Informatique

CPGE

Marc de Falco

Table des matières

Introduction

1. Introduction

Ce site présente des documents personnels autour de l'informatique.

Il s'agit pour le moment d'un premier jet, et il est amené à beaucoup évoluer. Il correspond à mon interprétation personnelle de certaines notions.

Pour le moment, ce sont des développements indépendants. Dans un second temps, ils seront éventuellement articulés autour d'une progression.

Des problèmes sont présentés au cours des documents, ils peuvent donner lieu à des développements en classe, des séances de travaux pratiques ou des problèmes. Plutôt que de donner un découpage figé en questions, ils sont présentés tels quels.

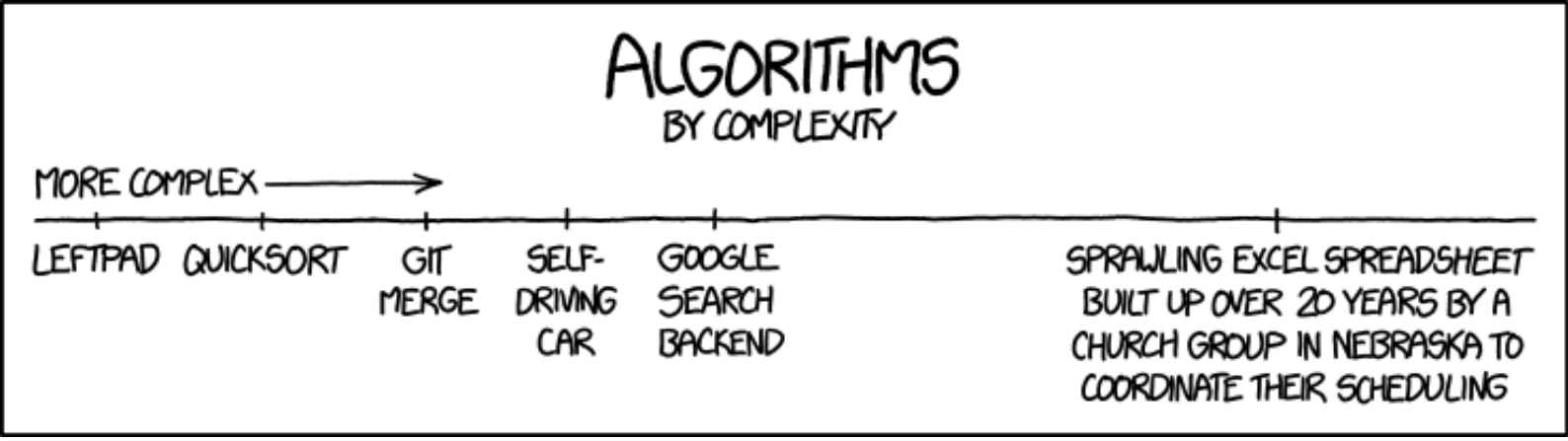
Mon idée principale avec ces ressources est de proposer un contenu assez riche, plutôt à destination des enseignants. Certains points un peu pointus ne sont pas forcément essentiels pour tous les étudiants.

Le but ici n'est pas de surpasser des références sur chaque domaine, je n'ai pas cette prétention, mais de présenter un livre *tout-en-un* ce qui implique des compromis dans les notions traités et la présentation.

I Feuille de route

— Faire une présentation uniforme pour les problèmes avec un index.
— Rajouter des macros pour compiler/exécuter et inclure la sortie des bouts de code.

II Algorithmique



2. Introduction à l'analyse des algorithmes

Source de l'image d'en-tête XKCD #1667

■ **Remarque 2.1** Ce chapitre présente les trois grands principes qui nous serviront de guide pour analyser les algorithmes et les programmes :

- La **terminaison** : l'algorithme termine-t-il au bout d'un nombre fini d'étapes quelle que soit l'entrée ?
- La **correction** : le résultat rendu est-il celui qui était attendu ?
- La **complexité** : combien de temps prend le programme selon la taille de l'entrée ? Combien d'espace mémoire occupe-t-il ?

Savoir répondre à ces questions, c'est pouvoir prédire, avant d'avoir écrit la moindre ligne de code, ce qui va se passer.

■

I Terminaison

Définition I.1 On dit qu'un algorithme **termine** quand il n'exécute qu'un nombre fini d'étapes sur toute entrée.

■ **Remarque 2.2** Cela n'empêche pas que ce nombre d'étapes puisse être arbitrairement grand en fonction des entrées.

■

Un algorithme qui n'utilise ni boucles inconditionnelles ni récursivité termine toujours. Ainsi, la question de la terminaison n'est à considérer que dans ces deux cas.

Considérons par exemple l'algorithme suivant qui, étant donné un entier naturel n strictement positif, inférieur à 2^{30}, détermine le plus petit entier k tel que $n \leq 2^k$:

```
int plus_grande_puissance2(int n)
{
    int k = 0;
    int p = 1;
```

```
    while (p < n)
    {
        k = k+1;
        p = p*2;
    }

    return k;
}
```

On remarque que `p` prend successivement pour valeurs toutes les puissances de 2 jusqu'à une éventuelle sortie de boucle. Or, il existe une puissance de 2 supérieure ou égale à `n`, donc, une fois atteinte, la condition de la boucle `while` n'est plus remplie et l'algorithme termine.

I.1 Variant de boucle

Pour prouver la terminaison d'une boucle conditionnelle, on utilise un **variant** de boucle.

Définition I.2 Un **variant de boucle** est une quantité entière **positive** à l'entrée de chaque itération de la boucle et qui diminue **strictement** à chaque itération.

Ainsi, si on a un variant de boucle qui vaut initialement n avant d'entrer dans la boucle, celle-ci effectue au plus n itérations car le variant diminue au moins de 1 à chaque étape.

Si une boucle admet un variant de boucle, elle termine.

Dans l'exemple précédent, la quantité $n - p$ est un variant de boucle :

- Au départ, $n > 0$ et $p = 1$ donc $n - p \geq 0$
- Comme il s'agit d'une différence de deux entiers, c'est un entier. Et tant que la condition de boucle est vérifiée $p < n$ donc $n - p > 0$.
- Lorsqu'on passe d'une itération à la suivante, la quantité passe de $n-p$ à $n-2p$ or $2p-p > 0$ car $p \geq 1$. Il y a bien une stricte diminution.

■ **Remarque 2.3** Ici, en sortie de boucle, $n - p \leq 0$. On fait donc bien attention de préciser que la quantité est positive tant que la condition de boucle est vérifiée.

■

I.2 Exemple de la recherche dichotomique

On considère ici la recherche dichotomique dans un tableau trié d'entiers. Étant donné un tableau t de taille $n > 0$ et un entier x dont on cherche à déterminer sa présence dans le tableau entre les indices i et j, on considère l'algorithme suivant :

- Si $i > j$, alors il n'y a pas de sous-tableau et on renvoie `false`.
- Sinon, soit m l'élément d'indice $p = \left\lfloor \frac{i+j}{2} \right\rfloor$.
  - Si $x = m$, on renvoie `true`
  - Si $x < m$, on continue la recherche dans le sous-tableau des indices i à $p - 1$.
  - Si $x > m$, on continue la recherche dans le sous-tableau des indices $p + 1$ à j.

Le programme suivant présente une implémentation de cet algorithme en C.

```
int rech_dicho(int *t, size_t n, int x)
{
    /* renvoie un indice de x
      si x est dans le tableau et -1 sinon */
    size_t i = 0;
    size_t j = n-1;
```

```
    while (i <= j)
    {
        size_t p = i + (j-i)/2;
        int m = t[p];

        if (x == m)
            return p;
        else if (x < m)
                j = p-1;
             else
                i = p+1;
    }

    return -1;
}
```

■ **Remarque 2.4** On a écrit `i+(j-i)/2` et non `(i+j)/2` afin d'éviter des erreurs de dépassement dans le calcul de `i+j`.

■

Ici, la terminaison n'est pas immédiate, on va la prouver à l'aide d'un variant de boucle. On considère ainsi la quantité $d(i,j) = j - i$.

- Comme le tableau est non vide, $d(0, n-1) \geq 0$. Ensuite, la condition de boucle est équivalente à $d(i,j) \geq 0$, donc cette quantité est bien entière et positive à l'entrée de chaque itération.
- Quand on passe à l'itération suivante, on passe
  - soit de $d(i,j)$ à $d(i, p-1)$. Or $d(i,j) - d(i,p-1) = j - i - \frac{i+j}{2} + 1 + i = 1 + \frac{j-i}{2} = 1 + \frac{d(i,j)}{2} > 0$.
  - soit de $d(i,j)$ à $d(p+1,j)$. Or $d(i,j) - d(p+1,j) = j - i - j + \frac{i+j}{2} + 1 + i = 1 + \frac{j-i}{2} > 0$.

  Dans tous les cas, $d(i,j)$ diminue strictement.

Ainsi, il s'agit d'un variant de boucle et l'algorithme termine.

■ **Remarque 2.5** On remarque que le programme récursif suivant réalise également cet algorithme.

```
(* rech_dicho : int array -> int -> int -> int -> int option *)
let rec rech_dicho t i j x =
    if i <= j
    then let p = (i+j)/2 in
         let m = t.(p) in
         if x = m
         then Some p
         else if x < m
         then rech_dicho t i (p-1) x
         else rech_dicho t (p+1) j x
    else None
```

Il suffit alors d'appeler `rech_dicho t 0 (Array.length t - 1) x` pour faire une recherche sur tout le tableau.

Ici, il n'y a pas explicitement de boucle mais le même principe peut être mis en place pour prouver que le nombre d'appel récursifs est majoré, et donc que toute exécution termine. En effet, la quantité $d(i,j) = j - i$ diminue pour les mêmes raisons à chaque appel récursif et reste entière positive.

■

II Correction

Pour parler de correction d'un algorithme, il est nécessaire d'identifier précisément ce qui doit être calculé par l'algorithme. Pour cela, on considère ici informellement des spécifications dépendant des entrées et du résultat de l'algorithme. On verra dans le chapitre sur la logique qu'il s'agit ici de prédicats logiques.

Voici des exemples de spécifications :

— le tableau t en sortie est trié dans l'ordre croissant

— la valeur renvoyée est le plus petit indice de x dans le tableau ou -1 s'il ne le contient pas.

Définition II.1 Un algorithme est **correct** vis-à-vis d'une spécification lorsque quelle que soit l'entrée

— il **termine**

— le résultat renvoyé vérifie la spécification.

On considère également la correction **partielle** en l'absence de terminaison :

Définition II.2 Un algorithme est **partiellement correct** vis-à-vis d'une spécification lorsque quelle que soit l'entrée le résultat renvoyé vérifie la spécification.

II.1 Invariant de boucle

Définition II.3 On considère ici une boucle (conditionnelle ou non).

Un prédicat est appelé un **invariant de boucle** lorsque

— il est vérifié avant d'entrer dans la boucle

— s'il est vérifié en entrée d'une itération, il est vérifié en sortie de celle-ci.

Quand la boucle termine, on déduit alors que l'invariant est vérifié en sortie de boucle. On cherche donc un invariant qui permette de garantir la spécification en sortie de boucle.

■ **Remarque 2.6** Pour les boucles inconditionnelles, il y a une gestion implicite de l'indice de boucle qui va se retrouver dans l'invariant. On peut alors considérer que la sortie de boucle s'effectue après être passé à l'indice suivant.

■

Dans le cas d'une boucle conditionnelle portant sur la condition P et ayant un invariant de boucle I, en sortie le prédicat $\neg P \wedge I$ (non P et I) sera vérifié.

On peut illustrer cela en reprenant la fonction `plus_grande_puissance2` vue à la partie Terminaison. On considère ici le prédicat $I(k,p) := 2^{k-1} < n$ et $p = 2^k$.

— En entrée de boucle, on a bien $2^{-1} < n$.

— Si le prédicat est vérifié en entrée d'itération. On a alors $2^{k-1} < n$ et comme on est entrée dans cette itération $p = 2^k < n$. Donc en sortie d'itération on aura bien $I(k+1, 2p)$ car $2p = 2^{k+1}$.

Ainsi, ce prédicat est bien un invariant et en sortie de boucle (ce qui arrive nécessairement car l'algorithme termine), le prédicat $I(k,p)$ signifie que $2^{k-1} < n$ **et la condition de sortie de boucle** qu'on a $n \leq 2^k$.

La valeur renvoyée est bien k tel que $2^{k-1} < n \leq 2^k$ ce qui était la spécification annoncée du programme.

II.2 Exemple du tri par sélection

Le programme suivant présente un algorithme de tri, appelé le *tri par sélection* dont on va analyser la complexité. Il s'agit d'un tri qui repose sur un principe simple, on va chercher le plus petit élément du tableau à trier et le placer à la position courante. On définit ainsi trois fonctions :

— `echange` réalise l'échange de valeurs entre deux cases du tableau

— `indice_minimum` renvoie l'indice de la plus petite valeur entre deux indices donnés
— `tri_par_selection` réalise le tri en parcourant le tableau du premier au dernier indice et en plaçant à la position courante le minimum restant.

```
void echange(int *tableau, int i, int j)
{
    int temp = tableau[i];
    tableau[i] = tableau[j];
    tableau[j] = temp;
}

void indice_minimum(int *tableau, int min_indice, int max_indice)
{
    int i = min_indice;

    for (int j = min_indice + 1; j <= max_indice; j++)
    {
        if (tableau[j] < tableau[i])
            i = j;
    }

    return i;
}

void tri_par_selection(int *tableau, int taille)
{
    for (int i = 0; i < taille; i++)
    {
        echange(tableau, i, indice_minimum(tableau, i, taille-1));
    }
}
```

Il n'y a pas de problèmes de terminaison ici car toutes les boucles sont inconditionnelles. Pour prouver sa correction, on va considérer séparément les deux boucles.

— Boucle dans `indice_minimum` : on va valider l'invariant $I(i,j) := \forall k \in [\![i, j-1]\!]$, tableau$[i] \leq$ tableau$[k]$.
  — En entrée de boucle, on a $I(\text{min\_indice}, \text{min\_indice}+1)$ vérifié directement.
  — Si en entrée d'itération, $I(i,j)$ est vérifié, ce qui signifie que tableau$[i]$ est plus petit que les valeurs compris entre les indices i et $j-1$. Alors, on distingue deux cas :
    — soit tableau$[j] <$ tableau$[i]$ et alors en sortie i devient $i' = j$. On a alors tableau$[i'] =$ tableau$[j] <$ tableau$[i] \leq$ tableau$[k]$ pour $k \in [\![1, j-1]\!]$. Donc $I(i', j+1)$ est vérifié.
    — soit tableau$[i] \leq$ tableau$[j]$ et ainsi on a pu prolonger le prédicat à $I(i, j+1)$.

  Ce prédicat est bien un invariant. Ainsi, en sortie de boucle, et donc avant de renvoyer sa valeur, on a bien $I(i, \text{taille})$ donc tableau$[i]$ est la plus petite valeur du tableau.
— Boucle dans `tri_par_selection` : on va valider l'invariant `$T(i) := $` le sous-tableau `tableau[0..i-1]` des indices 0 à $i-1$ est trié et ne contient que des valeurs plus petites que celles du sous-tableau `tableau[i..taille-1]`.
  — En entrée de boucle, le sous-tableau est vide donc trié.
  — Si en entrée d'itération, le prédicat est vérifié. On récupère l'indice j du minimum du sous-tableau `tableau[i..taille-1]` à l'aide la fonction `indice_minimum`, par hypothèse `tableau[j]` est alors supérieur ou égal à chaque élément de `tableau[0..i-1]`, en le plaçant à l'indice i, on a bien `tableau[0..i]` qui est trié et par construction

la valeur de `tableau[i]` est inférieure à toutes celels de `tableau[i+1..taille-1]`. On a ainsi $T(i+1)$ vérifié en sortie d'itération.

Ce prédicat est bien un invariant. Ainsi, en sortie de boucle, $T(\text{taille})$ est vérifié : le tableau est trié.

III Complexité

III.1 Complexité dans le pire des cas

Considérons un algorithme pour lequel on peut associer à chaque entrée une notion de taille (par exemple le nombre d'éléments d'un tableau). Pour $n \in \mathbb{N}$, on note ainsi I_n l'ensemble des entrées de taille n pour cet algorithme. Pour une entrée e, on note $t(e)$ le temps pris, par exemple en seconde, par l'algorithme sur l'entrée e. De même, on note $s(e)$ l'espace mémoire maximal, par exemple en octets, occupé par l'algorithme au cours de cette exécution **sans compter la taille des entrées**.

Définition III.1 On appelle :
- **complexité temporelle dans le pire des cas**, la suite $(C_n^t)_{n\in\mathbb{N}}$ telle que pour tout $n \in \mathbb{N}$, $C_n^t = \max_{e\in I_n} t(e)$.
- **complexité spatiale dans le pire des cas**, la suite $(C_n^s)_{n\in\mathbb{N}}$ telle que pour tout $n \in \mathbb{N}$, $C_n^s = \max_{e\in I_n} s(e)$.

Comme on va le voir, calculer explicitement ces suites n'a pas beaucoup d'intérêt tant elles sont dépendantes de la manière dont on mesure le temps et l'espace. Ce qui compte ici, c'est de connaître l'*ordre de grandeur* de ces complexités en fonction de n.

Pour un tableau de taille n, ce programme va effectuer n itérations et sa complexité est ainsi de l'ordre de n. Il est possible d'être très précis en considérant les temps pris
- pour mettre en place l'appel de fonction et le passage des arguments
- par la gestion de l'indice de la boucle for
- pour la comparaison, puis pour l'affectation éventuelle
- pour mettre en place la valeur de retour afin que le résultat soit lu

On peut remarquer que la notion de pire cas dépend de la précision à laquelle on se place. Ici, si on ne s'intéresse qu'à l'ordre de grandeur, tous les tableaux de taille n sont équivalents. Par contre, si on cherche avec précision le pire cas, il est atteint avec un tableau trié par ordre croissant car c'est le cas qui effectue une affectation à chaque itération.

```c
int maximum(int *tableau, int taille)
{
    int M = INT_MIN;

    for(int i=0; i<taille; i++)
    {
        if (M > tableau[i])
            M = tableau[i];
    }

    return M;
}
```

III.2 Comparer des complexités

Avant de pouvoir comparer les complexités des algorithmes ou des programmes, il est nécessaire de mettre en place des outils pour en parler à la fois avec précision mais également sans rentrer dans des détails inextricables d'implémentation.

En effet, comparons les deux fonctions suivantes permettant de calculer le maximum d'un tableau non vide :

```c
int maximum(int *tableau, int taille)
{
    int M = INT_MIN;

    for(int i=0; i<taille; i++)
    {
        if (M > tableau[i])
            M = tableau[i];
    }

    return M;
}
```

```c
int maximum(int *tableau, int taille)
{
    int M = tableau[0];

    for(int i=1; i<taille; i++)
    {
        if (M > tableau[i])
            M = tableau[i];
    }

    return M;
}
```

La fonction de gauche semble moins efficace que celle de droite car elle effectue une itération de boucle de moins. Mais on doit se poser la question de la pertinence de cette optimisation selon la taille du tableau.

De la même manière, il faut déterminer ce que l'on souhaite compter précisément :

— si on s'intéresse au temps mis, certaines opérations prennent moins de temps que d'autre (par exemple une addition par rapport à une multiplication) mais est-ce vraiment important à l'échelle considérée ?
— si on s'intéresse à l'espace mémoire, doit-on considérer la taille précise en octets ou se contenter d'une estimation plus grossière ?

Mis à part dans certains cadres assez spécifiques, on se contente le plus souvent d'un ordre de grandeur pour ces complexité. Pour cela, on utilise des relations de comparaisons de suites et une échelle de grandeur usuelle pour les comparer.

■ **Note 2.1** Redite ici avec la partie précédente. A reprendre.

■

La notation grand O

Définition III.2 Soit $(u_n)_{n\in\mathbb{N}}$ et $(v_n)_{n\in\mathbb{N}}$ deux suites de nombres réels non nuls, on dit que la suite $(u_n)_n$ est dominée par $(v_n)_n$ lorsque la suite quotient $\left(\frac{u_n}{v_n}\right)_n$ est bornée.

On note alors $u_n = O(v_n)$.

Cette dernière notation se lit u_n *est un grand* O *de* v_n.

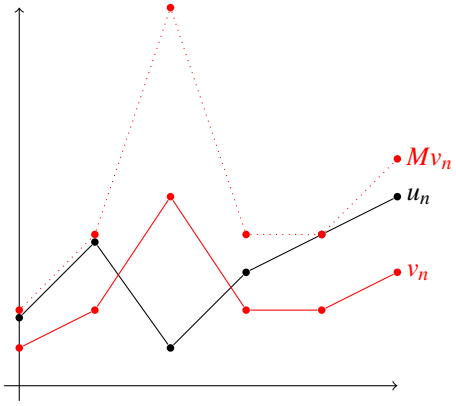
■ **Remarque 2.7** C'est bien cette locution qu'il faut avoir en tête quand on pense aux grands O et il faut faire attention de ne pas considérer l'égalité en tant que telle sans s'assurer que ce l'on fait est licite. Quand on écrira par la suite $O(v_n)$ on signifiera *n'importe quelle suite qui soit un* $O(v_n)$.

■

Si $u_n = O(v_n)$, cela signifie qu'il existe un facteur $M > 0$ tel que pour tout entier n, on ait $-M|v_n| \leq u_n \leq M|v_n|$. Les variations de la suite $(u_n)_n$ sont ainsi entièrement contrôlées par les variations de $(v_n)_n$.

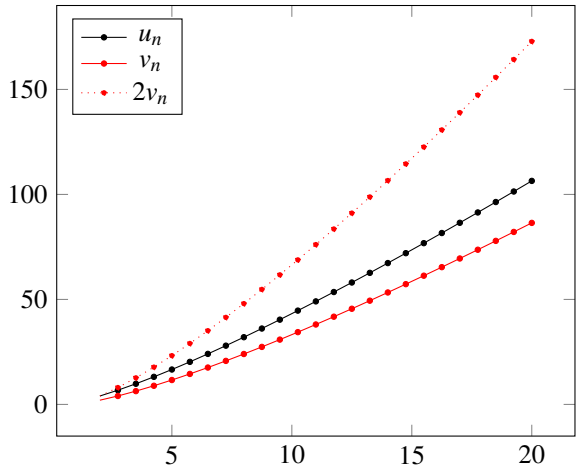
En informatique, on ne considère pour la complexité que des suites positives, ce qui permet de simplifier la relation : si $(u_n)_n, (v_n)_n$ sont des suites de réels strictement positifs, alors $u_n = O(v_n) \iff \exists M > 0, \forall n \in \mathbb{N}, u_n \leq Mv_n$. C'est le cadre dans lequel on se place implicitement dans la suite de ce document.

On peut visualiser graphiquement cette relation :

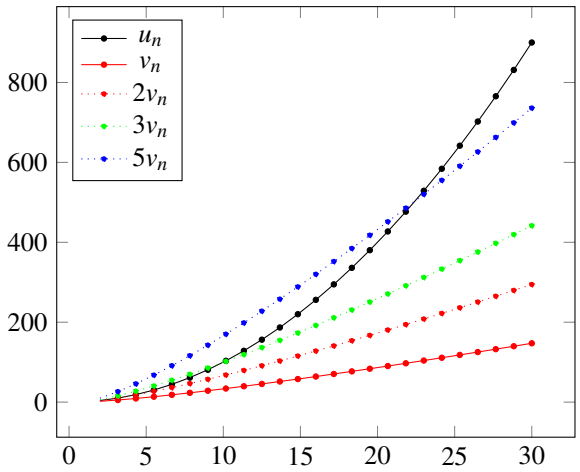


On a $u_n = O(v_n)$ si et seulement s'il est possible de multiplier les ordonnés de chaque point (n, v_n) par une constante afin que ces nouveaux points soient tous au-dessus des points (n, u_n). On peut voir que la courbe déduite des v_n enveloppe, à un facteur près, celle des u_n.
Remarque : *On a relié ici les valeurs des suites pour mieux mettre en valeur cette notion d'enveloppe.*

Cette relation est une notion **asymptotique** : elle n'a d'intérêt que lorsqu'on considère des rangs au voisinage de l'infini. En effet, pour un nombre fini de termes, il est toujours possible de trouver un tel M, mais pour un nombre infini, ce n'est pas le cas.



Ici, on compare asymptotiquement les suites $(u_n)_n$ et $(v_n)_n$ où pour $n \in \mathbb{N}$, $u_n = n + n\log_2 n$ et $v_n = n\log_2 n$. Pour simplifier la visualisation, on a tracé les fonctions correspondantes. On remarque qu'on a bien $n + n\log_2 n = O(n\log_2 n)$.



Par contre, si on compare les suites $(u_n)_n$ et $(v_n)_n$ où pour $n \in \mathbb{N}$, $u_n = n^2$ et $v_n = n\log_2 n$, on remarque que quelle que soit la valeur choisie pour M, il y aura un rang à partir duquel $u_n > Mv_n$.
Ici, $n^2 \neq O(n\log_2 n)$.

■ **Remarque 2.8** On a ici utilisé le logarithme en base 2, noté $\log_2$, qui est essentiel informatique : si $x = \log_2(n)$ alors $n = 2^x$ où x est un réel. On considère aussi $p = \lceil \log_2(n) \rceil$ qui est le plus petit entier égal ou supérieur à $\log_2(n)$. On parle de **partie entière supérieure** et on a alors $2^{p-1} < n \leq 2^p$. Cet entier p correspond alors au plus petit nombre de chiffre nécessaire pour pouvoir écrire n en binaire. On a $\lceil \log_2(n) \rceil = O(\log_2(n))$ et ainsi, le plus souvent, on ne

considère pas la partie entière explicitement. De la même manière, $\log_2(n) = \frac{\ln n}{\ln 2} = O(\ln n)$.

■

Un cas important de grand O est celui des $O(1)$. Si $u_n = O(1)$, cela signifie que $(u_n)_{n\in\mathbb{N}}$ est une suite bornée.

Échelle de comparaison

On rappelle les limites obtenues en mathématiques que l'on nomme **croissances comparées** :

$$\forall \alpha, \beta > 0, \lim_{n\to+\infty} \frac{(\ln n)^\alpha}{n^\beta} = 0$$

$$\forall \alpha \in \mathbb{R}, \forall \beta > 1, \lim_{n\to+\infty} \frac{n^\alpha}{\beta^n} = 0$$

Or, si $\frac{u_n}{v_n} \xrightarrow[n\to+\infty]{} 0$ *a fortiori* le quotient est borné et $u_n = O(v_n)$. Ainsi, on a les relations suivantes :

$$\forall \alpha, \beta > 0, (\log_2 n)^\alpha = O(n^\beta)$$

$$\forall \alpha \in \mathbb{R}, \forall \beta > 1, n^\alpha = O(\beta^n)$$

De plus, si $\alpha \geq \beta > 0$, $n^\beta = O(n^\alpha)$, $(\log_2 n)^\beta = O((\log_2 n)^\alpha)$ et $\beta^n = O(\alpha^n)$.

On se ramène souvent à des complexités qui sont des grand O de produits de ces suites.

Ordre de grandeur et relation Θ

On vient de voir que $\log_2 n = O(n)$, mais on a également $\log_2 = O(n^2)$. Quand on cherche à caractériser la complexité par un grand O, on va souvent chercher le grand O le plus proche de la suite.

Il est possible de définir cela précisément en considérant des suites qui sont chacune des grand O l'une de l'autre.

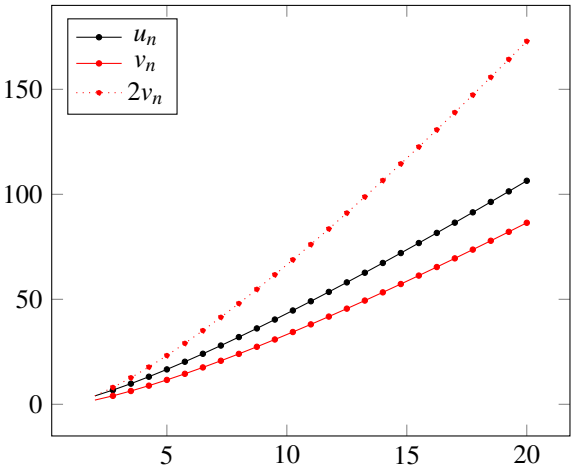
Par exemple, on a vu que $n \log_2 n + n = O(n \log_2 n)$, mais on a également $n \log_2 n = O(n + n \log_2 n)$.

Quand $u_n = O(v_n)$ et $v_n = O(u_n)$, on note $u_n = \Theta(v_n)$ qui est une relation symétrique qui correspond à la notion *avoir le même ordre de grandeur*. Très souvent, lorsque l'on parle de complexité, on utilise des grand O quand, en fait, on exprime des Θ. Par exemple, l'accès à un élément dans un tableau est en $O(1)$ et il ne serait pas précis de dire que c'est en $O(n)$ **même si c'est parfaitement correct**.

On peut visualiser cette relation Θ en considérant qu'il existe ainsi $M, M' > 0$ tels que $u_n \leq M v_n$ et $v_n \leq M' u_n$. Mais on a alors

$$1/M' v_n \leq u_n \leq M v_n$$

Ainsi, $u_n = \Theta(v_n)$ signifie qu'on peut encadrer $(u_n)_n$ entre deux multiples de $(v_n)_n$.



En reprenant la figure précédente, on observe visuellement

$$n \log_2 n \leq n \log_2 n + n \leq 2n \log_2 n$$

Avoir $u_n = \Theta(v_n)$ signifie donc que u_n évolue entre deux guides suivant les variations de v_n.

Opérations sur les grands O

Si $u_n = O(w_n)$ et $v_n = O(w_n)$ alors $u_n + v_n = O(w_n)$. Ainsi, des grand O de même ordre s'ajoutent.

■ **Remarque 2.9** Comme on l'a vu précédemment, un grand O n'est pas très précis, et il est possible que par ajout on puisse obtenir un meilleur grand O. Par exemple : $n = O(n)$ et $\log_2 n - n = O(n)$ mais $n + \log_2 n - n = \log_2 n = O(n)$. Comme on ne considère ici que des suites strictement positifs, ce phénomène de compensation n'aura pas lieu.

■

Si $u_n = O(v_n)$ et w_n est une autre suite de réels strictement positifs, alors $u_n w_n = O(v_n w_n)$. On en déduit ainsi un principe qui nous sera utile par la suite $nO(1) = O(n)$.

III.3 Complexités en temps classiques

On parle ici de complexité par raccourci pour parler de complexité dans le pire des cas en temps.

■ **Note 2.2** Pas convaincu de l'intérêt de ce raccourci par rapport à l'imprécision qui en résulte sur un chapitre d'introduction.

■

Complexité constante

On dit qu'un algorithme a une complexité constante quand $C_n^t = O(1)$. Il existe ainsi une constante M telle que le temps pris par l'algorithme **sur une entrée quelconque** soit inférieur à M.

De nombreuses opérations sont en temps constant sur les structures de données usuelles. Parmi celles-ci, citons-en deux essentielles :

— accéder à une case d'indice quelconque dans un tableau
— accéder à la tête ou à la queue d'une liste chaînée

Les algorithmes ou opérations en temps constant jouent un rôle primordiale dans l'analyse de la complexité d'algorithmes, comme on le verra dans la partie suivante, car elles permettent de se concentrer sur les répétitions de ces opérations pour déterminer la complexité : une boucle qui se répète n fois et n'effectue que des opérations en temps constant dans son corps sera de complexité $nO(1) = O(n)$.

Complexité linéaire

On dit qu'un algorithme a une complexité linéaire quand $C_n^t = O(n)$.

Cette complexité correspond à un traitement de temps constant sur chaque élément d'une entrée de taille n. C'est le cas de la recherche d'un élément dans un tableau ou de la recherche de son maximum.

Pour la recherche linéaire d'un élément, correspondant par exemple au programme ci-contre, le pire cas correspond à ne pas avoir `x` dans `tableau` ce qui oblige à effectuer toutes les itérations. On a bien une complexité temporelle en pire cas de $O(n)$.

```c
int recherche(int *tableau, int taille, int x)
{
    /* renvoie le plus petit indice i tel que tableau[i]
       ou -1 si x n'est pas dans le tableau */
    for(int i = 0; i < taille; i++)
    {
        if (tableau[i] == x)
            return i;
    }

    return -1;
}
```

Complexité quadratique, polynomiale

On dit qu'un algorithme a une complexité quadratique quand $C_n^t = O(n^2)$. Par extension, on dit qu'il a une complexité polynomiale quand il existe $k \in \mathbb{N}$ tel que $C_n^t = O(n^k)$. Par extension, on parle parfois de complexité polynomiale pour des complexité plus précise en $O(n^\alpha)$ où α est un réel strictement positif.

L'exemple classique d'un algorithme quadratique est celui dû à un double parcours d'un tableau. On reprend ici l'algorithme de tri par sélection vu dans la partie Exemple du tri par sélection.

Afin d'analyser sa complexité, on procède fonction par fonction pour un tableau de taille n :

- `echange` est en temps constant. $O(1)$
- `indice_minimum` réalise un parcours du tableau et effectue des opérations en temps constant à chaque étape. La complexité est donc linéaire. $O(n)$
- `tri_par_selection` réalise également un parcours du tableau mais à chaque étape, on appelle `indice_minimum` qui est en $O(n)$, la complexité est donc en $nO(n) = O(n^2)$: elle est quadratique.

Complexité logarithmique

On dit qu'un algorithme a une complexité logarithmique quand $C_n^t = O(\log_2 n)$.

Pour illustrer cette complexité, on reprend l'algorithme de recherche dichotomique vu dans la partie Exemple de la recherche dichotomique.

Chaque opération effectuée étant en temps constant, la complexité de cet algorithme correspond au nombre d'itérations, soit ici au nombre d'appels récursifs.

Si on considère un sous-tableau de $n = j - i + 1$ éléments lors de l'appel, un appel récursif se fera nécessairement sur un sous-tableau de $\lfloor n/2 \rfloor$ éléments. Ainsi, si $2^{k-1} < n \leq 2^k$, l'algorithme effectue moins de k itérations. En passant au logarithme, on a donc $k - 1 < \log_2 n \leq k$. Donc, le nombre d'itérations est en $O(\log_2 n)$ et c'est ainsi la complexité de l'algorithme.

■ **Note 2.3** Esquisser dès maintenant le lien entre longueur d'une branche dans un arbre de décision et complexité logarithmique ?

■

Complexité quasi-linéaire

On dit qu'un algorithme a une complexité quasi-linéaire quand $C_n^t = O(n \log_2 n)$. C'est le cas de la plupart des algorithmes efficaces de tri de n éléments. On peut même montrer qu'il s'agit de la complexité optimale.

Comme de nombreux algorithmes commencent par effectuer un tri avant d'effectuer un traitement linéaire, on retrouve des algorithmes quasi-linéaire par simple utilisation de ce tri.

Complexité exponentielle

On dit qu'un algorithme a une complexité exponentielle quand $C_n^t = O(a^n)$ pour $a > 0$.

Un exemple fondamental d'un tel algorithme est celui de l'énumération de données, par exemple pour chercher une solution par force brute. En effet, il y a 2^n entrées codées sur n bits et un algorithme cherchant une solution ainsi parmi ces entrées aura une complexité en $O(2^n)$.

Estimation de l'impact des complexités sur le temps

Afin de mesurer l'impact d'une complexité, on va considérer un algorithme qui s'exécute en 1 seconde sur un entrée de taille n, et on va calculer combien de temps prendrait ce même algorithme sur une entrée de taille $10n$.

Pour simplifier, on considère à chaque fois que C_n^t correspond exactement à l'ordre du grand O.

Complexité	Temps pour 10n	Temps pour 100n
1	1s	1s
$\log_2 n$	1,003s	1,007s
n	10s	1m40s
$n \log_2 n$	14,7s	3m13s
n^2	1m40s	2h46m40s
2^n	10^{19} années	10^{289} années.

■ **Remarque 2.10** Pour déterminer ces valeurs, on a considéré une unité de mesure de 1000ms afin d'en déduire une valeur de n.

Ainsi, si $\log_2 n = 1000$ on a $n = 2^{1000}$. Bien sûr, ici, ce nombre 2^{1000} n'est pas réaliste. Dans un contexte de mémoire finie, une complexité logarithmique est identifiable à une complexité constante. Cela justifie la terminologie quasi-linéaire.

Si $n \log_2 n = 1000$ alors $n \approx 140,2$. Or, $1402 \log_2 1402 \approx 14700ms$.

Si $2^n = 1000$, alors $n \approx 10$. Or $2^{100} \approx 10^{30}$.

■

III.4 Calculer des complexités

Deux principes fondamentaux pour calculer des complexités :

- Si on effectue deux passes successives chacune en $O(u_n)$ alors la complexité globale est en $O(u_n)$. Il ne s'agit que de reformuler l'addition des grand O. Quand on a deux passes de complexité différente, il suffit d'utiliser la plus grande complexité. Par exemple, un algorithme qui commence par un tri en $O(n \log_2 n)$ et qui effectue ensuite un traitement en $O(n)$ sera de complexité globale $O(n \log_2 n)$ car le traitement est également en $O(n \log_2 n)$.
- Si on effectue u_n itérations et que chaque itération est en $O(v_n)$ alors l'algorithme a une complexité de $O(u_n v_n)$. Cela permet de compter le nombre de boucles imbriquées et de se contenter de regarder ce qui se passe dans le corps des boucles.

III.5 Complexité à plusieurs paramètres

Jusqu'ici on a considéré des entrées dépendant d'un unique paramètre n, mais il est possible d'avoir des données dépendant de plusieurs paramètres.

On adapte directement la notation des grands O : si $(u_{n,p})$ et $(v_{n,p})$ sont deux suites de réels non nuls dépendant de deux paramètres, on note toujours $u_{n,p} = O(v_{n,p})$ quand le quotient est borné.

Données multidimensionnelles

Le cas le plus usuel de complexité dépendant de plusieurs paramètres est celui des données multidimensionnelle comme une image.

Si on considère une opération effectuant un traitement en temps constant sur chaque pixel d'une image de $w \times h$ pixels, cette opération aura une complexité en $O(wh)$. On ne peut plus parler de complexité linéaire ou quadratique ici car cela dépend d'une éventuelle relation entre w et h : si on ne travaille que sur des images de taille $1 \times h$ alors la complexité est $O(h)$, mais on ne travaille que sur des images carrées, donc pour lesquelles $w = h$, la complexité est $O(h^2)$.

Plus généralement, si on considère des données organisées dans des tableaux imbriqués, on effectuera un traitement sur chaque donnée à l'aide de boucles imbriquées non conditionnelles. La complexité sera alors celle du corps de boucles multipliée par le produit du nombre d'itérations de chaque boucle.

Compromis entre paramètres

Dans certains cas, en particulier pour les graphes, on peut effectuer des traitements successifs dont la complexité ne s'exprime pas en fonction du même paramètre. Imaginons par exemple un programme ayant la structure suivante :

```
for (int i = 0; i<n; i++)
{
    /* corps de boucle en O(1) */
}

for (int j = 0; j<p; j++)
{
    /* corps de boucle en O(1) */
}
```

La complexité de la première boucle est en $O(n)$ et celle de la deuxième en $O(p)$. La complexité globale est en $O(n + p)$ car $n \leq n + p$ et $p \leq n + p$.

III.6 Complexité en moyenne

On reprend ici les notations de la partie Complexité dans le pire des cas.

Définition III.3 Lorsque pour tout $n \in \mathbb{N}$, I_n est fini, on appelle :
- **complexité temporelle en moyenne** la suite $(C_n^{t,m}) = \frac{1}{|I_n|} \sum_{e \in I_n} t(e)$.
- **complexité spatiale en moyenne** la suite $(C_n^{s,m}) = \frac{1}{|I_n|} \sum_{e \in I_n} s(e)$.

On peut étendre cette définition à un cadre infini en considérant une distribution de probabilité sur I_n et T_n la variable aléatoire associée à t sur I_n. Si T_n est d'espérance finie, on pourra parler de complexité en moyenne pour la suite des $E(T_n)$. Concrètement, on considère alors une fonction $p_n : I_n \to [0, 1]$ telle que $\sum_{e \in I_n} p(e) = 1$ et, lorsque la somme est définie, on note ainsi

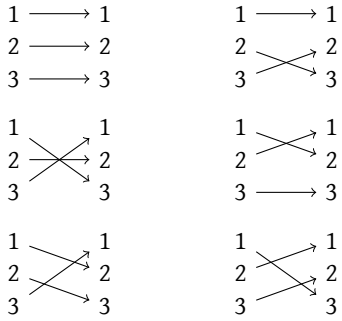
$$C_n^{t,m} = \sum_{e \in I_n} p(e)t(e)$$

$$C_n^{s,m} = \sum_{e \in I_n} p(e)s(e)$$

Un exemple usuel de calcul de complexité en moyenne est celui des tris. En effet, même si les entrées de taille n sont infinies, on peut considérer qu'un tableau de valeurs deux à deux distinctes est l'image par une permutation du tableau triée. Si le tableau est de taille n, on aura ainsi $n!$ permutations ce qui permet, du moment que l'algorithme de tri considéré ne dépend que cette permutation, de calculer la complexité en moyenne sur l'ensemble des permutations.

■ **Remarque 2.11** Les permutations d'un ensemble sont les applications bijectives de cet ensemble dans lui-même. Si l'ensemble contient n éléments, il y a $n!$ permutations.

Par exemple, les six permutations sur l'ensemble $\{1, 2, 3\}$ correspondent aux diagrammes sagittaires suivants :



Ces six permutations correspondant elles-mêmes, de gauche à droite et de haut en bas, aux tableaux `{1,2,3}`, `{1,3,2}`, `{3,2,1}`, `{2,1,3}`, `{2,3,1}` et `{3,1,2}`.

■

Exemple de calcul de complexité temporelle en moyenne

On considère la recherche linéaire vue dans la partie Complexité linéaire. L'ensemble des entrées est ici infini, on va donc supposer pour faire le calcul qu'on ne considère que des tableaux de valeurs deux à deux distinctes et qu'on recherche un élément présent dans le tableau, chaque élément étant équiprobable.

Si on recherche le i-ème élément du tableau, l'algorithme effectue i itérations avant d'y accéder et de renvoyer son indice. Ainsi, le temps pour cet entrée est de iC où C est le coût d'une itération.

La complexité temporelle en moyenne est alors $C_n^{t,m} = \sum_{i=1}^{n} \frac{1}{n} iC = \frac{(n+1)C}{2} = O(n)$. On retrouve ici la même complexité que la complexité dans le pire des cas. La sortie prématurée de la boucle n'a donc aucune influence sur la complexité.

III.7 Complexité amortie

Dans le cadre de l'étude des structures de données, il est fréquent de considérer non pas la complexité dans le pire des cas d'une opération mais celle d'une succession d'opérations divisée par le nombre d'opérations effectuées. Ainsi, on peut très bien avoir une opération ponctuellement plus coûteuse que les autres, mais en procédant ainsi on lisse le surcoût sur l'ensemble des opérations. On parle alors de **complexité amortie**.

■ **Remarque 2.12** Cette notion ne masque pas le fait qu'une opération puisse prendre ponctuellement plus de temps. Dans des contextes temps réel où il est importat de maitriser pleinement les complexités, il est peu judicieux d'utiliser de telles complexités. Par exemple, dans une visualisation en 3D, pour maintenir un débit constant d'images par secondes, chaque image doit prendre un temps similaire. Se reposer sur une structure de donnée ayant une faible complexité

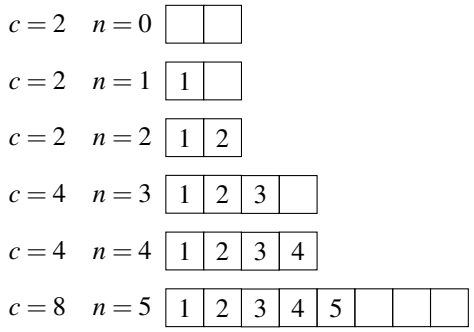
amortie mais une complexité dans le pire des cas importante, c'est risquer d'avoir des sacades avec une image qui prendrait plus de temps pour être calculée.

■

Un exemple simple est donnée par la structure de données des tableaux dynamiques. C'est la structure de données utilisées dans de nombreux langages de haut niveau pour implémenter le type abstrait des listes. La différence principale entre cette structure de données et celle des tableaux de C est qu'on peut ajouter et supprimer des éléments.

Un tableau dynamique d'entiers est un triplet `(t,c,n)` où `t` est un tableau de taille `c`, appelée la capacité du tableau dynamique, et `n` est un autre entier représentant la taille logique du tableau. A tout moment $c \geq n$. Dans `t` il y a ainsi $c - n$ cases déjà allouées qui permettent de rajouter un élément en temps constant. Quand $c = n$, on alloue une nouvelle zone mémoire, souvent de taille $2c$, on déplace le tableau `t` dans cette zone et on a donc pour cet ajout prévu un certain nombre de cases d'avance.

La figure suivante présente une succession d'ajouts :



Une implémentation de ces opérations est proposée dans le programme suivant :

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

typedef struct {
    int *t;
    size_t c;
    size_t n;
} tableau_dynamique ;

tableau_dynamique tableau_dynamique_creer()
{
    tableau_dynamique d;
    const size_t capacite_initiale = 2;

    d.t = malloc(capacite_initiale * sizeof(int));
    d.c = capacite_initiale;
    d.n = 0;

    return d;
}

void tableau_dynamique_ajout(tableau_dynamique *d, int x)
{
    if (d->c == d->n) {
        d->c = 2 * d->c;
        d->t = realloc(d->t, d->c * sizeof(int));
    }
    d->t[d->n] = x;
```

```
    d->n++;
}

void tableau_dynamique_print(tableau_dynamique d)
{
    printf("c=%d\tn=%d\t|", d.c, d.n);
    for (size_t i = 0; i < d.n; i++) {
        printf("%d|", d.t[i]);
    }
    for (size_t i = d.n; i < d.c; i++) {
        printf(" |");
    }
    printf("\n");
}

int main(void) {
    tableau_dynamique d = tableau_dynamique_creer();
    tableau_dynamique_print(d);
    for (int i = 1; i < 6; i++) {
        tableau_dynamique_ajout(&d,i);
        tableau_dynamique_print(d);
    }
}
```

Ce programme, une fois exécuté produit la sortie suivante qui permet de retrouver exactement le comportement attendu :

```
c=2 n=0 | | |
c=2 n=1 |1| |
c=2 n=2 |1|2|
c=4 n=3 |1|2|3| |
c=4 n=4 |1|2|3|4|
c=8 n=5 |1|2|3|4|5| | | |
```

Calculons la complexité amortie de l'ajout d'un élément. Si on considère un ajout d'élément qui provoque une réallocation du tableau, celle-ci sera en $O(c)$ et les $c-1$ opérations suivantes d'ajout seront en $O(1)$. La complexité globale de ces c opérations d'ajout est alors en $O(c)$, ce qui donne une complexité amortie en $O(1)$. On peut ainsi considérer que l'ajout d'un élément dans un tableau dynamique est en complexité temporelle amortie constante.

■ **Note 2.4** Il faut faire un choix entre reprendre ici une preuve plus précise ou la garder pour un chapitre ultérieur avec au moins un exemple de méthode du potentiel (Skew Heaps ?)

■

III.8 Pertinence de la complexité spatiale

Même si la complexité temporelle est le plus souvent celle qui est importante à calculer, certains algorithmes ont une complexité temporelle faible mais en contrepartie une complexité spatiale élevée. On parle alors de compromis temps-mémoire.

Un exemple classique d'un tel compromis est celui de la programmation dynamique où on passe d'une complexité temporelle exponentielle à une complexité temporelle polynomiale en stockant des valeurs intermédiaires pour ne pas les recalculer. En procédant ainsi, on passe d'une complexité spatiale constante à polynomiale.

Cela est illustré dans le programme suivant qui permet de déterminer le n-ième terme de la suite de Fibonacci, ce qui n'a pas d'intérêt informatique mais est caractéristique de récurrence que l'on résoudra par la programmation dynamique.

```
(* Fibonacci exponentiel *)
let rec fibo n =
    if n = 0
    then 0
    else if n = 1
    then 1
    else fibo (n-1) + fibo (n-2)

(* Fibonacci linéaire *)
let fibo n =
    let prec = Array.make (n+1) 0 in
    prec.(0) <- 0;
    prec.(1) <- 1;
    for i = 2 to n do
        prec.(i) <- prec.(i-1) + prec.(i-2)
    done;
    prec.(n)
```

3. Algorithmique exacte

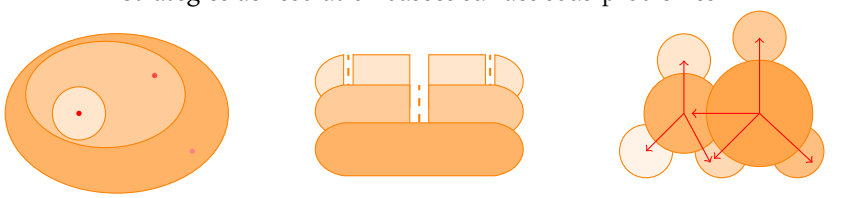
Source image : https://www.flickr.com/photos/x6e38/3440634940/

■ **Remarque 3.1** Dans ce chapitre, on étudie des problèmes pour lesquels on va exprimer des algorithmes permettant d'obtenir des solutions exactes. C'est à contraster avec le chapitre sur l'algorithmique approchée.

■

Une grande partie des stratégies de résolution est basée sur la résolution de sous-problèmes. On verra ainsi trois types de stratégies résumées dans le schéma suivant :

Stratégies de résolution basées sur des sous-problèmes



Algorithme Glouton
Optimal = Choix Glouton + Sous-problème optimal

Diviser pour Régner
Optimal = Sous-problèmes disjoints + Fusion

Programmation dynamique
Optimal = Sous-problèmes superposés + Stratégie de calcul

I Recherche par force brute

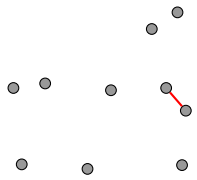
I.1 Principe

Considérons un problème du type trouver un $x \in V$ vérifiant une propriété $P(x)$. Par exemple, V est l'ensemble des chaînes de caractère et P vérifie si une chaîne est un mot de passe qu'on cherche. Dans certains problèmes, un tel x n'est pas unique et on cherche à tous les énumérer.

Une recherche par force brute, ou recherche exhaustive, consiste à parcourir l'ensemble V jusqu'à obtenir une solution. Pour la recherche du mot de passe, on pourrait commencer par énumérer les chaînes de longueur 1, puis de longueur 2, et ainsi de suite.

Le plus souvent, l'ensemble V est fini (pour les mots de passe, cela peut consister à limiter la longueur maximale du mot de passe). Ainsi, une recherche par force brute effectue $O(|V|)$ itérations.

Considérons le problème *PlusProchePaire* qui, étant donné un ensemble de n points ($n \geq 2$), détermine la paire constituée des deux points les plus proches.



Une implémentation naïve de la recherche par force brute consiste à énumérer les $\frac{n(n-1)}{2}$ paires et donc à effectuer $O(n^2)$ itérations.

```
let plus_proche_paire points =
    let n = Array.length points in
    let min_paire = ref (distance points.(0) points.(1), (0, 1)) in
    for i = 0 to n - 1 do
        for j = i+1 to n - 1 do
            let d = distance points.(i) points.(j) in
            if d < fst !min_paire
            then min_paire := (d, (i, j))
        done
    done;
    snd !min_paire
```

I.2 Raffinement : droite de balayage

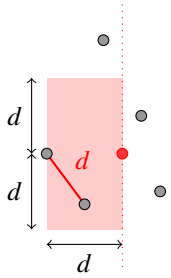
Il est parfois possible d'accélérer la recherche par force brute en ordonnant le parcours des candidats pour pouvoir éviter de tester certains d'entres eux.

En géométrie algorithmique, une approche classique consiste à ordonner les objets selon leur abscisse et à parcourir les objets par abscisse croissante. On parle alors de **droite de balayage** (en anglais, *sweep line*) car cela revient à balayer le plan par une droite verticale en ne traitant que les objets avant cette ligne.

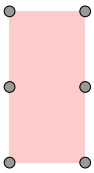
Reprenons le problème précédent, on considère que les points sont triés par abscisse croissante : $(x_0, y_0), \ldots, (x_{n-1}, y_{n-1})$. On va parcourir les points dans cet ordre en maintenant un ensemble de points à gauche du point courant, appelés *points actifs*, et en ne calculant que les intersections avec les points actifs.

Si on a parcouru les N premiers points et qu'on a obtenu que la plus petite distance était d, lorsqu'on considère le point (x_N, y_N), il est inutile de tester les points qui sont forcément à distance $> d$ de celui-ci. C'est-à-dire qu'on peut éliminer les points qui ne sont pas dans le rectangle $[x_N - d, x_N] \times [y_N - d, y_N + d]$ du test. Les points dont l'abscisse est $< x_N - d$ peuvent être éliminés définitivement vu que l'on raisonne par abscisse croissante, par contre, les points d'ordonnées invalides doivent être conservés pour les points ultérieurs.

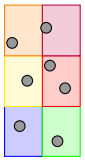
Ce rectangle est représenté sur le schéma suivant ainsi qu'une ligne imaginaire qui correspond à l'abscisse du point courant et qu'on peut imaginer parcourant le plan de gauche à droite pour traiter les points au fur et à mesure.



Afin de déterminer la complexité de cet algorithme, il est nécessaire de connaitre le nombre maximal de points dans le rectangle. Comme ces points ont été pris en compte précédemment, ils sont forcément à distance au moins d les uns des autres. Il s'agit donc de déterminer le nombre maximum de points qu'on peut placer dans ce rectangle à distance au moins d. On remarque tout d'abord qu'on peut placer six points ainsi :



Si jamais on avait au moins sept points, on peut voir qu'il y a forcément un des six sous-rectangles suivants qui contiendrait au moins deux points :



Or, ces sous-rectangles sont de longueur $\frac{1}{2}d$ et de hauteur $\frac{2}{3}d$, donc la distance maximale entre deux de leurs points correspond à la longueur des diagonales : $\sqrt{\frac{1}{4}+\frac{4}{9}}d = \frac{5}{6}d < d$.

Comme un de ces six points est le point courant, il y a toujours au plus 5 points dans l'ensemble des points actifs.

Voici le principe de l'algorithme que l'on va implémenter :

- On trie le tableau `points` par ordre croissant. **Complexité :** $O(n \log n)$
- On initialise la plus petite distance `d` courante à la distance entre les deux premiers points
- On crée un ensemble `actifs`, ordonné par les ordonnées, de points contenant initialement les deux premiers points
- Pour chaque point (x, y) en partant du deuxième :
  - On supprime les points (x', y') tels que $x' < x - d$ de `actifs`. **Complexité :** sur l'ensemble des itérations on ne pourra jamais supprimer deux fois un point, donc on effectue au maximum n suppressions chacune en $O(\log n)$ donc $O(n \log n)$.
  - On parcourt les points de `actifs` dont les ordonnées sont comprises entre $y - d$ et $y + d$. **Complexité :** pour récupérer le premier point de l'ensemble, il faut $O(\log n)$ en pire cas (tous les points actifs) et ensuite on effectue au plus 5 itérations comme on vient de le prouver.

On remarque ainsi que la complexité en temps et en pire cas de cet algorithme est de $O(n \log n)$. Ici, le fait d'avoir la structure `actifs` ordonnée par les ordonnées est crucial pour garantir la complexité. Pour la réalisation d'une structure d'ensemble ordonnée ayant ces complexité, voir le chapitre FIXME.

Ici, on utilise le module `Set` d'OCaml pour réaliser la structure d'ensemble, pour cela on commence par créer le module `PointSet` pour les ensembles de points :

```
module Point = struct
    type t = float * float
    let compare (x1,y1) (x2,y2) = Stdlib.compare y1 y2
end

module PointSet = Set.Make(Point)
```

Puis on définit une fonction permettant de parcourir les points entre deux ordonnées :

```
let set_iter_entre f set bas haut =
    try
        let e = PointSet.find_first (fun p -> snd p >= bas) set in
        let seq = PointSet.to_seq_from e set in
        let rec aux seq =
            match seq () with
            | Seq.Nil -> ()
            | Seq.Cons (p, seq_suite) ->
                    if snd p <= haut
                    then begin
                        f p;
                        aux seq_suite
                    end
        in aux seq
    with Not_found -> ()
```

On implémente alors assez directement l'algorithme décrit précédemment :

```
let plus_proche_paire_balayage points =
    let compare (x1,y1) (x2,y2) =
        if x1 = x2
        then if y1 < y2 then -1 else 1
        else if x1 < x2 then -1 else 1
    in
    Array.sort compare points;
    let n = Array.length points in
    let d = ref (distance points.(0) points.(1)) in
    let couple = ref (points.(0), points.(1)) in
    let actifs = ref (PointSet.empty
            |> PointSet.add points.(0) |> PointSet.add points.(1)) in

    let gauche = ref 0 in

    for i = 2 to n-1 do
        let xi, yi = points.(i) in

        while fst points.(!gauche) < xi -. !d do
            actifs := PointSet.remove points.(!gauche) !actifs;
            incr gauche
        done;

        set_iter_entre (fun pj ->
            let dip = distance points.(i) pj in
            if dip < !d
```

```
            then begin
                couple := (points.(i), pj);
                d := dip
            end) !actifs (yi -. !d) (yi +. !d);

        actifs := PointSet.add points.(i) !actifs
    done;
    !d
```

Problème : test d'intersection pour un ensemble de segments

Considérons le problème suivant *IntersectionEnsemble* : étant donné n segments dans le plan, il s'agit de déterminer si au moins deux des segments s'intersectent.

■ **Remarque 3.2** On peut considérer ici que l'on dispose d'une fonction

```
intersecte : (float * float) * (float * float)
    -> (float * float) * (float * float) -> bool
```

qui teste l'intersection entre deux segments.

Cependant, il est possible d'écrire une telle fonction avec un peu de géométrie élémentaire.

Si on considère que les deux segments sont $[A_1B_1]$ et $[A_2B_2]$, avec $A_1 \neq B_1$ et $A_2 \neq B_2$, alors chaque point du segment $[A_1B_1]$ est de la forme $A_1 + t\overrightarrow{A_1B_1}$ où $t \in [0,1]$. De même les points du segments $[A_2B_2]$ sont de la forme $A_2 + u\overrightarrow{A_2B_2}$ où $u \in [0,1]$.

S'il y a une intersection, c'est qu'il existe $(t,u) \in [0,1]^2$ tel que

$$A_1 + t\overrightarrow{A_1B_1} = A_2 + u\overrightarrow{A_2B_2} \iff \overrightarrow{A_2A_1} + t\overrightarrow{A_1B_1} = u\overrightarrow{A_2B_2}$$

L'idée est alors d'utiliser une opération appelée **produit vectoriel** sur les vecteurs. Comme ici, tout est plan, le produit vectoriel est uniquement déterminé par sa troisième coordonnée, celle qui sort du plan, et on peut se contenter de calculer celle-ci. On note ainsi $(x,y) \times (x',y') = xy' - yx'$ cette coordonnée. On a donc $u \times u = 0$.

On peut alors composer l'égalité par $\times\overrightarrow{A_2B_2}$:

$$\overrightarrow{A_2A_1} \times \overrightarrow{A_2B_2} + t\left(\overrightarrow{A_1B_1} \times \overrightarrow{A_2B_2}\right) = 0$$

Notons $\Delta = \overrightarrow{A_1B_1} \times \overrightarrow{A_2B_2}$, si $\Delta \neq 0$, alors

$$t = -\frac{\overrightarrow{A_2A_1} \times \overrightarrow{A_2B_2}}{\Delta} = \frac{\overrightarrow{A_1A_2} \times \overrightarrow{A_2B_2}}{\Delta}$$

On procède de même avec $\times\overrightarrow{A_1B_1}$ pour obtenir une expression de u : $\overrightarrow{A_2A_1} \times \overrightarrow{A_1B_1} = u\left(\overrightarrow{A_2B_2} \times \overrightarrow{A_1B_1}\right) = -u\Delta$ et donc

$$u = -\frac{\overrightarrow{A_2A_1} \times \overrightarrow{A_1B_1}}{\Delta} = \frac{\overrightarrow{A_1A_2} \times \overrightarrow{A_1B_1}}{\Delta}$$

Si $\Delta \neq 0$, on peut donc alors exprimer u et t et vérifier qu'ils sont dans $[0,1]$.

Si $\Delta = 0$ c'est que les deux segments sont de directions parallèles ou confondues.

— Si $\overrightarrow{A_1A_2} \times \overrightarrow{A_1B_1} \neq 0$ alors $\overrightarrow{A_1A_2}$ et $\overrightarrow{A_1B_1}$ sont non colinéaires donc les deux segments sont sur des droites parallèles distinctes et ne peuvent s'intersecter.

— Sinon, les segments reposent sur une même droite et il s'agit de vérifier leurs positions sur la droite. Pour cela, on exprime $A_2 = A_1 + t_A\overrightarrow{A_1B_1}$ de même pour $B_2 = A_1 + t_B\overrightarrow{A_1B_1}$. Plus précisement, on calcule $\overrightarrow{A_1A_2} \cdot \overrightarrow{A_1B_1} = t_A||\overrightarrow{A_1B_1}||^2$ à l'aide du produit scalaire et on a $t_A = \frac{\overrightarrow{A_1A_2}\cdot\overrightarrow{A_1B_1}}{||\overrightarrow{A_1B_1}||^2}$. De même, $t_B = \frac{\overrightarrow{A_1B_2}\cdot\overrightarrow{A_1B_1}}{||\overrightarrow{A_1B_1}||^2}$. On doit alors vérifier si l'intervalle $[t_A, t_B]$ (ou $[t_B, t_A]$ selon leur position) intersecte $[0, 1]$.

Voici une fonction *OCaml* qui correspond à ce raisonnement

```
let intersecte (a1,b1) (a2,b2) =
    let vec (x1,y1) (x2,y2) = (x2-.x1,y2-.y1) in
    let cross (x1,y1) (x2,y2) = x1 *. y2 -. y1 *. x2 in
    let dot (x1,y1) (x2,y2) = x1 *. x2 +. y1 *. y2 in
    let proche0 x = let eps = 1e-20 in
        if x < 0. then -.x < eps else x < eps in
    let a1b1 = vec a1 b1 in let a2b2 = vec a2 b2 in
    let a1a2 = vec a1 a2 in let a1b2 = vec a1 b2 in

    let delta = cross a1b1 a2b2 in

    if proche0 delta
    then
         if proche0 (cross a1a2 a1b1)
         then let na1b1 = dot a1b1 a1b1 in (* colinéaires *)
              let tA = (dot a1a2 a1b1) /. na1b1 in
              let tB = (dot a1b2 a1b1) /. na1b1 in
              if tA < tB
              then not (tB < 0. || tA > 1.)
              else not (tA < 0. || tB > 1.)
         else false (* parallèles *)
    else let t = (cross a1a2 a2b2) /. delta in (* se croisent *)
         let u = (cross a1a2 a1b1) /. delta in
         t >= 0. && t <= 1. && u >= 0. && u <= 1.
```

■

■ **Note 3.1** réécrire cela avec le déterminant de deux vecteurs du plan qui est au programme de mathématiques de seconde.

■

La recherche par force brute va alors énumérer l'ensemble des paires de segments distincts et tester deux à deux les intersections. On peut ainsi écrire le programme suivant qui est assez simple et effectuera effectivement $O(|v|^2)$ itérations dans le pire cas, i.e. lorsqu'il n'y a pas d'intersections.

```
exception Trouve

let intersection_ensemble (v: ((float * float) * (float * float)) array) : bool =
    let n = Array.length v in
    try
        for i = 0 to n - 1 do
            for j = i+1 to n-1 do
                if intersecte v.(i) v.(j)
                then raise Trouve
            done
```

```
        done;
        false
    with Trouve -> true
```

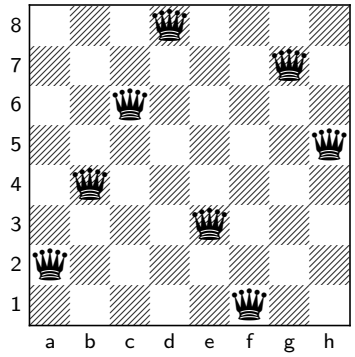
TODO approche par droite de balayage : algorithme de Shamos et Hoey (1976)

I.3 Recherche par retour sur trace (backtracking)

Dans des problèmes admettant des solutions partielles, on peut construire une solution par essai de toutes les possibilités en complétant tant qu'on a bien une solution partielle. La recherche par retour sur trace repose sur ce constat pour énumérer l'ensemble des solutions en utilisant la récursivité (d'où la notion de *retour sur trace*) pour les essais.

L'exemble classique de ce problème est celui des huit reines : étant donné un échiquier, peut-on placer huit reines de sorte qu'aucune reine ne puisse prendre une autre reine ? Plus précisément : sur un plateau de 8x8 cases, peut-on placer huit pions tels que deux pions quelconques ne soient jamais sur la même ligne ou la même diagonale ?

Exemple de solution :



Ce problème admet effectivement des solutions partielles en ne considérant que k reines à placer. Pour énumérer les solutions, on peut même se contenter de solutions partielles où les k reines sont placées sur les k premières rangées.

Voici ainsi un algorithme pour énumérer les solutions :

- Supposons que k reines aient été placées et qu'on dispose d'une solution partielle.
  - Si $k = 8$ alors toutes les reines sont placées et la solution est complète, on la comptabilise
  - Sinon, on continue la recherche pour chaque position de la $k + 1$ reine sur la $k + 1$ rangée qui préserve le fait d'être une solution partielle.

Ici, quand on dit qu'on continue la recherche, ce qu'on signifie c'est qu'on effectue un appel récursif.

Pour programmer cette méthode, on va définir une fonction récursive de signature :

```
val resout_reines : (int * int) list -> (int * int) list list
```

Un appel à `resout_reines part` va ainsi renvoyer la liste des solutions complètes construites à partir de la solution partielle `part`. Les solutions sont représentées par des listes de couples de coordonnées sur l'échiquier, donc dans $[|0; 7|]^2$

Voici une implémentation où on explore les solutions à l'aide d'une boucle impérative dans l'appel récursif. La fonction `valide` permet de tester si le placement d'une reine est possible avant d'effectuer un appel.

```
let rec valide (x1,y1) l =
    match l with
```

```
    | [] -> true
    | (x2,y2)::q ->
        x1 <> x2 && abs (x2-x1) <> abs(y2-y1) && valide (x1,y1) q

let rec resout_reines part =
    let k = List.length part in
    if k = 8
    then [ part ]
    else begin
        let resultats = ref [] in
        for x = 0 to 7 do
            let essai = (x,k) :: part in
            if valide (x,k) part
            then begin
                resultats := (resout_reines essai) @ !resultats;
            end
        done;
        !resultats
    end
```

et, ici, une autre implémentation purement récursive à l'aide d'une fonction récursive.

```
let rec resout_reines part =
    let k = List.length part in
    if k = 8
    then [ part ]
    else
        let rec aux x acc =
            if x < 0
            then acc
            else let essai = (x,k) :: part in
                 let nacc = if valide (x,k) part
                            then (resout_reines essai) @ acc
                            else acc in
                    aux (x-1) nacc
        in
        aux 7 []
```

Une partie de l'arbre de recherche est présenté sur l'image suivante :

L'arbre complet comporte 2057 noeuds dont 92 feuilles correspondant aux solutions du problème. A titre de comparaison, l'arbre exhaustif correspondant à faire tous les choix de placement à raison d'une reine par ligne compterait $8^8 = 16777216$ noeuds. On voit bien que le backtracking est plus économe en exploration.

Problème : résolution de Sudoku

La recherche par retour sur trace se prête très bien à la résolution de problèmes comme le Sudoku. On va ici tout simplement tenter de remplir chaque case du haut vers le bas tant qu'on satisfait les contraintes du Sudoku. Le programme sera ainsi très proche de la résolution des huit reines.

Commençons par rappeler le principe du Sudoku :

— On part d'une grille de 81 cases réparties en une grille de 3x3 sous-grilles de 3x3 cases et

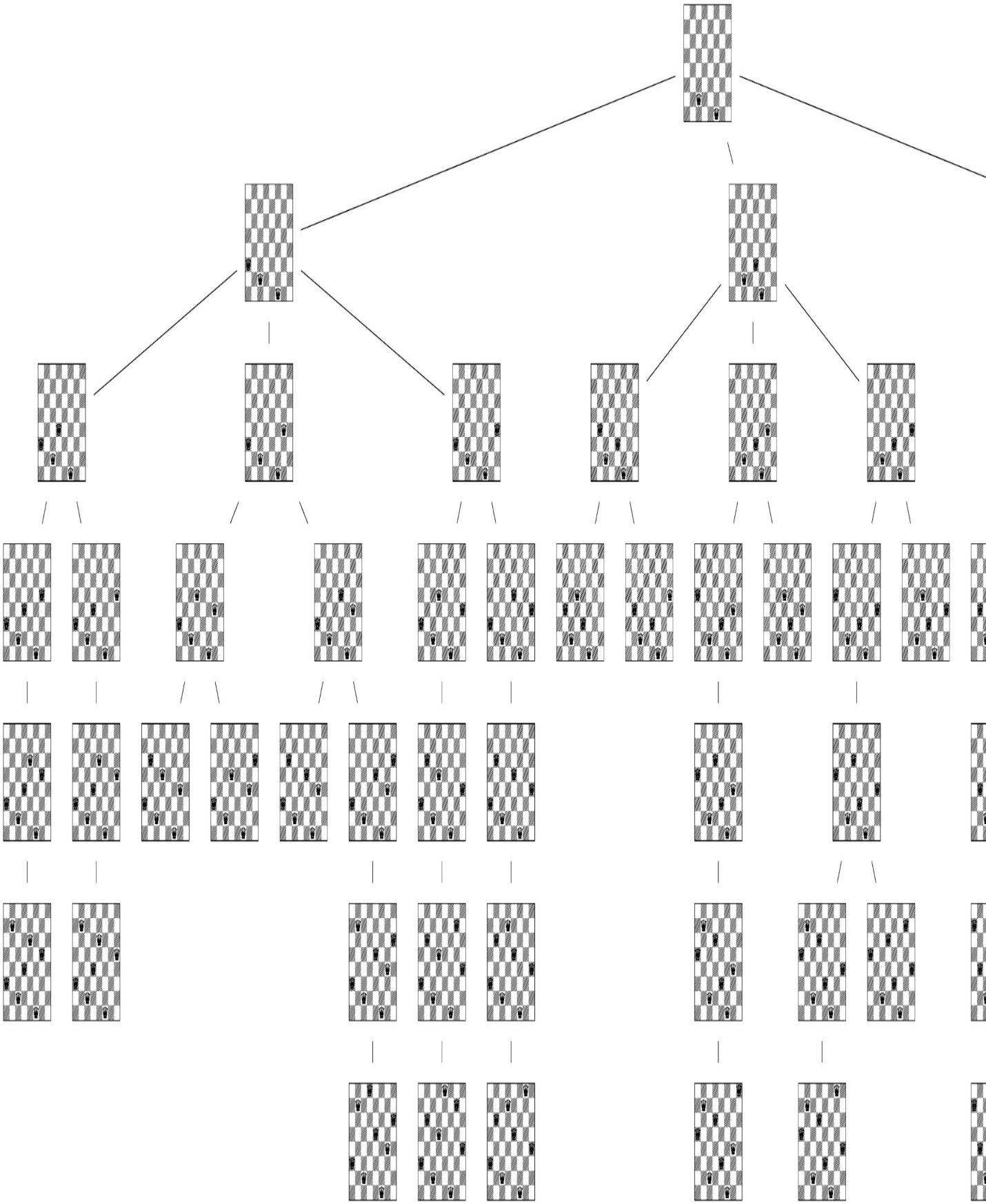


FIGURE 3.1 – Arbre de recherche pour les huit reines

1								6
		6		2		7		
7	8	9	4	5		1		3
			8		7			4
				3				
	9				4	2		1
3	1	2	9	7			4	
	4			1	2		7	8
9		8						

comportant des chiffres de 1 à 9 dans certaines cases.

— L'objectif est de remplir chaque case avec un chiffre de 1 à 9 de sorte que chaque ligne, chaque colonne et chaque sous-grille 3x3 comporte une et une seule fois chaque chiffre.

— Un sudoku admet une unique solution.

Pour représenter une grille de Sudoku en `OCaml` on utilise un `(int option) array array`, la valeur `None` signifiant que la case est vide et la valeur `Some x` qu'elle est remplie avec la valeur x.

```
type grille = (int option) array array
```

On fait le choix de représenté la grille par un tableau de lignes, ce qui signiie que pour accèder à la case de coordonnée (x, y) dans `g` il faut écrire `g.(y).(x)`.

Le problème donnée précédemment est alors représenté par la valeur suivante :

```
let probleme = [|
    [| Some 1; None; None;    None; None; None;   None; None; Some 6 |];
    [| None; None; Some 6;    None; Some 2; None;   Some 7; None; None |];
    [| Some 7; Some 8; Some 9;    Some 4; Some 5; None;   Some 1; None; Some 3 |];

    [| None; None; None;    Some 8; None; Some 7;   None; None; Some 4 |];
    [| None; None; None;    None; Some 3; None;   None; None; None |];
    [| None; Some 9; None;    None; None; Some 4;   Some 2; None; Some 1 |];

    [| Some 3; Some 1; Some 2;    Some 9; Some 7; None;   None; Some 4; None |];
    [| None; Some 4; None;    None; Some 1; Some 2;   None; Some 7; Some 8 |];
    [| Some 9; None; Some 8;    None; None; None;   None; None; None |];
    |]
```

Afin de définir la fonction de résolution, on définit une première fonction `suivant` de signature :

```
val suivant : grille -> (int * int) -> (int * int) option
```

telle que l'appel à `suivant g (x,y)` renvoie `Some (xi,yi)` quand (x_i, y_i) sont les coordonnées de la prochaine case libre, dans l'ordre gauche à droite puis haut vers bas, après (x, y) ou `None` quand il n'existe pas de telle case libre. Cela signifie alors que la grille est entièrement remplie.

```
let rec suivant g (x,y) =
    if y > 8
    then None
```

```
    else if g.(y).(x) = None
    then Some (x,y)
    else if x < 8 then suivant g (x+1, y)
    else suivant g (0, y+1)
```

On définit également une fonction `valide` de signature

```
val valide : grille -> int -> int -> bool
```

telle que l'appel à `valide g x y` renvoie `true` si et seulement si la valeur placée en coordonnée (x, y) n'invalide pas la grille. Ne pas prendre cette valeur en paramètre permettant d'écrire un peu plus simplement cette fonction. La fonction est assez directe, étant donné (x, y) on va parcourir sa ligne, sa colonne et sa sous-grille pour vérifier qu'un nombre n'a pas été placé deux fois à l'aide d'un tableau de drapeaux :

```
let valide g x y =
    let v = ref true in
    let vus_colonne = Array.make 9 false in
    for y0 = 0 to 8 do
        match g.(y0).(x) with
        | None -> ()
        | Some k ->
                if vus_colonne.(k-1)
                then v := false;
                vus_colonne.(k-1) <- true
    done;
    let vus_ligne = Array.make 9 false in
    for x0 = 0 to 8 do
        match g.(y).(x0) with
        | None -> ()
        | Some k ->
                if vus_ligne.(k-1)
                then v := false;
                vus_ligne.(k-1) <- true
    done;
    let vus_grille = Array.make 9 false in
    let xb = (x / 3) * 3 in
    let yb = (y / 3) * 3 in
    for xd = 0 to 2 do
        for yd = 0 to 2 do
            match g.(yb+yd).(xb+xd) with
            | None -> ()
            | Some k ->
                    if vus_grille.(k-1)
                    then v := false;
                    vus_grille.(k-1) <- true
        done
    done;
    !v
```

On peut alors définir la fonction `resout` qui va résoudre le Sudoku en effectuant tous les remplissages tant qu'on a une grille valide. Dès qu'une solution est trouvé, on s'arrête. Pour cela, on utilise le mécanisme des exceptions pour permettre une sortie prématurée. On a fait le choix de travailler en place dans la grille, ainsi à la fin de l'exécution de la fonction, la grille correspond à la solution.

```
exception Solution

let resout g =
    let rec aux xi yi = match suivant g (xi, yi) with
        | None -> raise Solution
        | Some (x,y) ->
            for i = 1 to 9 do
                g.(y).(x) <- Some i;
                if valide g x y
                then begin
                    aux x y
                end
            done;
            g.(y).(x) <- None
    in
    try
        aux 0 0
    with Solution -> ()
```

II Algorithmes gloutons

II.1 Principe

On considère ici un problème d'énumération comme dans la section précédente muni d'une fonction d'objectifs qui attribue une valeur numérique aux solutions et aux solutions partielles.

Soit $f : P \to \mathbb{R}$ une telle fonction, où $S \cup P$ est l'ensemble des solutions du problème d'énumération et P l'ensemble des solutions partielles, on se pose maintenant le problème de l'optimalité vis-à-vis de f : déterminer $x \in S$ tel que $f(x) = \max_{y \in S} f(y)$ on note souvent $x = \text{argmax}_{y \in S} f(y)$. On parle alors de problème d'optimisation combinatoire.

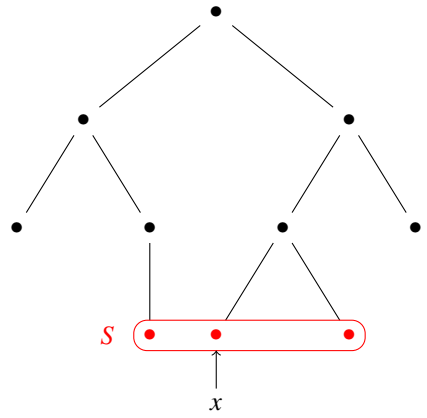
■ **Remarque 3.3** — En considérant $g : y \mapsto -f(y)$, on transforme un problème de maximisation en un problème de minimisation.

— Il y a une ambiguïté sur $\text{argmax}_{y \in S} f(y)$ quand plusieurs éléments de S réalisent ce maximum. Dans la plupart des algorithmes gloutons qu'on va considérer, on commence par donner un ordre sur S et on considère le plus petit y pour cet ordre réalisant le maximum. L'ordre choisi est alors crucial dans la preuve de correction. C'est aussi une des raisons pour lesquelles les algorithmes gloutons sont souvent de complexité temporelle $O(n \log_2 n)$.

■

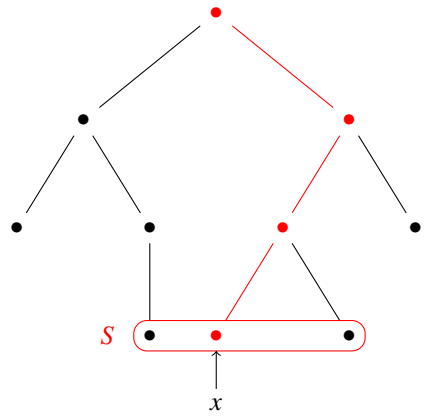
Une première stratégie très élémentaire consiste alors à énumérer S, de manière exhaustive ou avec une stratégie plus fine comme le retour sur trace, puis à déterminer un élément maximal de manière directe.

Cela revient donc à déterminer l'arbre des solutions puis à trouver une feuille maximisant l'objectif :

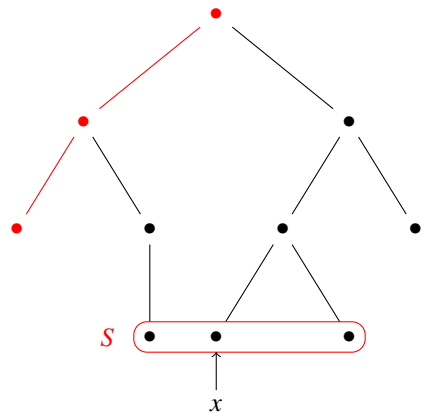


Un algorithme glouton va suivre une approche beaucoup plus efficace : à chaque étape de construction de la solution, on choisit la branche qui maximise la fonction d'objectif. C'est-à-dire que si en partant d'une solution partielle $x \in P$ il est possible de l'étendre en d'autres solutions partielles $p_x = \{y_1, ..., y_n\}$, on va choisir $y = \text{argmax}_{t \in p_x} f(t)$ la solution qui maximise localement f.

Sur l'arbre précédent, cela reviendrait à n'emprunter qu'une seule branche :



Cela a l'air très efficace mais il y a un problème majeur : il n'y a aucune garantie qu'on aboutisse ainsi à une solution, encore moins à une solution optimale. En effet, on aurait très bien pu faire les choix suivants :



et ne pas aboutir à une solution.

Considérons par exemple le problème du **rendu de monnaie** : étant donné, une liste de valeurs faciales de pièces $P = (v_1, \ldots, v_p) \in (\mathbb{N}^*)^p$ avec $1 = v_1 < \cdots < v_p$ et une somme $n \in \mathbb{N}^*$, on cherche la manière d'exprimer cette somme avec le plus petit nombre de pièces possible.

Plus précisément, l'ensemble des solutions $S = \{(k_1, \ldots, k_p) \in N^p \mid k_1 v_1 + \cdots + k_p v_p = n\}$

et la fonction d'objectif est $f : (k_1, \dots, k_p) \mapsto k_1 + \cdots + k_p$. Les solutions partielles ici sont les réalisations de valeur $< n$. On cherche alors $x = argmin_{y \in S} f(y)$.

Comme $1 = v_1$, $S \neq \emptyset$ car $(n, 0, \dots, 0) \in S$ et ainsi $f(x) \leq n$.

L'algorithme glouton va utiliser la plus grande pièce possible à chaque étape puis on applique l'algorithme glouton sur la somme restante sauf si elle est nulle, ce qui constitue la condition d'arrêt.

Exemple 1

- $P = (1, 2, 5, 10)$
- $n = 14$
- On utilise la plus grande pièce possible $10 \leq 14$ puis on exprime $4 = 14 - 10$
- Ici, la plus grande pièce est 2 et on continue avec $2 = 4 - 2$
- La plus grande pièce est encore 2 et on s'arrête car $0 = 2 - 2$.
- En conclusion, on a obtenu $x = (0, 2, 0, 1)$.
- Une exploration exhuastive permet de s'assurer qu'on a effectivement obtenu une décomposition minimale. En effet, ici l'ensemble des décompositions est : { (14,0,0,0), (12,1,0,0), (8,3,0,0), (6,4,0,0), (4,5,0,0), (2,6,0,0), (0,7,0,0), (9,0,1,0), (7,1,1,0), (5,2,1,0), (3,3,1,0), (1,4,1,0), (4,0,2,0), (2,1,2,0), (0,2,2,0), (4,0,0,1), (2,1,0,1), (0,2,0,1) }.

Exemple 2

- $P = (1, 2, 7, 10)$
- $n = 14$
- L'algorithme glouton va ici procéder comme dans l'exemple 1 et on va obtenir $x = (0, 2, 0, 1)$.
- Mais on remarque que ce n'est pas un minimum car $x' = (0, 0, 2, 0)$ convient avec $f(x') = 2 < 3 = f(x)$.

Conclusion l'algorithme glouton n'a effectivement pas de raisons d'être optimal.

On peut se poser la question des algorithmes pour lesquels l'algorithme glouton aboutit nécessairement à une solution optimale.

■ **Note 3.2** TODO - Ajouter un paragraphe simple sur les matroïdes qui puisse se décliner sous la forme d'un problème.

■

II.2 Algorithme d'Huffman - Compression

On va étudier ici un principe de compression parfaite (sans perte d'information à la décompression) de données appelé l'algorithme de Huffman et qui repose sur ce principe simple : coder sur moins de bits les caractères les plus fréquents.

Par exemple si on considère le mot `abaabc`, en le codant avec un nombre de bits fixes, par exemple 2 avec le code `a=00,b=01,c=10`, on aurait besoin de 12 bits pour représenter le mot. Mais si on choisit le code suivant : `a=0,b=10,c=11`, il suffit de 9 bits. On a donc gagné 3 bits soit un facteur de compression de 75%.

On remarque que pour pouvoir décompreser, il n'aurait pas été possible de faire commencer le code de `b` ou `c` par un `0`, sinon on aurait eu une ambiguité avec la lecture d'un `a`. On parle alors de code préfixe :

Définition II.1 Soit $X \subset \{0, 1\}^*$, on dit que X est un code préfixe lorsque pour tous $x, y \in X$, x n'est pas un préfixe de y et y n'est pas un préfixe de x.

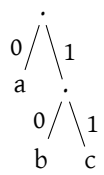
On se pose alors la question du code préfixe optimal pour un texte donné.

Plus précisément, étant donné un alphabet fini Σ et une application $f : \Sigma \to [0, 1]$ associant à chaque lettre sa fréquence dans le texte considéré, c'est-à-dire $f(x) = \frac{\text{occurences}(x)}{\text{longueur}}$ et donc $\sum_{x \in \Sigma} f(x) = 1$, on cherche un code préfixe X et une application $c : \Sigma \to X$ telle que $\sum_{x \in \Sigma} f(x)|c(x)|$ soit minimale.

En effet, si le texte considéré est de longueur n, il y a exactement $f(x)n$ occurences de x dans le texte et $f(x)n|c(x)|$ bits après codage. La longueur du texte codé est donc $n\sum_{x\in\Sigma} f(x)|c(x)|$.

L'application de codage c peut être représenté par un arbre binaire où les arêtes gauches correspondent à 0, les arêtes droites à 1 et les feuilles aux éléments de Σ dont les étiquettes des chemins y menant depuis la racine de l'arbre correspondent à leur image par c.

Par exemple, pour le code $a = 0, b = 10, c = 11$ on aurait l'arbre :



Avec un tel arbre, il est très simple de décoder le texte codé car il suffit de suivre un chemin dans l'arbre jusqu'à tomber sur une feuille, produire la lettre correspondante, puis repartir de la racine de l'arbre. La longueur du code associé à une lettre est alors égale à la profondeur de la feuille correspondante. L'optimalité du codage préfixe est ainsi équivalente à la minimalité de l'arbre vis-à-vis de la fonction d'objectif $\varphi(t) = \sum_{x\in\Sigma} f(x)p(t,x)$ où $p(t,x)$ est la profondeur de la feuille d'étiquette x dans l'arbre t ou 0 si x n'est pas une des étiquettes, cet extension permettant d'étendre la fonction d'objectif aux solutions partielles.

L'algorithme d'Huffman va construire un arbre correspondant à un codage optimal à l'aide d'une file de priorité d'arbres. On étend pour cela l'application f à de tels arbres en définissant que si t est un arbre de feuilles $x_1, \ldots, x_n$ alors $f(t) = f(x_1) + \cdots + f(x_n)$.

- Au départ, on place dans la file des arbres réduits à une feuille pour chaque élément $x \in \Sigma$ et dont la priorité est $f(x)$.
- Tant que la file contient au moins deux éléments
  - on retire les deux plus petits éléments x et y de la file de priorité $f(x)$ et $f(y)$
  - on ajoute un arbre $z = Noeud(x, y)$ de priorité $f(z) = f(x) + f(y)$.
- On renvoie l'unique élément restant dans la file.

L'implémentation de cet algorithme est alors assez directe avec une file de priorité.

■ **Note 3.3** FIXME : utiliser un module file de priorité fait dans un chapitre précédent plutôt ■

```
module Heap = Core_kernel.Heap

let compare_pair f a b = Stdlib.compare (f a) (f b)

type arbre = Feuille of char | Noeud of arbre * arbre

let huffman freq =
    let arbres = Heap.create ~cmp:(compare_pair fst) () in
    let compte = ref (Array.length freq) in
    for i = 0 to Array.length freq - 1 do
        let c, f = freq.(i) in
        Heap.add arbres (f, Feuille c)
    done;
    while !compte > 1 do
        let fx, x = Heap.pop_exn arbres in
        let fy, y = Heap.pop_exn arbres in
        Heap.add arbres (fx+.fy, Noeud(x,y));
        compte := !compte - 1
    done;
    snd (Heap.pop_exn arbres)
```

■ **Remarque 3.4** FIXME : Rajouter la compression/décompression

■

L'algorithme de Huffman est un algorithme glouton car si on considère pour solution partielle la fôret présente dans la file et pour objectif la fonction φ étendue aux fôrets en sommant la valeur de φ sur chaque arbre, alors fusionner dans la fôret F deux arbres x et y en la transformant en une fôret F' va avoir l'impact suivant sur la fonction d'objectif :

$$\varphi(F') = \varphi(F) + f(x) + f(y)$$

car, en effet, on va rajouter 1 à la profondeur de chaque feuille et donc on passe pour la contribution de x de $\varphi(x) = \sum_{c\in\Sigma} f(c)p(x,c)$ à $\sum_{c\in\Sigma} f(c)(p(x,c)+1) = \varphi(x)+\sum_{c\in\Sigma} f(c) = \varphi(x) + f(x)$.

On remarque ainsi que la fusion qui minimise localement φ est celle qui fusionne les deux arbres de plus petite valeur pour f.

Pour montrer que l'algorithme glouton produit ici un codage minimal, on va utiliser une technique classique qui consiste à montrer qu'étant donné une solution optimale, on peut toujours la transformer sans augmenter sa valeur pour obtenir, de proche en proche, la solution renvoyée par le glouton.

Théorème II.1 Supposons que les lettres les moins fréquentes soient a et b, il existe un arbre optimal dont les deux feuilles étiquettées par a et b descendent du même noeud et sont de profondeur maximale.

Démonstration. Considérons un arbre optimal t et soient c l'étiquette d'une feuille de profondeur maximale. On remarque qu'elle a forcément une feuille sœur car sinon, on pourrait omettre le noeud et l'arbre obtenu serait de plus petite valeur par φ.

FIXME : dessin

Soit d l'étiquette de cette feuille sœur. Sans perte de généralités, on suppose que $f(c) \leq f(d)$ et $f(a) \leq f(b)$. Comme a a la plus petite fréquence, on a $f(a) \leq f(c)$ et comme b est la deuxième, on a $f(b) \leq f(d)$. De plus, $p(t,a) \geq p(t,c)$ et $p(t,b) \geq p(t,d)$.

Si on échange les étiquettes a et c, seule les termes associées à ces lettres changent dans l'évaluation de φ. Si on note t' le nouvel arbre obtenu après cet échange, on a

$$\varphi(t') = \varphi(t) - f(a)p(t,a) - f(c)p(t,c) + f(a)p(t,c) + f(c)p(t,a)$$

Or, $f(c) \geq f(a)$ et $p(t,a) \geq p(t,c)$ donc

$$\varphi(t') = \varphi(t) + (f(c) - f(a))(p(t,a) - p(t,c)) \leq \varphi(t)$$

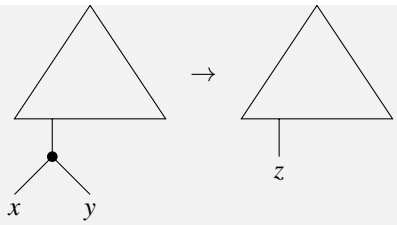
L'échange préserve le caractère optimal. En fait, ici, on a nécessairement une égalité pour ne pas aboutir à une contradiction, donc soit les feuilles étaient à même profondeur, soit les lettres avaient la même fréquence.

Comme on a les mêmes relations entre b et d, on peut effectuer le même argument et échanger les étiquettes en préservant le caractère optimal.

■

Le théorème suivant permet de raisonner par récurrence en diminuant le nombre de lettres.

Théorème II.2 Soit t un arbre ayant x et y comme feuilles soeurs et t' l'arbre obtenu en remplaçant le noeud liant x et y par une feuille étiquettée par z où z est une nouvelle lettre telle que $f(z) = f(x) + f(y)$.



On a alors $\varphi(t) = \varphi(t') + f(z)$.

Démonstration. Seule les termes portant sur x, y et z sont influencés par le changement et on a :

$$\begin{aligned}\varphi(t) &= \varphi(t') + f(x)p(t,x) + f(y)p(t,y) - f(z)p(t',z)\\ &= \varphi(t') + f(z)(p(t',z)+1) - f(z)p(t',z)\\ &= \varphi(t') + f(z)\end{aligned}$$

■

Théorème II.3 L'algorithme de Huffman renvoie un arbre optimal.

Démonstration. Par récurrence sur $|\Sigma|$.

Initialisation : si Σ ne contient qu'une lettre, il n'y a qu'un arbre qui est nécessairement optimal.

Hérédité : si la propriété est vraie pour un alphabet de $n-1 \geq 1$ lettres, alors soit Σ contenant n lettres et x et y les deux lettres les moins fréquentes.

On pose Σ' obtenue en remplaçant x et y par une nouvelle lettre z et on suppose que $f(z) = f(x) + f(y)$. L'hypothèse de récurrence assure qu'on obtient un arbre optimal t' en appliquant l'algorithme d'Huffman sur Σ'. Comme la première étape d'Huffman va fusionner les feuilles x et y, on sait que l'arbre t obtenu en partant de Σ se déduit de t' en remplaçant z par $Noeud(x,y)$. Le théorème précédent assure alors que $\varphi(t) = \varphi(t') + f(z)$.

Soit t_o un arbre optimal pour Σ dans lequel x et y sont soeurs, possible en vertu du premier théorème, et soit t'_o l'arbre obtenue en remplaçant dans t_o le noeud liant x et y par une feuille étiquettée par z. On a ici encore $\varphi(t_o) = \varphi(t'_o) + f(z) \geq \varphi(t') + f(z) \geq \varphi(t)$ car t' est optimal.

Ainsi, on a bien l'égalité $\varphi(t_o) = \varphi(t)$ et t est optimal.

■

II.3 Preuve d'optimalité

Dans le paragraphe précédent, on retrouve un schéma de preuve classique pour les preuves d'optimalité des algorithmes gloutons :

- Montrer qu'à partir d'une solution optimale, il est possible de déterminer une solution optimale ayant fait le même choix que l'algorithme glouton. Pour Huffman c'était le fait d'avoir un arbre optimal ayant les deux lettres les moins fréquentes comme sœurs à profondeur maximale.
- Montrer qu'une solution optimale se comportant comme le résultat de l'algorithme glouton à une étape ne peut être meilleure que le résultat de l'algorithme glouton.

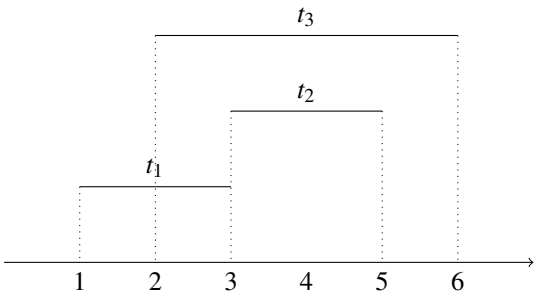
II.4 Sélection d'activités

Description

Étant donné un ensemble d'activités données par leur temps de début et leur temps de fin (on considère les temps comme des entiers pour simplifier), on se pose la question du nombre

maximal d'activité que l'on puisse sélectionner sans que deux activités soient en conflits. Cela correspond par exemple à l'organisation du planning d'un employé.

On dit que deux activités (d_1, f_1) et (d_2, f_2) sont en conflits quand $[d_1, f_1[\cap [d_2, f_2[\neq \emptyset$.



Ici, t_1 et t_2 sont en conflits avec t_3. Mais t_1 et t_2 ne sont pas en conflit. On considère que deux activités peuvent se succéder directement : $f_1 = d_2$.

On considère donc en entrée de ce problème une suite finie $((d_1, f_1), \ldots, (d_n, f_n))$ et on cherche un sous-ensemble $I \subset [\![1, n]\!]$ de plus grand cardinal tel que pour tous $i, j \in I$, si $i \neq j$ alors (d_i, f_i) et (d_j, f_j) ne sont pas en conflits. On dit que I est un **ensemble indépendant**.

Algorithme glouton et implémentation

Pour résoudre ce problème, on considère l'algorithme glouton associé à la fonction d'objectif cardinal et **en triant les activités** ordre croissant de temps de fin.

Cet algorithme est implémenté dans le programme suivant :

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

typedef struct {
    unsigned int id;
    unsigned int debut;
    unsigned int fin;
    unsigned char selectionnee;
} activite;

int compare_activites(const void *t1, const void *t2)
{
    return ((activite *)t1)->fin - ((activite *)t2)->fin;
}

void selectionne(activite *activites, size_t nb_activites)
{
    size_t derniere_activite = 0;
    /* on commmence par trier en O(n log2 n) les activites
     * selon le temps de fin */
    qsort(activites, nb_activites,
        sizeof(activite), compare_activites);
    activites[0].selectionnee = 1;
    for (size_t i = 1; i < nb_activites; i++)
    {
        if (activites[i].debut >= activites[derniere_activite].fin)
        {
            activites[i].selectionnee = 1;
            derniere_activite = i;
        }
    }
```

```
}

int main()
{
    activite activites[] = {
        { 0, 1, 3, 0 }, { 1, 3, 4, 0 }, { 2, 2, 5, 0 },
        { 3, 5, 9, 0 }, { 4, 11, 12, 0 }, { 5, 8, 10, 0 },
        { 6, 0, 7, 0 }
    };

    size_t nb_activites = sizeof(activites) / sizeof(activite);

    selectionne(activites, nb_activites);

    for (size_t i = 0; i < nb_activites; i++)
    {
        printf("Activité %d (%d,%d) : %d\n",
                activites[i].id, activites[i].debut,
                activites[i].fin, activites[i].selectionnee);
    }

    return 0;
}
```

Ce programme produit alors la sortie :

```
Activité 0 (1,3) : 1
Activité 1 (3,4) : 1
Activité 2 (2,5) : 0
Activité 6 (0,7) : 0
Activité 3 (5,9) : 1
Activité 5 (8,10) : 0
Activité 4 (11,12) : 1
```

■ **Remarque 3.5** Comme l'algorithme commence par effectuer un tri, on a rajouté dans la structure `activite` un champ permettant d'identifier une activité autrement que par son indice.

■

Preuve d'optimalité

On va prouver que l'algorithme glouton renvoie un ensemble indépendant optimal. Le fait que l'ensemble soit indépendant étant direct, on se concentre sur la preuve d'optimalité en présentant un schéma de preuve qui correspond à celui identifié dans le paragraphe précédent.

Théorème II.4 Si $a_1, \ldots, a_n$ sont des activités énumérées dans l'ordre croissant de leur temps de fin, alors il existe un ensemble indépendant optimal contenant a_1.

■ **Remarque 3.6** Cela signifie qu'il fait le même choix que l'algorithme glouton à la première étape.

■

Démonstration. Soit I un ensemble indépendant optimal ne contenant pas $a_1 = (d_1, f_1)$ (sinon c'est direct). Si $a_k = (d_k, f_k)$ est l'activité de plus petit indice dans I, alors $f_k \geq f_1$ donc pour tout $a_i = (d_i, f_i)$ dans $I' = I \backslash \{a_k\}$ on a $d_i \geq f_k \geq f_1$ et ainsi a_1 et a_i ne sont pas en conflit. Ainsi $I' \cup \{a_1\}$ est un ensemble indépendant contenant a_1 de même cardinal que I donc optimal. ■

Théorème II.5 Soit $A = \{a_1, \ldots, a_n\}$ des activités ordonnées par ordre croissant de temps de fin et I un ensemble indépendant optimal contenant $a_1 = (d_1, f_1)$ (ce qui est possible selon le théorème précédent).
$I' = I \backslash \{a_1\}$ est optimal pour $A' = \{\ (d, f) \in A \mid d \geq f_1\ \}$.

Démonstration. Si, par l'absurde, I' est pas optimal pour A' alors $J \subset A'$ est un ensemble indépendant de cardinal strictement plus grand que celui de I'. Or, $A' \cup \{a_1\}$ est indépendant pour l'ensemble des activités et est de cardinal strictement plus grand que I. Contradiction.

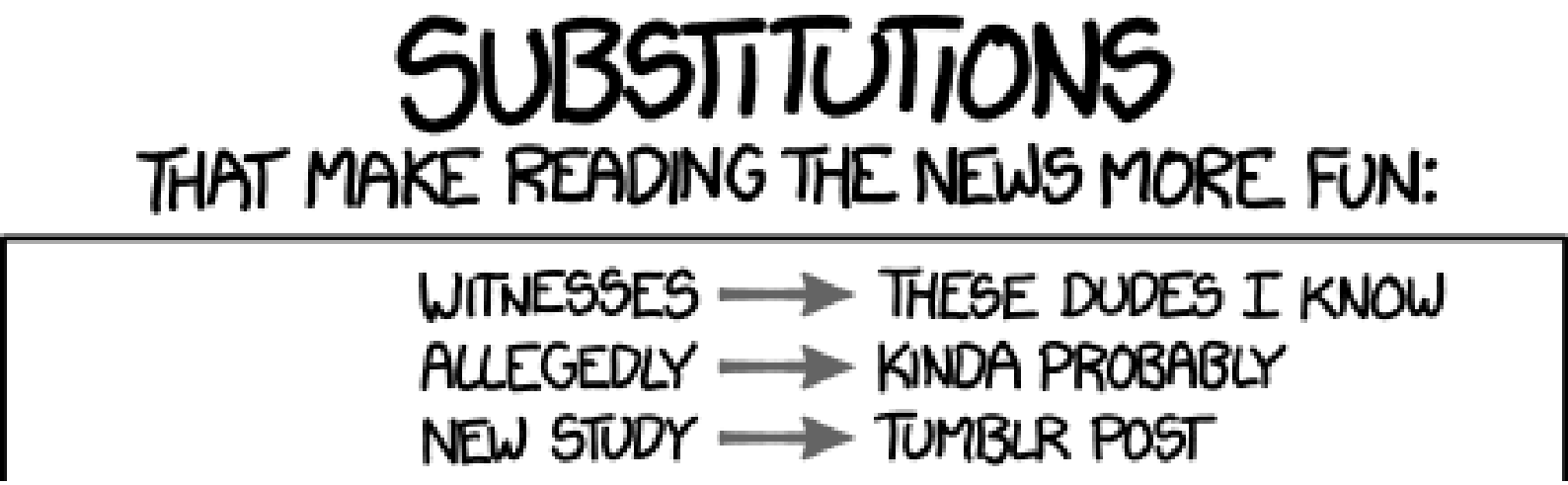
■

Théorème II.6 L'algorithme glouton renvoie un ensemble indépendant optimal.

Démonstration. Par récurrence forte sur le nombre d'activités.

- Initialisation : Pour une activité a_1, le glouton renvoie $\{a_1\}$ qui est directement optimal.
- Hérédité : Si la propriété est vérifiée pour $k \leq n$ activités, soit $A = \{a_1, \ldots, a_n\}$ des activités ordonnées par temps de fin. Soit I un ensemble indépendant optimal contenant a_1 et $I' = I \cap \{a_1\}$. Le théorème précédent assure que I' est optimal sur $A' = \{\ (d, f) \in \mid d \geq f_1\ \}$.
  Par hypothèse de récurrence, l'algorithme glouton sur A' produit un ensemble indépendant optimal G', donc tel que $|G'| = |I'|$. Par construction l'algorithme glouton sur A renvoie $G = G' \cup \{a_1\}$ de même cardinal que I, donc optimal.

■



4. Algorithmique du texte

Source image : https://xkcd.com/1288/

■ **Note 4.1** version très partielle uniquement pour présenter Rabin-Karp

■

I Algorithme de Rabin-Karp

I.1 Principe

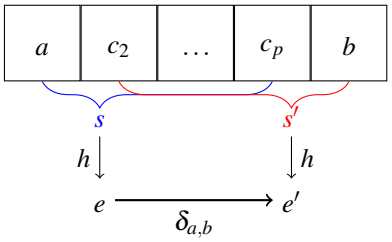
L'algorithme de Rabin-Karp est un algorithme de recherche d'un motif dans un texte qui utilise une notion d'empreinte pour déterminer, en temps constant, si il est probable que la position actuelle corresponde à une occurrence du motif.

Pour cela, si on cherche un motif de longueur p sur l'alphabet Σ, on considère une **fonction de hachage** $h : \Sigma^p \to X$. Les éléments de l'ensemble X sont appelés des empreintes et on suppose que l'égalité entre deux empreintes se vérifie en temps constant contrairement à l'égalité dans Σ^p qui se vérifie en $O(p)$ dans le pire des cas. Le plus souvent, on choisit pour X un type entier machine.

■ **Note 4.2** Sûrement mettre ici des renvois vers la partie portant le plus sur la notion de fonction de hachage pour la définition la plus complète.

■

Bien qu'il soit normalement aussi coûteux de calculer l'image par h d'une sous-chaîne de longueur p que de tester l'égalité entre cette sous-chaîne et le motif, le point essentiel de l'algorithme de Rabin-Karp est d'utiliser une fonction de hachage permettant un calcul incrémental en temps constant :



Ici, on considère donc, pour $a, b \in \Sigma$, une fonction de mise à jour $\delta_{a,b} : X \to X$ telle que pour tout $c_2, \ldots, c_p \in \Sigma$ on ait $\delta_{a,b}(h(ac_2 \ldots c_p)) = h(c_2 \ldots c_p b)$.

L'algorithme de Rabin-Karp procède alors ainsi pour chercher m de longueur p dans la chaîne $s = c_0 \ldots c_{n-1}$ où $n \geq p$:

- calcul de $e_m = h(m)$ et $e = h(c_0..c_{p-1})$.
- Pour i allant de 0 à $n - p$:
  - Si $e_m = e$, on renvoie un succès pour la recherche à la position i si $m = c_i \ldots c_{i+p-1}$
  - si $i < n - p$ on met à jour l'empreinte $e = \delta_{c_i, c_{i+p}}(e)$.

La complexité temporelle liée à la gestion des empreintes est donc en $O(n + p) = O(n)$ car $n \geq p$. Par contre, pour calculer la complexité liée à la recherche $m = c_i \ldots c_{i+p-1}$, il est nécessaire d'estimer la proportion de faux positifs, c'est-à-dire de positions i telles que $e_m = e$ mais $m \neq c_i \ldots c_{i+p-1}$. On va voir dans la partie suivante qu'on peut supposer qu'elle est négligeable, ce qui permet de considérer que l'algorithme de Rabin-Karp est linéaire.

I.2 Choix d'une fonction de hachage

Réaliser une bonne fonction de hachage est une question très complexe qui dépasse le cadre du cours d'informatique de MPI. Cependant, il est possible de réaliser ici une fonction de hachage répondant aux contraintes de Rabin-Karp assez facilement.

Pour cela, on considère que les caractères sont des entiers compris entre 0 et 255, ce qui correspond au type des caractères non signés sur un octet. On peut alors identifier une chaîne de longueur p avec un nombre entre 0 et $r^p - 1$ où $r = 2^8$, on note ainsi

$$P(c_0 \ldots c_{p-1}) = \sum_{i=0}^{p-1} c_i r^{p-1-i} = c_0 r^{p-1} + c_1 r^{p-2} + \cdots + c_{p-1}$$

On considère de plus un entier premier q et on pose $h(s) = P(s) \mod q$ c'est-à-dire le reste de $P(s)$ dans la division euclidienne par q. On peut ainsi définir $\delta_{a,b}(e) = (r(e - ar^{p-1}) + b) \mod q$.

Si on précalcule $r^{p-1} \mod q$ il suffit d'un nombre d'opération constant, et indépendant de p, pour calculer la nouvelle empreinte à l'aide de $\delta_{a,b}$.

Le point essentiel est alors de déterminer un nombre premier q tel qu'il soit peu probable d'obtenir des faux positifs. Une analyse mathématique permet d'affirmer que chaque élément de $[|0; q-1|]$ a de l'ordre de $\frac{r^p}{q}$ antécédents par h. Ainsi, si on choisit deux chaînes aléatoirement dans Σ^p, il y aura collision avec probabilité proche de $\frac{1}{q}$. En considérant q proche de la taille maximale pour le type entier considéré, on minimise donc cette probabilité.

■ **Remarque 4.1** On peut également s'intéresser à des nombres q pour lesquels le modulo soit rapide à calculer. Un exemple classique est $q = 2^{31} - 1$ car on peut déduire la division euclidienne de a par q de l'écriture de a en base 2^{31}. En effet, si $a = \sum_{k=0}^{n} a_k 2^{31k}$ comme $2^{31} - 1 | 2^{31k} - 1$ pour $k \geq 1$, on a $2^{31k} \equiv 1\ [q]$ et ainsi $a \equiv \sum_{k=0}^{n} a_k [q]$. On remarque que $a_k = (a >> 31k) \& 2^{31}$, on a alors soit $a_k < q$ et alors $a_k \mod q = a_k$, soit $a_k = q$ et $a_k \mod q = 0$. Il suffit donc de faire un masquage pour obtenir directement $a_k \mod q = (a >> 31k) \& q$.

On obtient alors le programme suivant :

```
let rec fastmod a =
    let s = ref 0 in
    let x = ref a in
    let q = 0x7fffffff in
    while !x > 0 do
        s := !s + (!x land q);
        x := !x lsr 31
    done;
    if !s > q
    then fastmod !s
    else if !s = q
    then 0
    else !s
```

■

Le programme suivant présente deux manières de calculer naïvement h et $\delta_{a,b}$:

```
let hash r q s =
    let p = ref 1 in
    let e = ref 0 in
    for i = String.length s - 1 downto 0 do
        e := (!p * (Char.code s.[i]) + !e) mod q;
        p := (r * !p) mod q
    done;
    !e

let delta r q rp a b e = (* rp est r^(p-1) mod q *)
    (r * (e - rp * (Char.code a)) + Char.code b) mod q
```

I.3 Implémentation

Une implémentation directe de l'algorithme de Rabin-Karp est donnée dans le programme qui suit. On se sert ici du caractère paresseux du && pour n'effecuter le test coûteux d'égalité des chaînes qu'en cas d'égalité des empreintes.

```
exception Trouve of int

let rabin_karp m s =
    let n = String.length s in
    let p = String.length m in
    let r = 256 in
    let q = 0x7fffffff in (* 2^(31)-1 *)
    let rp = pow r (p-1) q in
    let me = hash r q m in
    let e = ref (hash r q (String.sub s 0 p)) in
    try
        for i = 0 to n-p+1 do
            if me = !e && m = String.sub s i p
            then raise (Trouve i);
            if i+p < n then e := delta r q rp s.[i] s.[i+p] !e
        done; None
    with Trouve k -> Some k
```

Si on suppose qu'il est improbable d'obtenir un faux positif, il est possible de renvoyer un succès dès que les empreintes sont égales. L'avantage d'une telle version est alors d'être un algorithme sans retour sur les données. C'est-à-dire qu'il n'est pas nécessaire de garder en mémoire ou de réaccéder à un caractère.

I.4 L'algorithme originel de Rabin et Karp

Si on regarde l'article originel de Rabin et Karp décrivant cette méthode, on peut être étonné du fait que la méthode précédemment décrite était considérée comme déjà connue dans la littérature par les auteurs. En fait, ce qu'ils décrivent et annoncent comme étant novateur est l'utilisation d'un algorithme probabiliste en choisissant aléatoirement une fonction de hachage à chaque lancement de l'algorithme. En pratique, il s'agit de choisir aléatoirement un nombre premier q parmi un ensemble précalculé de nombres premiers.

L'algorithme que l'on vient de décrire a un pire cas qui est très improbable car on considère que la probabilité d'un faux positif est à peu près de $1/q$, donc moins de 5.10^{-10} pour $q = 2^{31} - 1$. Le problème ici est la notion de probabilité sur les entrées : est-on certain que l'algorithme recevra une entrée choisie uniformément ? Rabin et Karp parlent d'un *adversaire intelligent* qui aurait connaissance de la fonction de hachage choisie pour produire des entrées en pire cas. On pourrait ainsi imaginer une *attaque* sur serveur effectuant une recherche avec Rabin-Karp suite à l'entrée d'un utilisateur. Un adversaire pourrait construire une entrée en pire cas et tenter de surcharger le serveur en l'effectuant de manière répétée.

Pour bien mettre en lumière ce phénomène, nous allons ici construire, dans un cas très simple de fonction de hachage, une telle chaîne problématique. Pour cela, considérons la fonction de hachage précédemment décrite dans le cas de motif de taille 2, avec Σ contenant les lettres de `a` à `z`, $r = 26$ et $q = 17$. On considère une recherche du motif `aa` dont l'empreinte est `0`, la même que celle des chaînes `ar` et `ra`. On peut donc considérer la chaîne `arar...ar` qui produira un faux positif à chaque étape.

■ **Remarque 4.2** Détail des calculs. Ici on associe à `a` la valeur 0, ..., à `z` la valeur 25. On a donc

$$h(aa) = (0 \times 26 + 0) \mod 17 = 0$$

$$h(ar) = (0 \times 26 + 17) \mod 17 = 0$$

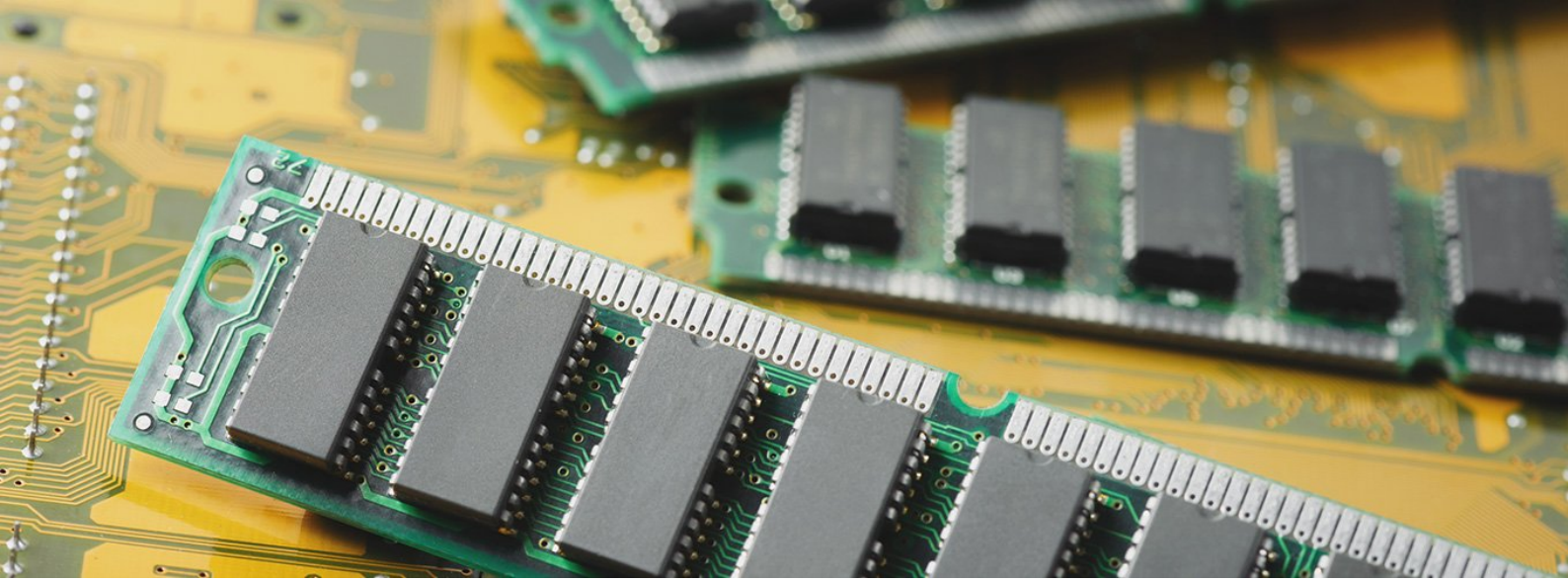
$$h(ra) = (17 \times 26 + 0) \mod 17 = 0$$

L'empreinte reste ainsi nulle tout au long de l'algorithme de Rabin-Karp et on a un faux positif à chaque itération.

■

III

Systèmes



5. Gestion de la mémoire dans un programme compilé

■ **Remarque 5.1** Ce chapitre se concentre sur la manière dont un programme compilé gère la mémoire. Il est question, en particulier, de la notion de variable. Le modèle dans lequel on se place est celui du langage C où une variable est un emplacement mémoire.

■

I Organisation de la mémoire

■ **Note 5.1** Ici, je fais le choix d'une présentation assez informelle pour ne pas qu'elle soit trop liée à la réalité d'un compilateur en particulier.

■

Un programme compilé gère la mémoire d'un ordinateur de deux manières très différentes

— statiquement : c'est le cas des variables locales ou globales définies dans le programme. Au moment de la compilation, le compilateur dispose de l'information suffisante pour prévoir de la place en mémoire pour stocker ces données.
— dynamiquement : c'est le cas des objets dont la taille n'est connue qu'à l'exécution et peut varier selon l'état du programme. C'est alors au moment de l'exécution que le programme va faire une demande d'allocation pour obtenir une place mémoire.

En terme d'allocation statique, on peut distinguer plusieurs types de mémoire :

- les variables globales initialisées qui seront stockées dans le binaire et placées en mémoire dans une zone spécifique chargée avec le binaire
- les variables globales non initialisées dont seule la déclaration sera dans le binaire et qui seront allouées, placées en mémoire et initialisées à 0 au moment du chargement du binaire
- les variables locales et les paramètres qui sont placés dans une pile afin de les allouer uniquement au moment de l'exécution du bloc ou de la fonction

L'allocation dynamique utilise une zone mémoire appelée tas dont une possible organisation est développée dans la partie sec. V.4.

Structure de la mémoire associée à un programme

Considérons le programme c suivant.

```c
const int a  = 42;
int b[] = { 1, 2, 3 };
int c;

int f(int x, int y)
{
    int z = x;
    z = z * y;
    return z;
}

int main(int argc, char **argv)
{
    const int d = 1664;

    c = f(a, d);

    return 0;
}
```

Il est possible d'observer la manière dont sa mémoire sera répartie en utilisant la commande `objdump` :

```
$ gcc -c memoire.c
$ objdump -x memoire.o

memoire.o:     file format elf64-x86-64
memoire.o
architecture: i386:x86-64, flags 0x00000011:
HAS_RELOC, HAS_SYMS
start address 0x0000000000000000

Sections:
Idx Name          Size      VMA               LMA               File off  Algn
  0 .text         00000053  0000000000000000  0000000000000000  00000040  2**0
                  CONTENTS, ALLOC, LOAD, RELOC, READONLY, CODE
  1 .data         00000000  0000000000000000  0000000000000000  00000093  2**0
                  CONTENTS, ALLOC, LOAD, DATA
```

```
  2 .bss          00000004  0000000000000000  0000000000000000  00000094  2**2
                  ALLOC
  3 .rodata       00000014  0000000000000000  0000000000000000  00000098  2**3
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  4 .comment      00000013  0000000000000000  0000000000000000  000000ac  2**0
                  CONTENTS, READONLY
  5 .note.GNU-stack 00000000  0000000000000000  0000000000000000  000000bf  2**0
                  CONTENTS, READONLY
  6 .eh_frame     00000058  0000000000000000  0000000000000000  000000c0  2**3
                  CONTENTS, ALLOC, LOAD, RELOC, READONLY, DATA
SYMBOL TABLE:
0000000000000000 l    df *ABS*  0000000000000000 orga.c
0000000000000000 l    d  .text  0000000000000000 .text
0000000000000000 l    d  .data  0000000000000000 .data
0000000000000000 l    d  .bss   0000000000000000 .bss
0000000000000000 l    d  .rodata    0000000000000000 .rodata
0000000000000000 l    d  .note.GNU-stack    0000000000000000 .note.GNU-stack
0000000000000000 l    d  .eh_frame  0000000000000000 .eh_frame
0000000000000000 l    d  .comment   0000000000000000 .comment
0000000000000000 g     O .rodata    0000000000000004 a
0000000000000000 g     O .data  000000000000000c b
0000000000000000 g     O .bss   0000000000000004 c
0000000000000000 g     F .text  000000000000001f f
000000000000001f g     F .text  0000000000000034 main


RELOCATION RECORDS FOR [.text]:
OFFSET           TYPE              VALUE
0000000000000042 R_X86_64_PLT32    f-0x0000000000000004
0000000000000048 R_X86_64_PC32     c-0x0000000000000004


RELOCATION RECORDS FOR [.eh_frame]:
OFFSET           TYPE              VALUE
0000000000000020 R_X86_64_PC32     .text
0000000000000040 R_X86_64_PC32     .text+0x000000000000001f
```

On retrouve dans les sections mémoire :

- `.text` contenant le programme binaire
- `.data` contenant les variables globales initialisées et non constantes
- `.bss` contenant les variables non initialisées
- `.rodata` contenant les variables globales initialisées mais constantes.

Ici, le schéma mémoire est un peu plus compliqué car une zone mémoire distincte est prévue pour les variables constantes initialisées pour des raisons de sécurité.

La pile et le tas sont automatiques et n'ont pas besoin de figurer dans le binaire, c'est pour cela qu'on ne les trouve pas dans la liste.

Dans cette description mémoire, on trouve la table des symboles qui décrit où vont se trouver en mémoire certaines variables.

Nom de variable	Sorte de déclaration	Section mémoire
a	constante globale initialisée	rodata
b	globale initialisée	data
c	globale non init.	bss

La section suivante permettra de compléter le tableau en étudiant comment les paramètres et variables locales sont gérés. On remarque cependant qu'il n y a pas de symboles pour ceux-ci. En effet, le nom des variables locales est a priori perdu après la compilation contrairement aux variables globales.

II Portée d'un identificateur

En ce qui concerne une variable dans un programme, on peut définir deux notions d'apparence assez similaire.

D'une part la portée d'un identificateur qui correspond au texte du programme :

Définition II.1 La **portée** d'un identificateur est définie par la zone du texte d'un programme dans laquelle il est possible d'y faire référence sans erreurs.

■ **Remarque 5.2** Dans le langage C, un identificateur peut être utilisé dès sa déclaration mais tant que la variable n'est pas initialisée, le comportement n'est pas spécifié et il faut considérer cela comme une erreur. Le compilateur produit ainsi un avertissement quand on utilise le paramètre `-Wall`.

Dans le cas d'une définition, il est ainsi possible de faire référence à l'identificateur dans l'expression de son initialisation : `int x = x`. Ce cas est pathologique et le fait qu'on compte la ligne de déclaration dans la portée ne devrait pas inciter à écrire ce genre de code qui produira, de toutes façons, une erreur avec les options `-Wall -Werror`.

■

Dans le programme :

```
1  int a = 1;
2
3  int f (int x)
4  {
5      int y = x + a;
6      return y;
7  }
8
9  int g()
10 {
11     int z = 3;
12     return z + f(z);
13 }
```

La portée des identificateurs est :

Identificateur	Portée
a	1-13
x	3-7

Identificateur	Portée
y	5-7
f	4-13
g	10-13
z	11-13

Pour une fonction, afin de pouvoir écrire des fonctions récursives, l'identificateur est utilisable dans le corps de la fonction.

Comme la portée est une notion associée aux identificateurs, elle est indépendante de la notion de variables. Si on considère le programme suivant :

```
1  int f()
2  {
3      int i = 3;
4
5      return i;
6  }
7
8  int g()
9  {
10     int i = 5;
11
12     return i+1;
13 }
```

L'identificateur `i` a pour portée les lignes 3-6 et 10-13.

Un autre phénomène plus complexe peut se produire quand on redéfinit un identificateur dans sa portée.

Considérons le programme suivant

```
1  int f()
2  {
3      int i = 3;
4
5      for (int j = 0; j < 3; j++)
6      {
7          int i = 4;
8
9          i += j;
10     }
11
12     return i;
13 }
```

Ici, l'identificateur `i` a pour portée les lignes 3-13 mais dans les lignes 7-10 il y a un phénomène dit de masquage où la première définition est cachée par la seconde.

L'identificateur associé à une variable globale a pour portée l'ensemble des lignes suivant sa déclaration.

■ **Remarque 5.3** En C, la portée d'un identificateur est **statique** : elle dépend uniquement du texte du programme au moment de la compilation.

En Python, la portée d'un identificateur est **dynamique** : elle peut dépendre de l'exécution d'un programme. Par exemple si on considère le programme Python

```
1 if condition:
2     x = 3
```

La portée de l'identificateur `x` dépend ici du fait que la `condition` soit réalisée ou non.

■

III Durée de vie d'une variable

Définition III.1 La **durée de vie** d'une variable correspond à la période de son exécution durant laquelle la variable est présente en mémoire.

Sauf indication contraire, la durée de vie d'une variable locale en C est définie par la portée de l'identificateur qui lui est associé : la place est réservée au début de sa portée et libérée à la fin.

La durée de vie d'une variable globale est l'intégralité du programme. En effet, comme on a pu le voir dans la section sec. I, il est possible de définir en C des variables locales dont la durée de vie dépasse sa portée. On parle alors de variables *statiques*. Ce point étant assez technique, il sera ignoré ici.

IV Piles d'exécution, variables locales et paramètres

On a vu qu'en raison de leur durée de vie, les variables globales étaient allouées dès le chargement du programme. Pour les variables locales ainsi que la mécanique des appels, on utilise une **pile**.

Cette pile d'exécution est représentée en mémoire par un tableau et un indicateur de fond de pile.

Le remplissage de ce tableau s'effectue souvent des adresses hautes vers les adresses faibles : on empile en faisant diminuer les adresses.

■ **Remarque 5.4** En fait, il existe des architectures où les adresses sont croissantes. Ce qui importe est que le tas et la pile aient des comportements opposées pour qu'ils puissent grandir dans la même zone mémoire.

■

Un compilateur peut faire le choix d'utiliser directement des registres processeurs pour les variables locales ou pour passer des paramètres à une fonction. Ici, pour simplifier, on va supposer que ce n'est pas le cas et que tout passe par la pile d'exécution.

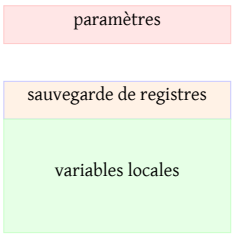
■ **Remarque 5.5** Afin de pouvoir appeler une fonction dans une bibliothèque potentiellement compilée avec un autre compilateur, il est nécessaire d'avoir une convention d'appels de fonctions. Une telle convention est appelée une *interface applicative binaire* (Application Binary Interface).

La convention `System V AMD64 ABI` qui est celle de Linux et macOS sur des architectures 64bits consiste à utiliser des registres entiers pour les six premiers arguments entiers ou pointeurs et des registres flottants pour les huit premiers arguments flottants. Les arguments suivants sont alors passés sur la pile (donc dès le septième entier/pointeur ou neuvième flottant).

La convention `cdecl` qui est assez répandue sur les architectures 32bits consiste à utiliser la pile. Par contre la valeur de retour est présente dans des registres comme pour la convention `System V AMD64 ABI`.

■

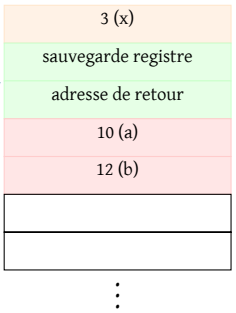
Lors d'un appel d'une fonction passant par la pile, on commence par empiler les paramètres (souvent de la droite vers la gauche) puis on empile l'adresse à laquelle doit revenir l'exécution une fois que la fonction aura terminé son exécution.
Au début de l'exécution de cette fonction, on place sur la pile l'adresse du fond de pile et on déplace celui-ci pour réserver de la place pour les variables locales.
L'empreinte sur la pile d'un appel de fonction est appelée une structure pile (**stack frame** en anglais). La pile est alors organisée, depuis l'appel au point d'entrée du programme, par empilement et dépilement de structures piles.



Structure pile associée à un appel de fonction

Voici un exemple possible de l'état de la pile d'exécution lors de l'exécution d'un programme compilé avec la norme `cdecl` (Il suffit d'ajouter l'argument `-m32` pour compiler en 32bits).

```
1
2  int f(int a, int b)
3  {
4      int c = 3;
5      /* pile ici après l'appel en l13 */
6      c = c + b;
7      c = c * a;
8      return c;
9  }
10
11 int main()
12 {
13     int x = f(10,12);
14 }
```



A chaque appel de fonction, on va donc empiler une structure de pile complète, puis la dépiler à la sortie. Ce mécanisme est essentiel pour permettre la récurrence car il permet d'effectuer plusieurs appels d'une même fonction sans risquer que la mémoire utilisée lors d'un des appels interfère avec un autre. On comprend également les limites de la récursivité ici car cet empilement successif de structures de piles peut dépasser la taille maximale de la pile d'exécution : on parle alors de *dépassement de pile* ou **stack overflow** en anglais.

■ **Remarque 5.6** Il est possible de définir des variables locales qui soient situées au même emplacement mémoire pour tous les appels d'une fonction, c'est ce qu'on appelé des variables *statiques* dans le paragraphe précédent.

Ce mécanisme est essentiellement géré comme les variables globales et il ne sera pas développé dans la suite.

■

■ **Note 5.2** Je me demande s'il faudrait parler plus précisement de la manière dont la pile est gérée avec les registres ebp/esp. Mais ça ne me semble pas apporter grand chose ici.

■

V Allocation dynamique

Comme cela a été vu dans la section sec. I, il est possible d'allouer dynamiquement de la mémoire. Pour gérer cette allocation dynamique, on passe par une zone mémoire appelé le *tas* ainsi que par un mécanisme d'allocation et de libération de mémoire au niveau du système.

Pour l'utilisateur, cette gestion interne est transparente et on peut se contenter de considérer qu'il y a deux mécanismes :

— l'**allocation** mémoire où on demande à ce qu'une zone mémoire d'une certaine taille soit allouée
— la **libération** mémoire où on signale que la zone mémoire peut être récupérée par le système.

Naturellement, la mémoire d'un ordinateur étant finie, il est très important de libérer au plus tôt la mémoire non utilisée pour éviter d'épuiser la mémoire. Quand un programme ne libère pas toute la mémoire qu'il alloue, on parle de **fuite mémoire**. L'empreinte mémoire d'un tel programme peut alors croître jusqu'à rendre le programme ou le système inutilisable.

V.1 Allocation

Pour allouer une zone mémoire, on utilise la fonction `malloc` dans `stdlib.h` de signature :

```
void *malloc(size_t size)
```

Ici `size` indique le nombre d'**octets** à allouer et la fonction renvoie un pointeur vers le premier octet alloué. Comme la fonction ne connait pas le type d'objets alloués, on utilise ainsi le type `void *`.

Ce type joue un rôle spécial et on peut changer directement le type de la valeur de retour sans rien avoir à écrire d'autre l'appel à malloc :

```
char *t = malloc(n);
```

■ **Remarque 5.7** Dans le langage C++ qui peut être vu comme un successeur de C, il est obligatoire de préciser ici le nouveau type à l'aide d'un un **transtypage**. Pour convertir la valeur `x` vers le type `char *` on écrit alors `(char *)x`. Ainsi, pour allouer un tableau de `n` caractères, on utilisera :

```
char *t = (char *) malloc(n);
```

Bien que ce ne soit pas nécessaire en C, il est fréquent de rencontrer des programmes présentant de tels transtypages qui sont superflus mais corrects syntaxiquement en C.

■

Pour obtenir la taille à allouer, il peut être utile d'utiliser l'opérateur `sizeof` qui prend en entrée un type ou une valeur et renvoie sa taille. Ainsi si on peut allouer un tableau de n entiers non signés ainsi :

```
unsigned int *t = malloc( sizeof(unsigned int) * n );
```

et cet appel ne dépend de la taille prise par un `unsigned int` sur l'architecture.

Une autre raison de l'utilisation de `sizeof` est l'extensibilité. Par exemple, si on a un `struct point` représentant des points en 2D :

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

on peut allouer un tableau de n points ainsi :

```
struct point *t = malloc( sizeof(struct point) * n );
```

Si jamais on change la structure pour représenter des points en 3D ainsi :

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

il sera inutile de changer le code d'allocation du tableau car `sizeof(point)` tiendra compte automatiquement du changement.

Si jamais une erreur empêche d'allouer la mémoire - ce qui peut être le cas s'il n'y a plus de mémoire disponible - le pointeur renvoyé par `malloc` a la valeur spéciale `NULL`.

■ **Remarque 5.8** La zone mémoire renvoyée par `malloc` n'est pas initialisée. On ne peut pas supposer qu'elle soit remplie de la valeur 0. Il faut donc manuellement initialiser la mémoire après le retour de `malloc`.

■

V.2 Libération

Pour libérer la mémoire, on utilise la fonction `free` également présente dans `stdlib.h` et dont la signature est :

```
void free(void *ptr);
```

■ **Remarque 5.9** Il est très important d'utiliser uniquement un pointeur obtenu précédemment par un appel à `malloc` et de ne pas l'utiliser plus d'une fois.

Le programme suivant provoque une erreur `free(): invalid pointer` à l'exécution mais est détécté par un avertissement du compilateur : `warning: attempt to free a non-heap object 'a'`.

```
#include <stdlib.h>

int main()
{
    int a;
    free(&a);
    return 0;
}
```

Le programme suivant alloue un tableau de deux `char` et appelle `free` sur l'adresse de la seconde case. En faisant cela, il n'y a pas d'avertissement car on appelle `free` sur un objet qui est effectivement sur le tas. On obtient alors à nouveau une erreur à l'exécution `free(): invalid pointer`.

```
#include <stdlib.h>

int main()
{
    char *a = malloc(2);
    free(&(a[1]));
    return 0;
}
```

Le programme suivant libère deux fois la mémoire et provoque l'erreur `free(): double free detected in tcache 2` à l'exécution.

```c
#include <stdlib.h>

int main()
{
    char *a = malloc(2);
    free(a);
    free(a);
    return 0;
}
```

■

V.3 Protection mémoire

Comme on l'a vu dans la première partie, quand un programme s'exécute il a un environnement mémoire constitué de plusieurs zones, parfois appelées **segments**, avec le droit d'écriture dans certaines d'entre elles.

Le système d'exploitation protège ainsi la mémoire et, au niveau matériel, l'unité de gestion de la mémoire connait les adresses accessibles à un programme. En cas d'accès anormal, le matériel provoque une erreur qui remonte au système d'exploitation qui termine l'exécution du programme avec une erreur souvent intitulée `Segmentation fault`.

Voici quelques exemples commentés produisant des erreurs de type `segmentation fault` à l'exécution.

Lecture à l'adresse 0, ce qui provoque toujours une erreur.
Même problème avec une adresse inaccessible ou invalide.

```c
int main()
{
    int *a = 0;

    return a[0];
}
```

Écriture dans une zone en lecture seule comme le segment du code.

```c
int main()
{
    int *a = (int*) (&main);
    // a pointe sur le corps de la fonction ma

    *a = 0;

    return 0;
}
```

V.4 Réalisation d'un système d'allocation de mémoire

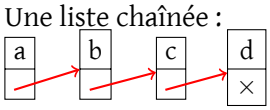
■ **Note 5.3** Prérequis : listes chaînées

■

Afin de comprendre comment fonctionne le tas, et en particulier malloc et free, on va simuler ici ce comportement en allouant une grande plage de mémoire avec malloc et en gérant le découpage et l'allocation de celle-ci.

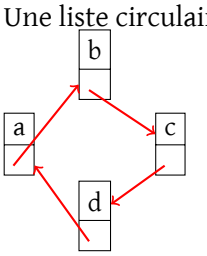
Pour gérer les blocs mémoires libres, on utilise une liste circulaire. Une liste circulaire est une liste chaînée avec un lien supplémentaire entre le premier et le dernier maillon, ce qui fait qu'on peut considérer n'importe quel maillon comme étant la *tête* de la liste.

Une liste chaînée :



Ici le dernier maillon comprend un pointeur qui ne pointe sur rien indiqué par ×, en pratique il a la valeur `NULL`

Une liste circulaire :



Le seul changement est donc de faire pointer le dernier maillon sur le premier. Le fait d'avoir preserver les valeurs dans les maillons permet ici de voir ce qu'est devenu le premier maillon, mais on peut accéder à cette liste par n'importe lequel de ces maillons.

Les noeuds de la liste circulaire de blocs libres auront pour valeur un triplet `(adresse,taille,libre)` qui indique qu'à l'adresse `adresse` il y a un bloc de `taille` octets et le booléen `libre` indique si ce bloc a été alloué ou non.

Pour cela on commence par définir une structure `bloc` et une fonctiond de création d'un bloc :

```
struct bloc {
    void *adresse;
    uint32_t taille;
    bool libre;
    struct bloc *suivant;
};

struct bloc *cree_bloc(void *adresse, uint32_t taille, bool libre)
{
    struct bloc *b = malloc(sizeof(struct bloc));
    b->adresse = adresse;
    b->taille = taille;
    b->libre = libre;
    return b;
}
```

On définit ensuite deux variables globales :

— `bloc_libres` qui va pointer sur un maillon de la liste circulaire des blocs

— `plage_memoire` qui pointe sur l'adresse de la plage mémoire que l'on va gérer et servira à la libérer en sortie de programme.

```
struct bloc *blocs_libres;
void *plage_memoire;
```

La fonction `creation_blocs_libres` permet de créer la liste circulaire avec un premier bloc qui pointe sur lui-même et qui correspond à l'adresse que l'on va placer dans `plage_memoire`.

```
void creation_blocs_libres(uint32_t taille_bloc_initial)
{
    plage_memoire = malloc(taille_bloc_initial);
    blocs_libres = cree_bloc(plage_memoire, taille_bloc_initial, true);
    blocs_libres->suivant = blocs_libres; // boucle initiale
}
```

Pour libérer la liste à la sortie du programme, on définit la fonction `destruction_blocs_libres` qui présente ainsi le parcours usuel d'une liste circulaire : on procède comme pour une liste chaînée classique mais, au lieu d'utiliser un test `bloc_courant->suivant == NULL` pour l'arrêt, il faut se souvenir du premier bloc et tester pour voir si on est revenu au point de départ. On n'oublie pas de libérer l'espace `plage_memoire` à la fin.

```
void destruction_blocs_libres()
{
    struct bloc *premier_bloc = blocs_libres;
    struct bloc *bloc_courant = premier_bloc;

    // on boucle pour libérer chaque maillon
    while (true) {
        struct bloc *bloc_suivant = bloc_courant->suivant;
        free(bloc_courant);
        if (bloc_suivant == premier_bloc) return;
        bloc_courant = bloc_suivant;
    }

    // on libère la plage mémoire initiale
    free(plage_memoire);
}
```

Pour allouer t octets, on parcourt la liste des blocs jusqu'à trouver un bloc b libre de taille $b.t$ telle que $b.t \geq t$. Si un tel bloc n'existe pas, on renvoie le pointeur `NULL` signe d'un échec d'allocation. Sinon, on indique que le bloc est occupé, on va renvoyer l'adresse du bloc obtenu mais, si $b.t > t$ on insère après b un nouveau bloc libre de taille $b.t - t$. Dans tous les cas, on fait pointer la liste des blocs libres vers le bloc qui suit b, qui est peut-être le bloc nouvellement créé et a de grandes chances d'être libre.

Ce mécanisme est implementé dans la fonction `allocation` :

```
void *allocation(uint32_t taille)
{
    struct bloc *premier_bloc = blocs_libres;
    struct bloc *bloc_courant = premier_bloc;

    while (!bloc_courant->libre || bloc_courant->taille < taille)
    {
        bloc_courant = bloc_courant->suivant;
        if (bloc_courant == premier_bloc)
        {
            // Retour au point de départ : échec d'allocation
            return NULL;
        }
    }

    // bloc_courant pointe sur un bloc libre de bonne taille

    void *adresse = bloc_courant->adresse;
    bloc_courant->libre = false;
    if (bloc_courant->taille > taille) {
        // on le sépare en deux pour récupérer la place
        struct bloc *bloc_libre = cree_bloc(adresse+taille,
                bloc_courant->taille-taille, true);
        bloc_courant->taille = taille;
```

```
        bloc_libre->suivant = bloc_courant->suivant;
        bloc_courant->suivant = bloc_libre;
    }

    // On pointe sur le bloc suivant qui est sûrement libre
    blocs_libres = bloc_courant->suivant;

    return adresse;
}
```

Pour libérer un bloc, on parcourt la liste jusqu'à trouver le bloc correspondant à l'adresse à libérer et on indique que le bloc est libre. Ici, il y a deux `assert` permettant de s'assurer que l'adresse est bien celle d'un bloc et que le bloc n'a pas déjà été libéré.

```
void liberation(void *adresse)
{
    struct bloc *premier_bloc = blocs_libres;
    struct bloc *bloc_courant = premier_bloc;

    while (bloc_courant->adresse != adresse)
    {
        bloc_courant = bloc_courant->suivant;
        // adresse invalide
        assert(bloc_courant != premier_bloc);
    }

    // pas de double libération
    assert(!bloc_courant->libre);

    // on libère l'adresse
    bloc_courant->libre = true;
}
```

Voici un premier programme de test de ces fonctions qui alloue 10 octets puis effectue plusieurs allocations. L'allocation de `c` échoue car il n'y a plus de place libre.

```
1  int main()
2  {
3      creation_blocs_libres(10);
4
5      uint8_t *a = allocation(5);
6      uint8_t *b = allocation(3);
7      uint8_t *c = allocation(3);
8      liberation(b);
9      uint8_t *d = allocation(2);
10
11     printf("Allocation a:%p b:%p c:%p d:%p\n",
12             (void *)a, (void *)b, (void *)c, (void *)d);
13
14     destruction_blocs_libres();
15
16     return 0;
17 }
```

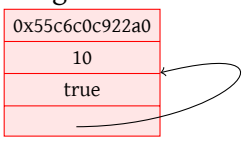
Ce programme affiche alors

```
Allocation a:0x55c6c0c922a0 b:0x55c6c0c922a5 c:(nil) d:0x55c6c0c922a5
```

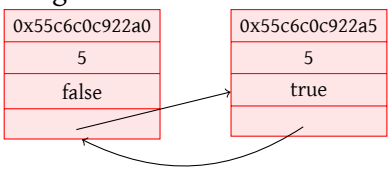
Voici l'évolution de la liste circulaire en présentant les maillons sous la forme :

L'évolution de la liste des blocs entre les lignes 3 et 9 est alors :

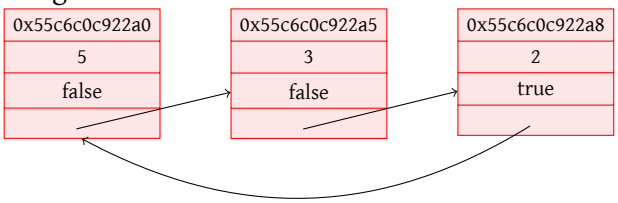
— Ligne 3



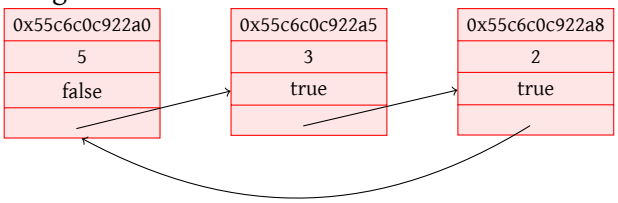
— Ligne 5



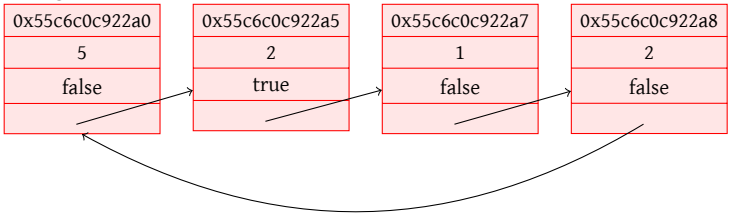
— Ligne 6



— Ligne 8



— Ligne 9



Cette méthode d'allocation a un défaut majeur : elle fragmente l'espace libre. Dans le programme suivant, il sera impossible d'allouer `b` car la liste circulaire contient deux blocs libres de 5 octets et non une bloc libre de 10 octets.

```
creation_blocs_libres(10);

uint8_t *a = allocation(5)
liberation(a);
uint8_t *b = allocation(10);
```

■ **Remarque 5.10** On peut observer ce phénomène sur le diagramme précédent à la ligne 8 où deux blocs contigus sont libres et pourraient être fusionnés en un unique bloc de 5 octets.

■

On peut éviter cela en effectuant une phase de coalescence des blocs libres à la libération. En vertu de la nature de la liste circulaire, il est nécessaire de fusionner un bloc libre avec les blocs suivants. En utilisant une liste circulaire doublement chaînée, on pourrait également fusionner avec les blocs précédents.

Pour cela on change la fonction `liberation` ainsi :

```
void liberation(void *adresse)
{
    struct bloc *premier_bloc = blocs_libres;
    struct bloc *bloc_courant = premier_bloc;

    while (bloc_courant->adresse != adresse)
    {
        bloc_courant = bloc_courant->suivant;
        // adresse invalide
        assert(bloc_courant != premier_bloc);
    }

    // pas de double libération
    assert(!bloc_courant->libre);

    // on libère l'adresse
    bloc_courant->libre = true;

    premier_bloc = bloc_courant;
    bloc_courant = bloc_courant->suivant;

    while (bloc_courant != premier_bloc && bloc_courant->libre)
    {
        struct bloc* actuel = bloc_courant;
        premier_bloc->taille += bloc_courant->taille;
        bloc_courant = bloc_courant->suivant;
        free(actuel); // on libere le bloc inutile
    }

    premier_bloc->suivant = bloc_courant;
}
```