un algorithme en $O(n^d \mathsf{log}(n))$ pour calculer A^* et donnez votre estimation de d.
5. Proposez un algorithme en $O(n^d)$ pour calculer A^+ à partir de A et A^* et donnez votre estimation de d.
6. On écrit $G(i, j, 0)$ si $(i, j) \in A$. Pour $k \in N$, on écrit $G(i, j, k)$ si (i) $(i, j) \in A$ ou (ii) $(i, i_1), \ldots, (i_\ell, j) \in A$ avec $\{i_1, \ldots, i_\ell\} \subseteq \{1, \ldots, k\}$ (en d'autres termes, il y a un chemin de i à j qui passe par $\{1, \ldots, k\}$). Montrez la *propriété suivante* :

$$G(i, j, k + 1) \quad \text{ssi} \quad G(i, j, k) \text{ ou } (G(i, k + 1, k) \text{ et } G(k + 1, j, k))$$

7. Dérivez un algorithme $O(n^d)$ pour calculer A^+ et donnez votre estimation de d.
8. Montrez que :

$$G(i, k, k) = G(i, k, k - 1) \qquad \text{et} \qquad G(k, j, k) = G(k, j, k - 1)$$

9. Dérivez un algorithme avec la même complexité que le précèdent mais qui utilise une seule matrice $n \times n$ (il effectue tous les calculs sur place).
10. Écrire la fonction `C` qui correspond à l'algorithme optimisé de la question précédente.

A.16 Algorithme de Kruskal pour le calcul d'un arbre de recouvrement

Soit $G = (N, A)$ un graphe non dirigé connecté avec $n = \sharp N$ et $m = \sharp A$ (n noeuds et m arêtes). Dans la suite, on suppose que G est réprésenté par un tableau avec n entrées où l'entrée i pointe à la liste des noeuds adjacents au noeud i. On se réfère au tableau en question comme *tableau des listes d'adjacence*. Un *arbre* est un graphe non dirigé *connecté* et *acyclique*. Un *arbre de recouvrement* pour G est un sous-graphe de G qui est un arbre avec n noeuds. On considère l'algorithme A suivant :

> Entrée : le graphe G représenté par un tableau des listes d'adjacence.
>
> Calcul :
> $T = \emptyset$ (le sous-graphe vide)
> $a_1, \ldots, a_m$ énumeration des arêtes de G
> for $(i = 1; i <= m; i++)\{$
> if$(T \cup \{a_i\}$ acyclique$)\{T = T \cup \{a_i\}\}$
>
> Sortie : T .

1. Montrez que l'algorithme A calcule un arbre de recouvrement (à défaut, vous pouvez supposer ce résultat).

On suppose les déclarations suivantes :

```
struct node {int name; struct node * next;};
struct edge {int name1; int name2; struct edge * enext;};
struct edge *allocate_edge(int n1, int n2){
   struct edge *p=(struct edge *)(malloc(sizeof(struct edge)));
   (p->name1)=n1; (p->name2)=n2; (p->enext)=NULL;  return p;}
struct node * tabnode[n];
```

2. Programmez une fonction d'en tête :
   `struct edge * enum_edge(int n, struct node * tabnode[n])`
   qui prend en argument le nombre de noeuds et le tableau des listes d'adjacence et retourne un pointeur à une liste qui contient toutes les arêtes du graphe (exactement une fois). Analysez la complexité asymptotique de votre fonction.
3. Programmez une fonction d'en tête :
   `short accessible(int n, struct node * tabnode[n], int i, int j)`
   qui prend en argument le nombre de noeuds, le tableau des listes d'adjacence et deux noeuds i et j et retourne 1 si les noeuds sont connectés dans le graphe et 0 autremenet.
4. En utilisant vos reponses aux questions 2 et 3, analysez la complexité d'une mise en oeuvre de l'algorithme A en fonction de m et n Est-ce possible d'avoir une borne $O(n^3)$? Et une borne $O(n^2)$?

La terminologie suivante est (assez) standard. Soit $N = \{0, \ldots, n-1\}$ un ensemble fini non vide. Une *partition* P de N est un ensemble $\{S_1, \ldots, S_k\}$ tel que $\bigcup_{i=1,\ldots,k} S_i = N$, $S_i \neq \emptyset$ et $S_i \cap S_j = \emptyset$ si $i \neq j$. On appelle chaque élément de P une *classe d'équivalence*. On introduit une structure de données pour représenter les partitions de N et qui permet d'exécuter deux opérations :

— equal(i,j) décide si les éléments i et j sont dans la même classe d'équivalence de la partition.
— union(i,j) génère une nouvelle partition dans laquelle les classes d'équivalence de i et j sont fusionnées.

On suppose les déclarations de type suivantes :

```
struct eqclass {int count; struct node * head;};
struct eqclass * belongs[n];
```

On utilise ces déclarations de la façon suivante :

— une struct eqclass sert à représenter une classe d'équivalence : nombre d'éléments dans la classe et pointeur au premier struct node d'une liste qui contient les éléments de la classe.
— le tableau belong sert à affecter à chaque élément de N sa classe d'équivalence (un pointeur vers une struct eqclass).

5. Décrivez (avec un dessein de préférence) une représentation possible de la partition suivante :

$$P = \{\{0,3\},\{1,2,4\},\{5,6\}\} ,$$

qui utilise les struct class et le tableau belong ci-dessus. Aussi expliquez les opérations qu'il faut faire pour implémenter l'opération equal(0,1) et l'opération union(2,5).

6. Peut-on implementer la fonction equal en temps $O(1)$ et la fonction union en temps $O(n)$? Expliquez.
7. Programmez une fonction d'en tête short equal(int n, struct eqclass $*$ belong[n], int i, int j) qui prend en entrée le nombre de noeuds n, le tableau belong et deux noeuds i et j et retourne 1 si les deux noeuds sont dans la même classe d'équivalence et 0 autrement.
8. Programmez une fonction d'en tête void union(int n, struct eqclass $*$ belong[n], int i, int j) qui prend en entrée le nombre de noeuds n, le tableau belong et deux noeuds i et j et fait l'union des classes d'équivalence de i et j (on supposera que i n'est pas dans la même classe que j).

Considérons la situation où à partir de la partition $P = \{\{0\},\{1\},\ldots,\{n-1\}\}$ on effectue $n-1$ opérations union. Supposons aussi que chaque fois qu'on fait l'union de deux classes d'équivalencce de la partition on effectue un travail qui est linéaire dans la taille de la classe d'equivalence plus petite (les éléments de la classe plus petite rejoignent ceux dans la classe plus grande).

9. Montrez que le coût de $(n-1)$ opérations union est $O(n \log n)$. Suggestion : combien de fois un elément de $N = \{0,\ldots,n-1\}$ peut-il changer de classe d'équivalence' ?
10. Comment peut-on utiliser la structure de données présentée dans le contexte de l'algorithme A ? Quelle est la complexité de l'algorithme A dans ce cas ?
11. Supposons que les arêtes du graphe G sont pondérées par des poids non-négatifs. Est-ce possible d'adapter l'algorithme A de façon à qu'il calcule un arbre de recouvrement de poids minimal ? Expliquez.

Bibliographie

[AB98] Mohamad Akra and Louay Bazzi. On the solution of linear recurrence equations. *Computational Optimization and Applications*, 10(2) :195–210, 1998.

[AKS04] Manindra Agrawal, Neeraj Kayal, and Nitin Saxena. PRIMES is in P. *Annals of Mathematics*, 160(2) :781–793, 2004.

[Bel54] Richard Bellman. The theory of dynamic programming. *Bulletin of the AMS*, 60(6) :503–516, 1954.

[Ben84] Jon Bentley. Programming pearls : algorithm design techniques. *Communications of the ACM*, 27(9) :865–873, 1984.

[BW88] Hans-Juergen Boehm and Mark Weiser. Garbage collection in an uncooperative environment. *Softw., Pract. Exper.*, 18(9) :807–820, 1988.

[CLRS09] Thomas Cormen, Charles Leiserson, Ronald Rivest, and Clifford Stein. *Introduction to algorithms.* MIT Press, 2009. Troisième édition. Existe aussi en français.

[CT65] James Cooley and John W. Tukey. An algorithm for the machine calculation of complex Fourier series. *Math. Comp.*, 19 :297–301, 1965.

[Dan48] George Dantzig. Programming in a linear structure. Technical report, United States Air Force, Washington DC., 1948.

[Dij59] Edsger Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1 :269–271, 1959.

[EK72] Jack Edmonds and Richard M. Karp. Theoretical improvements in algorithmic efficiency for network flow problems. *J. ACM*, 19(2) :248–264, 1972.

[FF56] Lester Ford and Delbert Fulkerson. Maximal flow through a network. *Canadian journal of mathematics*, 8(3) :399–404, 1956.

[Hoa61] C. A. R. Hoare. Algorithm 64 : Quicksort. *Commun. ACM*, 4(7) :321, 1961.

[Huf52] David Huffman. A method for the construction of minimum-redundancy codes. *Proceedings of the IRE*, 40(9) :1098–1101, 1952.

[Kar84] Narendra Karmarkar. A new polynomial-time algorithm for linear programming. *Combinatorica*, 4(4) :373–396, 1984.

[Kha79] Leonid Khachiyan. A polynomial algorithm in linear programming. *Akademiia Nauk SSSR. Doklady*, 244 :1093–1096, 1979.

[KO62] Anatoly Karatsuba and Yuri Ofman. Multiplication of many-digital numbers by automatic computers. *Proceedings of the USSR Academy of Sciences*, 145 :293–294, 1962. Traduction dans *Physics-Doklady*, 7 (1963).

[KP17] Brian Kernighan and Robert Pike. *La programmation en pratique.* Vuibert, 2017.

[KR14] Brian Kernighan and Dennis Ritchie. *Le langage C.* Dunod, 2014.

[Mil76] Gary L. Miller. Riemann's hypothesis and tests for primality. *J. Comput. Syst. Sci.*, 13(3) :300–317, 1976.

[MPS92] J. Ian Munro, Thomas Papadakis, and Robert Sedgewick. Deterministic skip lists. In *Proceedings of the Third Annual ACM/SIGACT-SIAM Symposium on Discrete Algorithms, 27-29 January 1992, Orlando, Florida.*, pages 367–375, 1992.

[Pri57] Robert Prim. Shortest connection networks and some generalizations. *Bell System Technical Journal*, 36(6) :1389–1401, 1957.

[Pug90] William Pugh. Skip lists : A probabilistic alternative to balanced trees. *Commun. ACM*, 33(6) :668–676, 1990.

[Rab80] Michael O. Rabin. Probabilistic algorithms in finite fields. *SIAM J. Comput.*, 9(2) :273–280, 1980.

[Sho05] Victor Shoup. *A computational introduction to number theory and algebra*. Cambridge University Press, 2005. Disponible en ligne.

[ST04] Daniel A. Spielman and Shang-Hua Teng. Smoothed analysis of algorithms : Why the simplex algorithm usually takes polynomial time. *J. ACM*, 51(3) :385–463, 2004.

[Str69] Volker Strassen. Gaussian elimination is not optimal. *Numer. Math.*, 13 :354–356, 1969.

Index