Cours : Algorithmique

Table des matières

1 Diviser pour régner

1.1 Principe

Le principe de la méthode *diviser pour régner* (*divide and conquer* en anglais) est fondamentalement récursif. On souhaite résoudre une instance de taille n d'un certain problème.

— Si notre instance est un cas de base, la solution est immédiate.
— Sinon, on résout le problème de la manière suivante :
  — On la sépare en un certain nombre d'instances strictement plus petites du même problème.
  — On résout récursivement chacune de ces instances.
  — On reconstruit la solution à notre instance initiale à partir des solutions aux sous-problèmes.

Si l'on divise le problème en k sous problèmes de taille n/p, et que l'on note $T(n)$ le temps de calcul maximal sur une instance de taille n, on aura

$$T(n) \leqslant \underbrace{f(n)}_{\text{séparation}} + \underbrace{kT(n/p)}_{\text{appels récursifs}} + \underbrace{g(n)}_{\text{fusion}}.$$

Remarque

⇨ Pour être parfaitement rigoureux, il faudrait ici des parties entières supérieures et inférieures. Cela complique énormément l'analyse pour un gain d'information nul. Il est possible de s'en débarrasser sans sacrifier la rigueur, et nous les omettrons donc par la suite.

1.2 Un exemple idiot : recherche du maximum

Considérons le problème suivant : on dispose d'un tableau de n entiers, et l'on souhaite en déterminer le maximum. Il est tout à fait possible d'utiliser une stratégie diviser pour régner :

— Les cas de base sont les tableaux de taille 1.
— Si la taille n du tableau est supérieure ou égale à 2, on effectue un appel récursif sur la partie gauche et un sur la partie droite, puis l'on renvoie le maximum des deux résultats.

En notant $f(n)$ le nombre de comparaisons pour un tableau de taille n, on obtient $f(2n) = 1 + 2f(n)$. Si l'on pose alors $u_k := f(2^k)$, on a $u_1 = 0$ et

$$\forall k \in \mathbb{N}, \quad u_{k+1} = 1 + 2u_k.$$

On en déduit facilement $u_k = 2^k - 1$, c'est-à-dire $f(2^k) = 2^k - 1$. On remarque que ce nombre de comparaisons est exactement le même que celui obtenu par l'algorithme « naturel » pour calculer le maximum : on n'a donc rien gagné et la stratégie diviser pour régner n'a aucun intérêt ici.

Remarques

⇨ La technique consistant à se concentrer sur la complexité pour une instance de taille p^i quand on divise en sous-problèmes de taille n/p sera systématiquement utilisée. Elle permet notamment de se débarrasser des parties entières.

⇨ Ici, il n'est pas difficile de montrer que l'algorithme élémentaire est optimal, donc on aurait pu savoir dès le début que notre approche n'avait aucun intérêt.

1.3 Exemple paradigmatique : le tri fusion

On redonne un code possible pour le tri fusion d'une liste en OCaml :

```
let rec tri_fusion (w : 'a list) : 'a list =
  match w with
  | [] | [_] -> w
  | _ ->
    let u, v = separe w in
    let u_trie = tri_fusion u in
    let v_trie = tri_fusion v in
    fusionne u_trie v_trie
```

Ici, les étapes d'une stratégie diviser pour régner apparaissent clairement et portent des noms on ne peut plus explicites :

— Le premier cas du `match` correspond aux cas de base.
— Dans les autres cas :
  — On commence par diviser le problème en deux sous-problèmes de taille $n/2$ à l'aide de la fonction `separe`.
  — On effectue les deux appels récursifs pour résoudre les sous-problèmes.
  — On fusionne les résultats à l'aide de la fonction `fusionne`.

Pour étudier la complexité du tri fusion, il est bien sûr nécessaire de connaître celles des fonctions `separe` et `fusionne`. On en rappelle une implémentation possible :

```
let rec separe (w : 'a list) : 'a list * 'a list =
  match w with
  | [] -> ([], [])
  | [ x ] -> ([ x ], [])
  | x :: y :: xs ->
      let u, v = separe xs in
      (x :: u, y :: v)

let rec fusionne (u : 'a list) (v : 'a list) : 'a list =
  match (u, v) with
  | [], _ -> v
  | _, [] -> u
  | x :: xs, y :: ys ->
      if x <= y then
        x :: fusionne xs v
      else
        y :: fusionne u ys
```

— La fonction `separe` effectue un parcours de son argument avec un travail constant pour chaque élément, sa complexité est donc en $\mathrm{O}(|w|)$, où w est la liste à séparer.
— La fonction `fusionne` parcourt les deux listes qu'on lui passe en argument. Elle s'arrête dès qu'elle arrive au bout de l'une des deux, là aussi avec un travail constant pour chaque élément. Sa complexité est donc en $\mathrm{O}(|u| + |v|)$.

Pour le tri fusion, en partant d'une liste de taille 2^i, on a donc :

— Un appel à `separe` sur une liste de taille 2^i, qui prend un temps majoré par $A2^i$, où A est une constante.
— Deux appels récursifs sur des listes de taille 2^{i-1}.
— Un appel à `fusionne` sur deux listes de taille 2^{i-1}, qui prend un temps majoré par $B2^i$.

Au total on obtient, en notant $T(n)$ la complexité dans le pire cas du tri fusion pour une liste de taille n :

$$\begin{aligned} T(2^i) &\leqslant 2T(2^{i-1}) + A2^i + B2^i \\ &\leqslant 2T(2^{i-1}) + C2^i \quad \text{où } C \coloneqq A + B. \end{aligned}$$

On divise alors par 2^i.

$$\frac{T(2^i)}{2^i} \leqslant \frac{T(2^{i-1})}{2^{i-1}} + C$$

On fait apparaître une somme télescopique.

$$\frac{T(2^i)}{2^i} - \frac{T(2^{i-1})}{2^{i-1}} \leqslant C$$

On somme pour $1 \leqslant i \leqslant k$.

$$\frac{T(2^k)}{2^k} - \frac{T(2^0)}{2^0} \leqslant Ck$$

On en déduit

$$T(2^k) \leqslant Dk2^k \quad \text{où } D \text{ est une constante.}$$

Si $n = 2^k$, on a $k = \log_2 n$ et donc $T(n) \leqslant Dn \log_2 n$. On voudrait en déduire que $T(n) = \mathrm{O}(n \log n)$ dans le cas général, mais on ne peut pas le faire directement puisque le calcul ci-dessus ne concerne que les puissances de deux. En supposant que T est croissante, ce qui est « essentiellement » vrai, on a :

$$\begin{aligned} T(n) &\leqslant T(2^{\lceil \log_2 n \rceil}) \\ &\leqslant D \cdot \lceil \log_2 n \rceil \cdot 2^{\lceil \log_2 n \rceil} \\ &\leqslant D(1 + \log_2 n) 2^{1+\log_2 n} \\ &= \mathrm{O}(n \log n). \end{aligned}$$

Théorème 1.1: Complexité du tri fusion

En supposant que la comparaison entre deux éléments se fasse en temps constant, le tri fusion a une complexité dans le pire cas en $\mathrm{O}(n \log n)$, où n est la longueur de la liste à trier.

Remarque

⇨ Nous verrons que cette complexité est optimale pour un tri par comparaison. Cela ne signifie pas que le tri fusion est « le meilleur tri du monde », mais simplement qu'un tri par comparaison fait un nombre de comparaisons qui est au moins de l'ordre de $n \log n$.

1.4 Tri rapide

Le *tri rapide* (*quicksort* en anglais), introduit par Hoare en 1959, est aujourd'hui l'un des algorithmes les plus utilisés pour trier des tableaux, malgré quelques défauts que nous évoquerons ultérieurement.

Table 1

NUMBER OF ITEMS	MERGE SORT	QUICKSORT
500	2 min 8 sec	1 min 21 sec
1,000	4 min 48 sec	3 min 8 sec
1,500	8 min 15 sec*	5 min 6 sec
2,000	11 min 0 sec*	6 min 47 sec

* These figures were computed by formula, since they cannot be achieved on the 405 owing to limited store size.

Comparaison expérimentale entre le tri fusion et le tri rapide, extrait de l'article original.

Son principe est le suivant :

- Si le tableau possède 0 ou 1 élément, c'est un cas de base.
- Sinon :
  - On choisit un élément appelé *pivot*.
  - On partage le tableau en deux : à gauche les éléments inférieurs au pivot, à droite ceux supérieurs au pivot.
  - On trie récursivement ces deux parties.

Contrairement au tri fusion, l'étape de fusion après le tri des deux parties est triviale : tous les éléments de la « petite » partie sont nécessairement inférieurs à ceux de la « grande » partie.

Il y a trois points à préciser :

- Que fait-on des éléments égaux au pivot ? On peut choisir de les mettre avec les petits ou créer une troisième partie ne contenant que ces éléments. La seule chose qu'on ne peut pas faire, c'est en mettre certains avec les petits et d'autres avec les grands.

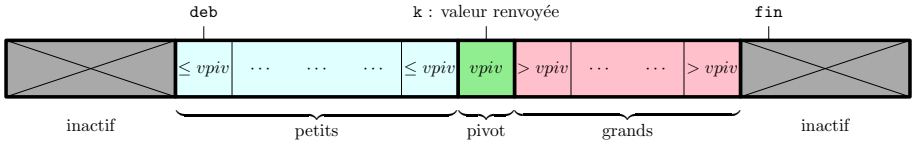
— Comment choisit-on le pivot ? On peut prendre systématiquement le premier élément, ou choisir au hasard, ou utiliser une heuristique pour essayer d'obtenir souvent un « bon » pivot, c'est-à-dire un pivot partageant le tableau en deux parties de tailles similaires. Nous y reviendrons, puisque c'est en fait une question importante.
— Le tri se fait-il *en place* ou non ? Un tri est dit « en place » s'il n'utilise pas de stockage auxiliaire, ou plutôt si la quantité de stockage auxiliaire utilisée est indépendante de la taille du tableau à trier. Il est un peu plus simple d'écrire une version qui n'est pas en place, mais un « vrai » tri rapide se fait toujours en place.

Dans ce qui suit, on utilise la notation $t[i \dots j[$ (qui n'existe ni en OCaml ni en C) pour désigner la partie du tableau t comprise entre l'indice i inclus et l'indice j exclu.

1.4.1 Fonction principale

Pour commencer, on va supposer que l'on dispose d'une fonction `partitionne` ayant la spécification suivante :
— Elle prend en entrée un tableau t de taille n et trois entiers deb, fin et i_{piv} vérifiant $0 \leqslant deb \leqslant i_{piv} < fin \leqslant n$.
— Elle renvoie un entier k vérifiant $deb \leqslant k < fin$;
— en notant t' l'état du tableau après l'appel et t l'état avant l'appel, on a :
  — $t'[0 \dots deb[= t[0 \dots deb[$ et $t'[fin \dots n[= t[fin \dots n[$; la fonction n'a rien modifié en dehors de la zone précisée.
  — $t'[k] = t[i_{piv}]$. L'indice renvoyé correspond à la position finale du pivot.
  — Tous les éléments de $t'[deb \dots k[$ sont inférieurs ou égaux à $t[k]$.
  — Tous les éléments de $t'[k+1 \dots fin[$ sont strictement supérieurs à $t[k]$.

Autrement dit, l'état après l'appel est le suivant :



État après partitionnement.

Exercice 1

⇨ Écrire en C une fonction `void tri_rapide(int t[], int n)`, utilisant la fonction `partitionne` que l'on supposera déjà écrite, et triant son argument par ordre croissant suivant l'algorithme décrit plus haut. Pour l'instant, on prendra comme pivot le premier élément de la section active.

1.4.2 Schéma de partitionnement de Lomuto

Il reste à écrire la fonction de partition, dont on souhaite qu'elle s'exécute en temps linéaire en la taille de la partie à partitionner et constant en espace. Il y a deux manières classiques de procéder. Celle que nous présentons ici est connue sous le nom de *schéma de partitionnement de Lomuto*. L'autre, connue sous le nom de *schéma de Hoare*, sera vue en travaux pratiques.
— On commence par échanger le pivot avec le dernier élément de la partie active.
— Ensuite, on parcourt cette partie de gauche à droite, en maintenant deux variables i et j et l'invariant suivant :
  — Si $x \in t[deb : i]$, alors $x \leqslant v_{piv}$.
  — Si $x \in t[i : j]$, alors $x > v_{piv}$.
  — Les éléments de $t[j : fin - 1]$ n'ont pas encore été étudiés.
— Finalement, on place le pivot au bon endroit et l'on renvoie son nouvel indice.

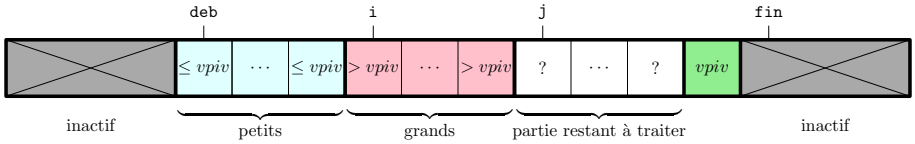


FIGURE 1 – Schéma de Lomuto.

Exercice 2

⇨ On dispose de la fonction suivante :

```c
void echange(int t[], int n, int i, int j) {
    int x = t[i];
    t[i] = t[j];
    t[j] = x;
}
```

Écrire une fonction `partitionne`, fonctionnant suivant le schéma de Lomuto et ayant la spécification donnée plus haut, et justifier sa correction.

```c
int partitionne(int t[], int n, int deb, int fin, int piv)
```

1.4.3 Complexité du tri rapide

Si t est un tableau de taille n, le nombre de comparaisons effectuées pour trier n est

$$C(t) = C(t[0 \dots k[) + C(t[k+1 \dots n[) + n - 1$$

avec k la position du pivot après partitionnement.

— Si l'on est capable de garantir à chaque étape que le pivot partage équitablement le tableau, on retrouve essentiellement la relation du tri fusion : $O(n)$ pour la partition/fusion et deux appels sur des tableaux deux fois plus petits. C'est le meilleur cas pour le tri rapide et l'on obtient alors une complexité en $O(n \log n)$.

— Si au contraire le pivot choisi est systématiquement le minimum (ou le maximum) de la partie considérée, la relation devient $C(n) = C(n-1) + n - 1$. On obtient alors une complexité en $O(n^2)$, ce qui est le pire cas pour le tri rapide.

Théorème 1.2: Complexité du tri rapide

Le tri rapide a une complexité de $O(n^2)$ dans le pire cas et de $O(n \log n)$ dans le meilleur cas. Si l'on prend comme pivot le premier élément de la partie active, la complexité sera quadratique si l'on appelle la fonction sur un tableau qui est déjà trié.

Remarque

⇨ Nous reviendrons sur la complexité du tri rapide ultérieurement. Il y a trois questions importantes :

— Peut-on choisir le pivot de manière à *garantir* un cas favorable ?

— Quelle est la complexité *en moyenne* ?

— Peut-on obtenir quelque chose qui soit efficace *en pratique* (avec une probabilité très proche de 1) ?

1.5 Algorithme de Karatsuba

On souhaite multiplier deux entiers en base b, de taille non bornée. On considère que multiplier ou ajouter deux *chiffres* est une opération élémentaire. En revanche, il est bien évident qu'une opération arithmétique sur deux *nombres* de n chiffres chacun n'en est pas une.

On suppose qu'un entier « long » $x = \sum_{i=0}^{n-1} x_i b^i$ est représenté par un tableau $(x_0, \dots, x_{n-1})$ d'entiers « machine » et que l'on dispose de trois fonctions :

— longueur(x) qui renvoie n et s'exécute en temps constant.

— chiffre(x, i) qui renvoie x_i si $0 \leqslant i < n$ et 0 sinon et s'exécute également en temps constant.

— normalise(x) qui s'exécute en temps linéaire en n et élimine les « zéros en tête ». Autrement dit,

$$\text{normalise}((x_0, \dots, x_{n-1}))$$

renvoie $(x_0, \dots, x_{p-1})$ avec p tel que $x_{p-1} \neq 0$ et $x_i = 0$ pour $p \leqslant i < n$.

Les entiers non nuls que nous considérerons seront toujours « normalisés », et nous ne renverrons que des entiers normalisés.

— *Addition* : On peut commencer par remarquer qu'ajouter deux nombres de n chiffres peut se faire en temps $O(n)$ par l'algorithme élémentaire, et que c'est clairement optimal (on est bien obligé de lire tous les chiffres).

fonction AJOUTE(x, y)
  $n \leftarrow \max(longueur(x), longueur(y))$
  $z \leftarrow (0, \dots, 0)$ (longueur $n+1$)
  $retenue \leftarrow 0$
  pour $i = 0$ à $n-1$ **faire**
    $c \leftarrow chiffre(x, i) + chiffre(y, i) + retenue$
    $z_i \leftarrow c \pmod b$
    $retenue \leftarrow c/b$ (quotient de la division euclidienne)
  fin pour
  $z_n \leftarrow retenue$
  renvoyer $normalise(z)$
fin fonction

— *Multiplication par une puissance de b* : Multiplier un nombre par une puissance de b peut se faire en temps proportionnel à son nombre de chiffres : on a MulBase$((x_0, \dots, x_{n-1}), k) = (0, \dots, 0, x_0, \dots, x_{n-1})$, avec k zéros au début.

— *Multiplication d'un nombre par un chiffre* : À nouveau, cette opération peut facilement s'exécuter en temps linéaire en le nombre de chiffres. On suppose que l'on dispose d'une fonction MulChiffre(x, c) qui prend en entrée un nombre x de n chiffres et un chiffre c et renvoie xc, en temps $O(n)$.

— *Multiplication naïve* : L'algorithme présenté ci-dessus pour l'addition est l'algorithme « école primaire ». On peut de même formaliser la manière dont on « pose » habituellement une multiplication.

Exercice 3

⇨ 1. Écrire en pseudo-code une fonction MulNaif(x, y) qui calcule le produit xy avec l'algorithme « école primaire ».
2. Déterminer la complexité de cette fonction. On supposera pour simplifier que longueur(x) = longueur$(y) = n$.

1.5.1 Diviser pour régner : première tentative

On considère deux entiers x et y possédant tous les deux $2n$ chiffres. On peut décomposer ces nombres comme suit :

$$x = x_{lo} + b^n x_{hi} \qquad \text{et} \qquad y = y_{lo} + b^n y_{hi}.$$

On a alors

$$xy = x_{lo}y_{lo} + (x_{lo}y_{hi} + x_{hi}y_{lo})b^n + x_{hi}y_{hi} \cdot b^{2n}.$$

Analysons les opérations à effectuer :
— Trois additions dont les opérandes ont au plus $2n$ chiffres, en $\mathrm{O}(n)$.
— Deux décalages (multiplications par des puissances de la base), encore en $\mathrm{O}(n)$.
— Quatre multiplications d'entiers possédant chacun n chiffres.
On obtient donc une relation de la forme

$$T(2n) = 4T(n) + \mathrm{O}(n).$$

Malheureusement, ce n'est pas mieux que la version naïve. En effet, on a $T(2n) \geqslant 4T(n)$, et l'on obtient alors facilement que $T(2^k) \geqslant A4^k$ (avec $A > 0$ une constante), ce qui est quadratique.

1.5.2 L'algorithme de Karatsuba

L'idée de l'algorithme de Karatsuba est d'éliminer l'une des multiplications en la remplaçant par des additions (ou soustractions, ce qui revient au même) supplémentaires. On pose :

$$\begin{cases} u \coloneqq x_{lo}y_{lo} \\ v \coloneqq x_{hi}y_{hi} \\ w \coloneqq (x_{lo} + x_{hi})(y_{lo} + y_{hi}) \end{cases}.$$

On a alors

$$xy = u + (w - u - v)b^n + vb^{2n}.$$

1.5.3 Complexité de l'algorithme de Karatsuba

Pour multiplier deux entiers de taille 2^i, on effectue :
— Un nombre constant d'opérations en temps linéaire (additions, soustractions, décalages), qui prennent au total un temps $\mathrm{O}(2^i)$.
— Trois appels récursifs sur des entiers de taille 2^{i-1}.
On a donc

$$T(2^i) \leqslant 3T(2^{i-1}) + A2^i.$$

En divisant par 3^i

$$\frac{T(2^i)}{3^i} - \frac{T(2^{i-1})}{3^{i-1}} \leqslant A(2/3)^i.$$

On en déduit en sommant de $i = 1$ à k

$$\frac{T(2^k)}{3^k} - B \leqslant A\sum_{i=1}^{k}(2/3)^i.$$

Or la somme de droite est bornée puisque c'est une somme géométrique de raison $2/3$. On obtient donc $T(2^k) \leqslant C3^k$ en remultipliant par 3^k. On observe à présent que

$$3^k = \exp(k \ln 3) = \exp\left(k \ln 2 \cdot \frac{\ln 3}{\ln 2}\right) = \left(2^k\right)^{\ln 3/\ln 2}.$$

Ainsi, si $n = 2^k$, on a $T(n) = \mathrm{O}(n^{\ln 3/\ln 2}) = \mathrm{O}(n^{1.59})$. Pour étendre le résultat aux n quelconques, il faut d'abord expliquer comment traiter le cas où le nombre de chiffres est impair, ce qui ne pose pas de problème, puis utiliser la croissance de T

$$T(n) \leqslant T(2^{\lceil \log_2 n \rceil}) \leqslant A\left(2^{\lceil \log_2 n \rceil}\right)^{\log_2 3} \leqslant A(2n)^{\log_2 3} \leqslant Bn^{\log_2 3}.$$

1.6 Complexité des algorithmes « diviser pour régner »

On rappelle le principe général de la méthode, et l'on fixe les notations pour cette partie :
— Si l'instance à traiter est un cas de base, on renvoie la réponse.
— Sinon :
  — On la sépare en a instances de taille n/b (en omettant les parties entières), ce qui prend un temps $f_s(n)$.
  — On fait les appels récursifs, ce qui prend un temps $aT(n/b)$.
  — On « fusionne », c'est-à-dire que l'on calcule le résultat final à partir des résultats des appels récursifs, ce qui prend un temps $f_f(n)$.

Au total, en notant $f = f_s + f_f$, on a la relation suivante :

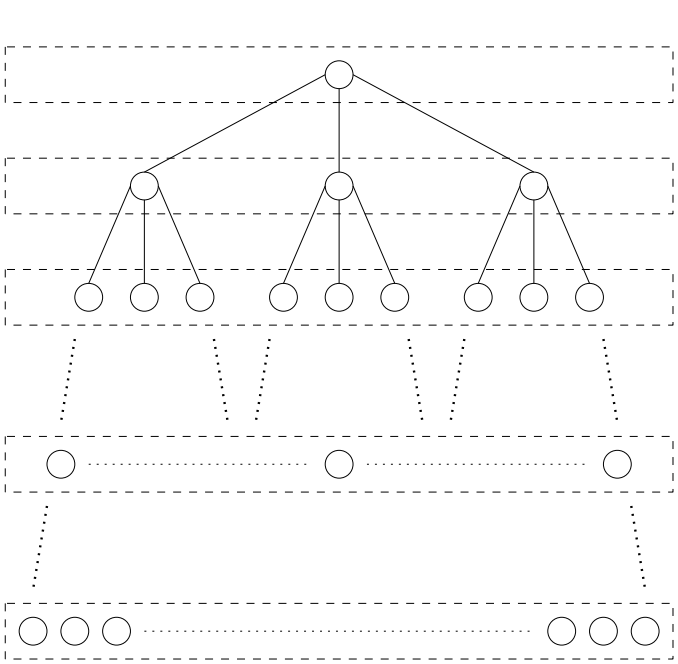
$$T(n) = aT(n/b) + f(n).$$

Avant de passer à l'analyse, on peut faire quelques remarques.
— On n'a pas précisé ce que représente $T(n)$, mais il s'agit logiquement de la complexité dans le pire des cas pour une instance de taille n. Comme il n'y a aucune raison que le pire cas pour n donne lieu à des appels récursifs sur des pires cas pour n/b, on aura plutôt $T(n) \leqslant aT(n/b) + f(n)$. Si l'on souhaite obtenir un O, ce n'est absolument pas un problème.
— Si n n'est pas un multiple b, ce qui précède n'a pas de sens. On fera donc l'analyse pour $n \coloneqq b^k$ et l'on généralisera ensuite.
— Pour pouvoir faire cette généralisation, on aura besoin que T soit croissante. Ce n'est pas le cas en général, même si ce n'est « pas loin d'être vrai » le plus souvent : on peut remplacer $T(n)$ par $T'(n) \coloneqq \max_{k \leqslant n} T(k)$ et faire disparaître le problème. On aura encore

$$T'(n) \leqslant aT'(n/b) + f(n),$$

et T' est croissante par construction.

La bonne manière de raisonner ici est d'observer la structure de l'arbre d'appels, et de déterminer la quantité de travail effectué à chaque niveau de cet arbre :



$$\text{Travail total : } T\left(b^k\right) = \sum_{i=0}^{k} a^i f\left(b^{k-i}\right)$$

FIGURE 2 – Analyse d'une relation $T(n) = aT(n/b) + f(n)$ (le schéma est fait avec $a = 3$).

Si l'on souhaite retrouver ce résultat sans faire de dessin, c'est très simple :
— On pose $u_i = T(b^i)/a^i$.
— On a alors $u_{i+1} = u_i + f(b^i)/a^i$.
— On somme les $u_{i+1} - u_i$ et l'on obtient

$$u_k - u_0 = \sum_{i=0}^{k-1} \frac{f(b^i)}{a^i}.$$

— On multiplie par a^k, on obtient

$$T(b^k) = a^k T(b^0) + \sum_{i=0}^{k-1} a^{k-i} f(b^i).$$

— En posant $f(1) = T(1)$, ce qui est parfaitement logique, on obtient exactement

$$T(b^k) = \sum_{i=0}^{k} a^{k-i} f(b^i).$$

Ensuite, on remarque que $T(n) \leqslant T\left(b^{\lceil \log_b n \rceil}\right)$ pour traiter le cas général (où n n'est pas une puissance de b). Dans le cas particulier où $f(n) = \Theta(n^p)$ pour une certaine constante p, on dispose d'un théorème classique mais hors programme et pas vraiment utile à retenir.

Théorème 1.3: Théorème maître

Si T vérifie $T(n) \leqslant aT(n/b) + An^p$ où A est une constante, et si T est croissante, on a :
— Si $b^p < a$, alors $T(n) = \mathrm{O}(n^{\log_b a})$.
— Si $b^p = a$, alors $T(n) = \mathrm{O}(n^{\log_b a} \log n)$.
— Si $b^p > a$, alors $T(n) = \mathrm{O}(n^p)$.

Remarques

⇨ Il faut bien comprendre à quoi correspondent les trois cas :
— Dans le premier cas, le terme en $\mathrm{O}(n^p)$, qui correspond au travail de séparation-fusion, est négligeable devant le coût des appels récursifs. Si l'on considère l'arbre d'appels, la majorité du temps est passé dans les feuilles. Autrement dit, la somme $\sum_{i=0}^{k} a^i f(b^{k-i})$ est de l'ordre de a^k.
— Dans le troisième cas, c'est le coût des appels récursifs qui est négligeable : la majorité du temps est passé à la racine de l'arbre d'appels, la somme est de l'ordre de $f(b^k)$.
— Dans le deuxième cas, l'étape de séparation-fusion et les appels récursifs ont un coût similaire : autrement dit, le coût total pour traiter chaque niveau de l'arbre d'appels est approximativement le même et le coût total s'obtient en multipliant ce coût par niveau par le nombre de niveaux.

⇨ Comme dit plus haut, je vous déconseille de chercher à retenir ce théorème : refaites le raisonnement, en partant de l'arbre d'appels ou directement par le calcul, pour le cas qui vous intéresse.

Exercice 4

⇨ Déterminer par cette méthode la complexité temporelle :

1. De la recherche dichotomique.
2. Du tri fusion.
3. De l'algorithme de Karatsuba.
4. De l'algorithme de Strassen, qui permet de réduire la multiplication de deux matrices $2n \times 2n$ à 7 multiplications de matrices $n \times n$ et un nombre constant d'additions de matrices $n \times n$ ou $2n \times 2n$.

2 Programmation dynamique

2.1 Un exemple simple et classique

Après un changement de l'équipe dirigeante, l'entreprise Mondelez International, qui fabrique les barres Toblerone, décide de rationaliser sa production pour maximiser ses revenus. En effet, leur chaîne de production fabrique des barres de n « carreaux » qui sont ensuite coupées en barres plus petites avant d'être vendues. Mais une récente étude de marché a déterminé le prix auquel on pouvait vendre des barres de longueur k (pour $1 \leqslant k \leqslant n$), et ce prix s'avère ne pas avoir de relation simple avec k. Le problème est donc de décider comment découper la barre initiale de n carreaux en des barres plus petites pour maximiser le prix de vente total. Pour simplifier, on néglige le coût de la découpe et de l'emballage.

Dans tout le problème, on considérera que l'on dispose d'un tableau `t`, indicé de 0 à n, tel que `t.(k)` soit le prix de vente d'une barre de longueur k, et que le prix d'un morceau de taille 0 est 0 (autrement dit que `t.(0) = 0`). Remarquez que si `t.(n)` est suffisamment grand, la solution optimale peut très bien être de ne pas découper la barre.

FIGURE 3 – Le cœur du problème.

On donne un exemple de tableau t pour $n = 10$.

$\boldsymbol{i}$	0	1	2	3	4	5	6	7	8	9	10
$\boldsymbol{t_i}$	0	1	5	8	9	10	17	17	20	24	26

La solution optimale pour cet exemple est de découper la barre en deux blocs de taille 2 et un bloc de taille 6, pour une valeur totale de 27.

Formalisation du problème

On considère donnés un entier $n \geqslant 0$ et un tableau t d'entiers positifs, de taille $n+1$, vérifiant $t.(0) = 0$. Dans tout ce qui suit, k désigne un entier de $[\![0, n]\!]$.

— Une *découpe* de taille k est un p-uplet $(c_1, \ldots, c_p)$ d'éléments de $\mathbb{N}^*$ vérifiant $\sum_{i=1}^{p} c_i = k$.
— On note $\mathcal{D}(k)$ l'ensemble des découpes de k.
— La *valeur* d'une découpe $d = (c_1, \ldots, c_p)$ est la quantité $v(d) = \sum_{i=1}^{p} t.(c_i)$.
— La *valeur optimale* d'une découpe de taille k est la quantité $v_{opt}(k) = \max_{d \in \mathcal{D}(k)} v(d)$.
— Une *découpe optimale* de taille k est un $d \in \mathcal{D}(k)$ vérifiant $v(d) = v_{opt}(k)$.

Le problème est de calculer $v_{opt}(n)$ et de fournir un exemple de découpe optimale.

2.1.1 Solution en force brute

La solution la plus évidente est d'énumérer toutes les découpes possibles et de déterminer celle ayant la plus grande valeur. On peut noter que deux découpes ne différant que par l'ordre des blocs, $(3, 3, 2, 2)$ et $(2, 3, 2, 3)$ par exemple, sont clairement équivalentes : on pourrait donc définir une notion de découpe canonique (morceaux triés par taille croissante, par exemple) et tenter de n'énumérer que ces découpes canoniques. Ce dernier point n'est en fait pas trivial, et l'on va donc se contenter d'énumérer *toutes* les découpes.

Pour cela, on peut établir une bijection entre l'ensemble des tableaux d de $n-1$ booléens et l'ensemble des découpes de n : on coupe après le $i+1$-ème « morceau » si et seulement si d_i est vrai.

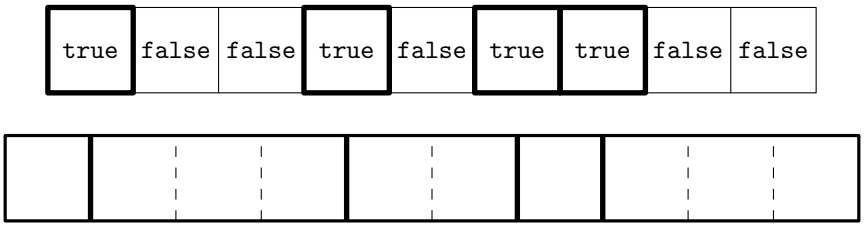


FIGURE 4 – La découpe $1, 3, 2, 1, 3$ et le tableau de booléens correspondant.

On peut alors

— Générer tous les tableaux (ou toutes les listes) de $n-1$ booléens.
— Convertir chacun de ces tableaux en une découpe comme décrit ci-dessus.
— Calculer la valeur de chacune de ces découpes.
— Renvoyer la valeur maximale, et un exemple de découpe optimale.

Complexité : Il y a 2^{n-1} tableaux à considérer. Les générer prendra un temps en $\Theta(2^n)$ ou $\Theta(n2^n)$ suivant la méthode choisie, et les traiter un temps en $\Theta(n2^n)$, puisqu'il faut tous les parcourir. Au total, on a donc une complexité en $\Theta(n2^n)$.

2.1.2 Sous-structure optimale : solution récursive

Pour arriver à une solution efficace, la première étape est de remarquer qu'une solution du problème, *i.e.* une découpe optimale d'une barre de taille n, « contient » des solutions à des versions plus petites du même problème. Pour simplifier, on peut supposer qu'à chaque étape on découpe ce qui reste de la barre en un bloc de taille k ($1 \leqslant k \leqslant n$) qui fera partie de la découpe finale et une nouvelle barre qui pourra à nouveau être découpée. Il est alors clair que cette nouvelle barre (de longueur $n-k$) devra elle aussi être découpée de manière optimale. On a donc

$$v(n) = \max_{1 \leqslant k \leqslant n} \big(t_k + v(n-k)\big).$$

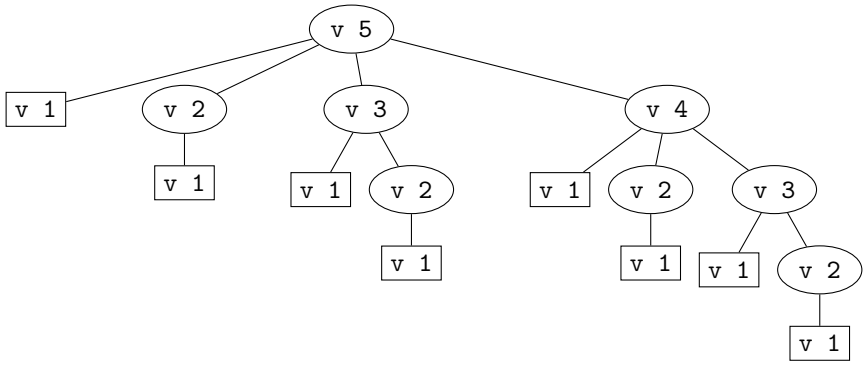
On obtient immédiatement une fonction OCaml :

```
let v_opt_rec (t : int array) : int =
  (* cas de base de val_aux : n = 0 et n = 1 *)
  let rec val_aux (n : int) : int =
    let v_opt = ref t.(n) in
    for k = 1 to n - 1 do
      let v = t.(n - k) + val_aux k in
      v_opt := max !v_opt v
    done;
    !v_opt in
  val_aux (Array.length t - 1)
```

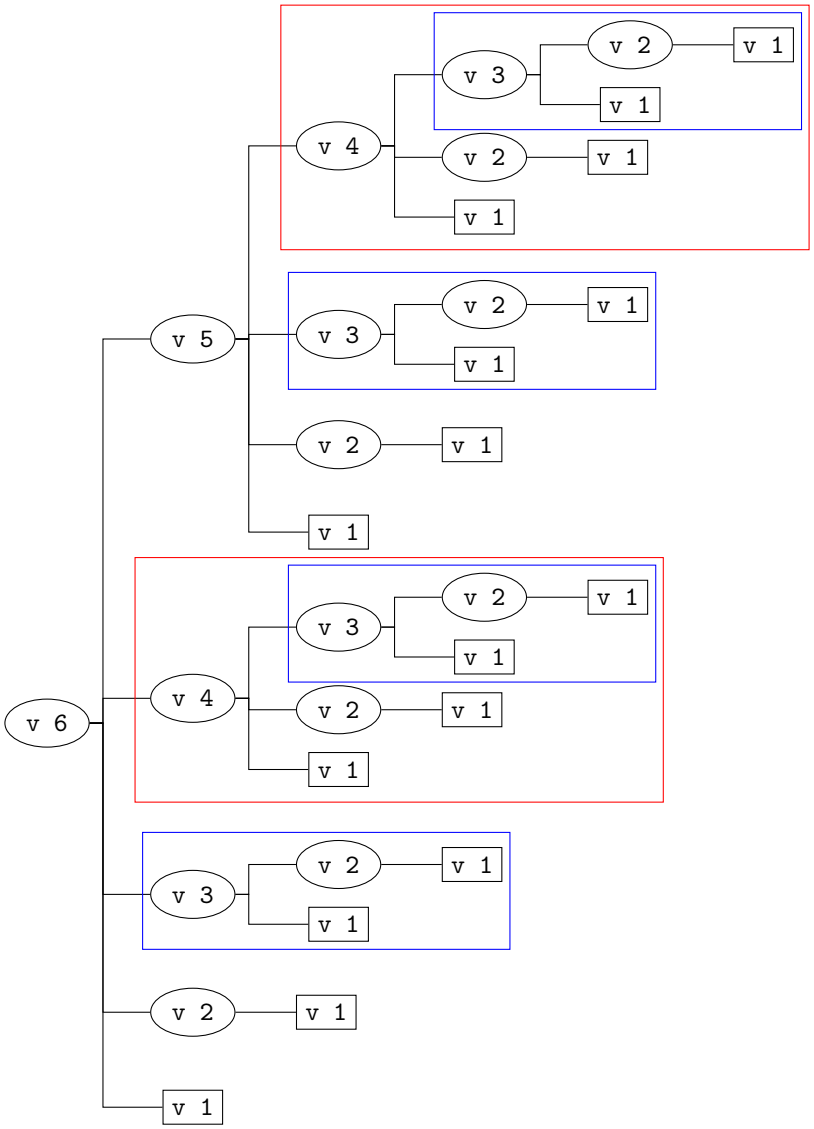
```
# v_opt_rec tab_exemple;;
- : int = 27
```

2.1.3 Analyse de la solution récursive naïve

Pour comprendre l'inefficacité de cette solution, le mieux est de commencer par contempler les arbres d'appels obtenus :



Découpage du Toblerone pour $n = 5$, récursif naïf



Découpage optimal pour $n = 6$, récursif naïf. Les sous-arbres encadrés d'une même couleur sont identiques.

Le problème est immédiatement apparent : on passe notre temps à recalculer la même chose ! C'est précisément dans les cas où les sous-problèmes se chevauchent, et donnent donc lieu à une répétition de calculs, que la programmation dynamique est intéressante.

Complexité : On peut assez aisément compter le nombre $\varphi(n)$ d'appels effectués par `v_opt_rec n`, c'est-à-dire le nombre total de nœuds de l'arbre correspondant. En effet, on a

$$\varphi(n) = 1 + \sum_{k=1}^{n-1} \varphi(n-k) = 1 + \sum_{k=1}^{n-1} \varphi(k).$$

Une récurrence immédiate montre alors que $\varphi(n) = 2^{n-1}$. Comme le nombre d'appels minore la complexité, celle-ci reste au moins exponentielle.

Remarque

⇨ On se convaincra sans trop de mal que la complexité est en fait en $\mathrm{O}(2^n)$.

2.1.4 Solution dynamique : version itérative

Une manière plus « intelligente » d'utiliser la relation de récurrence que l'on a trouvée est de construire un tableau auxiliaire contenant les v_k. On l'initialise à 0, on le remplit au fur et à mesure en utilisant exactement la même relation de récurrence que plus haut, et quand on a terminé on récupère le dernier élément qui correspond à $v(n)$, la valeur cherchée.

```
1 let v_opt_dyn (t : int array) : int =
2   let n = Array.length t - 1 in
3   let v = Array.make (n + 1) 0 in
4   for k = 0 to n do
```

```
    let value = ref 0 in
    for i = 1 to k do
      value := max !value (t.(i) + v.(k - i))
    done;
    v.(k) <- !value
  done;
  v.(n)
```

```
# v_opt_dyn tab_exemple;;
- : int = 27
```

Complexité : La boucle interne s'exécute en temps $\mathrm{O}(k)$, et donc la boucle externe en temps $\mathrm{O}\left(\sum_{k=0}^{n} k\right) = \mathrm{O}(n^2)$. L'initialisation se faisant en temps $\mathrm{O}(n)$, on a donc au total une complexité quadratique.

2.1.5 Solution dynamique : récursion avec mémoïsation

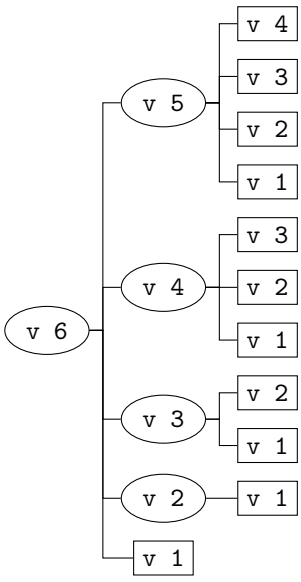
En fait, il est tout aussi simple de garder une solution récursive : il suffit de s'arranger pour ne pas faire plusieurs fois le même calcul. Pour cela, on utilise la technique dite de *mémoïsation* (pourquoi pas *mémorisation* ? Aucune raison valable, mais c'est comme ça).

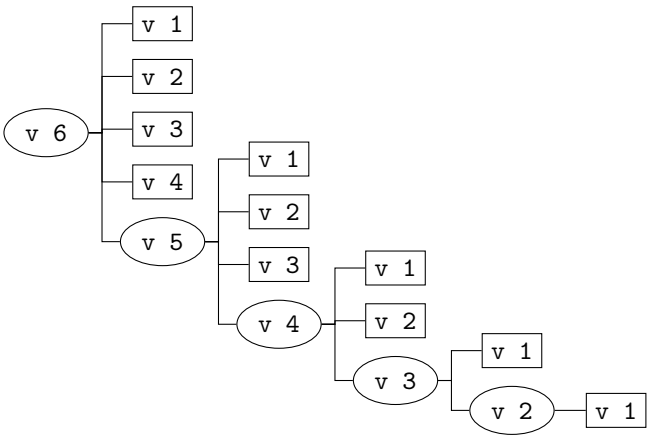
- On crée un tableau `v` destiné à recevoir les valeurs de $v(k)$.
- Ce tableau est initialisé avec une valeur « impossible ». Ici, on peut prendre `-1`, dans le cas général on utilisera des options et l'on initialisera à `None`.
- La fonction auxiliaire, récursive, prend ce tableau en plus de ses arguments « normaux ».
- Quand on l'appelle sur une valeur k :
  - Elle regarde dans le tableau `v` si la case k contient une « vraie » valeur.
  - Si c'est le cas, alors on a déjà calculé $v(k)$ et l'on peut directement renvoyer cette valeur.
  - Sinon, on effectue le calcul comme dans la version récursive naïve, puis l'on stocke le résultat, puis l'on renvoie ce résultat.
- La fonction principale, non récursive, se contente essentiellement de créer le tableau `v` et d'appeler la fonction auxiliaire.

```
let rec v_opt_mem_aux (n : int) (t : int array) (v : int array) : int =
  if v.(n) = -1 then begin
    let m = ref t.(n) in
    for k = 1 to n - 1 do
      m := max !m (v_opt_mem_aux k t v + t.(n - k))
    done;
    v.(n) <- !m
  end;
  v.(n)

let v_opt_mem (t : int array) : int =
  let n = Array.length t - 1 in
  let v = Array.make (n + 1) (-1) in
  v_opt_mem_aux n t v
```

On peut à nouveau regarder ce que donnent les arbres d'appels : c'est beaucoup plus raisonnable !





Dans les figures ci-dessus on trouve à gauche l'arbre d'appel pour la version récursive mémoïsée du découpage du Toblerone pour $n = 6$. L'arbre d'appel de droite correspond à l'arbre obtenu si on inverse le sens de la boucle `for`.

Complexité : On analyse la version dont le code est au-dessus pour une boucle `for` ascendante (arbre de gauche). Si l'on a déjà effectué les appels `v opt_mem_aux k` pour $1 \leqslant k < n$, l'appel `v_opt_mem_aux n` s'effectue en temps $\mathrm{O}(n)$ puisque chaque appel récursif renvoie le résultat immédiatement. Au total, pour un appel à `v_opt_mem n`, on a donc :

— Une initialisation en $\mathrm{O}(n)$ pour la création du tableau `v`.

— Un appel « nouveau » à `v_opt_mem_aux n`, qui se fait en temps

$$\mathrm{O}\left(\sum_{k=1}^{n-1} k\right) = \mathrm{O}(n^2).$$

Finalement, on obtient une complexité en $\mathrm{O}(n^2)$, comme pour la solution dynamique itérative.

2.1.6 Reconstruction de la découpe optimale

Pour l'instant, on a déterminé la valeur optimale d'une découpe de la barre, mais on n'est pas en mesure d'exhiber une découpe optimale. C'est à peu près systématique dans ce genre de problème : on commence par se préoccuper de déterminer la valeur optimale, puis on regarde ce qu'il faut modifier dans l'algorithme pour reconstruire la solution. Souvent, le tableau des valeurs calculées suffit ; parfois, il faut stocker un peu plus de choses.

Ici, supposons que l'on dispose du tableau `v` contenant les valeurs de $v(k)$ pour $k \in [\![0, n]\!]$. Il est alors possible de reconstruire une découpe optimale en temps $\mathrm{O}(n)$:

— On part de la case n du tableau.

— On se déplace vers la gauche du tableau en cherchant une case d'indice $n - k$ vérifiant $v(n-k) + t_k = v(n)$. Une telle case existe nécessairement d'après la définition récursive de v.

— On sait alors que l'on peut prendre un bloc de taille k comme dernier morceau de la découpe, et l'on complète cette découpe en cherchant une découpe optimale pour $n - k$.

Pour commencer, on modifie très légèrement notre fonction `v_opt` pour qu'elle renvoie le tableau `v` et non pas seulement le contenu de sa dernière case :

```
let v_opt_dyn (t : int array) : int array =
  let n = Array.length t - 1 in
  let v = Array.make (n + 1) 0 in
  for k = 0 to n do
    for i = 1 to k do
      v.(k) <- max v.(k) (t.(i) + v.(k - i))
    done
  done;
  v
```

On peut par exemple écrire le code suivant :

```
let rec reconstruit (v : int array) (t : int array) (n : int) : int list =
  if n = 0 then []
  else begin
    let i = ref 1 in
    while v.(n - !i) + t.(!i) <> v.(n) do
      i := !i + 1
    done;
    !i :: reconstruit v t (n - !i)
  end
```

Remarques

⇨ Si l'on avait voulu une reconstruction plus simple, et un peu plus efficace (temps proportionnel au nombre de blocs de la découpe et non à la taille totale), on aurait pu stocker les couples $v(n), k_0$ où $t_{k_0} + v(n - k_0) = \max_{1 \leqslant k \leqslant n}(t_k + v(n-k))$ dans le tableau au lieu de ne stocker que les $v(n)$. Ici, cela n'apporte pas grand-chose.

⇨ On reconstruit *une* découpe optimale. Reconstruire *toutes* les découpes optimales ne peut pas se faire rapidement, car leur nombre peut être très grand.

2.2 Principe général

Au départ, la programmation dynamique s'applique à des problèmes d'optimisation ayant les deux propriétés suivantes :

— *Sous-structure optimale* : Une solution optimale contient des solutions optimales pour des instances plus petites du même problème.
— *Chevauchement des sous-problèmes* : Une solution récursive en suivant le principe du « diviser pour régner » conduit à résoudre plusieurs fois les mêmes sous-problèmes.

Notons que les idées de la programmation dynamique peuvent être utilisées dans un cadre un peu plus large : on verra plus loin des exemples qui ne sont pas des problèmes d'optimisation. Le plus souvent, la difficulté est de transformer le problème initial, typiquement en calculant un peu plus que ce qui est demandé, pour que la propriété de sous-structure optimale soit vérifiée.

Parenthésage optimal : Le problème est le suivant. Étant donné un produit de k matrices $A_1 \cdots A_k$ à effectuer, et en considérant que la multiplication d'une matrice (n, p) par une matrice (p, q) demande npq multiplications élémentaires, choisir un parenthésage optimal minimisant le nombre total de multiplications élémentaires. Si l'on considère par exemple des matrices $A_1, \ldots, A_4$ de taille respective $(5, 2)$, $(2, 10)$, $(10, 4)$ et $(4, 1)$, on a les possibilités suivantes :
— $(A_1A_2)(A_3A_4)$ pour un coût de $5 \cdot 2 \cdot 10 + 10 \cdot 4 \cdot 1 + 5 \cdot 10 \cdot 1 = 190$.
— $A_1(A_2(A_3A_4))$ pour un coût de $10 \cdot 4 \cdot 1 + 2 \cdot 10 \cdot 1 + 5 \cdot 2 \cdot 1 = 70$.
— $A_1((A_2A_3)A_4)$ pour un coût de $2 \cdot 10 \cdot 4 + 2 \cdot 4 \cdot 1 + 5 \cdot 2 \cdot 1 = 98$.
— $((A_1A_2)A_3)A_4$ pour un coût de $5 \cdot 2 \cdot 10 + 5 \cdot 10 \cdot 4 + 5 \cdot 4 \cdot 1 = 320$.
— $(A_1(A_2A_3))A_4$ pour un coût de $2 \cdot 10 \cdot 4 + 5 \cdot 2 \cdot 4 + 5 \cdot 4 \cdot 1 = 140$.

Choisir un parenthésage revient à choisir un arbre binaire dont tous les nœuds internes sont étiquetés $\times$ et dont les feuilles, lues de gauche à droite, sont étiquetées $A_1, \ldots, A_k$. Il y a donc autant de parenthésages possibles que de formes d'arbres binaires à $k-1$ nœuds internes, c'est-à-dire c_{k-1}. Or $c_k \sim 4^k/\sqrt{\pi k^3}$: une solution en force brute n'est pas raisonnable pour k un tant soit peu grand.
— *Sous-structure optimale* : Dans une solution optimale, chaque sous-arbre, dont les feuilles sont étiquetées $A_i, A_{i+1}, \ldots, A_j$, fournit une solution optimale au problème pour la multiplication $A_1 \cdots A_j$.
— *Chevauchement des sous-problèmes* Pour choisir qui mettre dans le sous-arbre gauche et dans le sous-arbre droit, il faut calculer les coûts optimaux pour $A_1, \ldots, A_i$ et $A_{i+1}, \ldots, A_n$ pour $1 \leqslant i < n$. Mais les sous-problèmes pour $A_1, \ldots, A_4$ et pour $A_1, \ldots, A_6$ ne sont clairement pas disjoints : il faudra résoudre le premier pour résoudre le second.

Plus formellement, en notant l_i, c_i la taille de A_i et $\mathrm{opt}(i, j)$ le coût minimal du calcul de $\prod_{m=i}^{j} A_m$, on a

$$\mathrm{opt}(a, b) = \min_{a \leqslant i < b} \left[\mathrm{opt}(a, i) + \mathrm{opt}(i + 1, b) + l_a c_i c_b\right].$$

Il faudra finalement calculer les $\mathrm{opt}(i, j)$ pour tous les couples $1 \leqslant i \leqslant j \leqslant n$, et la programmation dynamique permettra de ne calculer chacune de ces valeurs qu'une seule fois.

2.3 *Top-down* ou *bottom-up*

Une fois que l'on a identifié une relation de récurrence se prêtant bien à une solution par programmation dynamique, on a deux choix :
— L'approche *ascendante* (*bottom-up*) consiste en gros à résoudre d'abord toutes les instances de taille 1, puis celles de taille 2 et ainsi de suite jusqu'à celle qui nous intéresse. Typiquement, on obtient une solution itérative.
— L'approche *descendante* (*top-down*) consiste à traduire directement la relation de récurrence en une fonction récursive et à mémoïser les appels.

En règle générale, l'approche *top-down* est plus simple à écrire et donc à privilégier puisqu'il n'y a pas à se préoccuper de l'ordre dans lequel les calculs sont effectués. Au niveau des performances et surtout de la complexité en espace, tout dépend du problème.
— S'il est simple de déterminer exactement quels calculs vont être nécessaires, l'approche *bottom-up* peut être intéressante. En particulier, on peut dans certains cas ne garder en mémoire que les résultats récents (ceux correspondant à des instances de taille $n-1$), et donc faire passer la complexité en espace de $\mathrm{O}(n^2)$ à $\mathrm{O}(n)$ ou de $\mathrm{O}(n)$ à $\mathrm{O}(1)$.
— En revanche, s'il n'est pas évident de déterminer *a priori* quelles instances il va réellement falloir résoudre, l'approche *top-down* peut éviter des calculs inutiles. Nous verrons cela dans le problème du sac à dos en TD.

Prenons un exemple sans autre intérêt que sa simplicité : Fibonacci.
— *Version récursive naïve* : On utilise directement la relation $u_{n+2} = u_{n+1} + u_n$.

```
let rec fibo_rec (n : int) : int =
  if n <= 1 then n
  else fibo_rec (n - 1) + fibo_rec (n - 2)
```

Chaque appel se fait en temps constant, sans compter les appels récursifs et le nombre d'appels vérifie $\varphi_{n+2} = \varphi_{n+1} + \varphi_n + 1$. En posant $v_n = \varphi_n + 1$, on a alors $v_{n+2} = v_{n+1} + v_n$ et $v_0 = v_1 = 1$. On en déduit que la complexité temporelle est proportionnelle à u_n et donc en $\Theta(\varphi^n)$ où $\varphi \coloneqq (1 + \sqrt{5})/2$. Chaque appel a une consommation mémoire en $O(1)$, et il y a au maximum n appels simultanément actifs (profondeur de l'arbre d'appels). On a donc une complexité mémoire en $O(n)$.

— *Version mémoïsée top-down* : Il faut créer un tableau de taille $n+1$ pour stocker les valeurs des u_i.

```
let fibo_mem (n : int) : int =
  let t = Array.make (n + 1) (-1) in
  t.(0) <- 0;
  if n >= 1 then t.(1) <- 1;
  let rec aux (k : int) : int =
    if t.(k) = -1 then t.(k) <- aux (k - 1) + aux (k - 2);
    t.(k) in
  aux n
```

Le tableau t occupe un espace proportionnel à n, et la profondeur d'appel est clairement majorée par n, donc la complexité spatiale est en $\mathrm{O}(n)$. Pour la complexité temporelle, supposons par exemple que l'appel `aux (k - 2)` soit effectué avant l'appel `aux (k - 1)` (l'analyse est légèrement différente dans le cas contraire, mais le résultat reste inchangé). L'appel `aux (k - 1)` s'effectuera alors en temps $\mathrm{O}(1)$, puisque t_{k-2} et t_{k-3} auront déjà été remplies. La complexité de `aux` vérifie donc $T(n) = T(n-2) + \mathrm{O}(1)$, ce qui donne $T(n) = \mathrm{O}(n)$: complexité temporelle linéaire.

— *Version dynamique bottom-up* : Si l'on ne réfléchit pas particulièrement, on obtient exactement les mêmes complexités que pour la version mémoïsée, mais l'on calcule explicitement les valeurs $u_0, \dots, u_n$ dans l'ordre.

```
let fibo_dyn n =
  let t = Array.make (n + 1) 0 in
  if n >= 1 then t.(1) <- 1;
  for i = 2 to n do
    t.(i) <- t.(i - 1) + t.(i - 2)
  done;
  t.(n)
```

On peut alors remarquer que les seules valeurs dont il faut se « souvenir » pour calculer u_k (et les u_i avec $i > k$) sont u_{k-1} et u_{k-2}. Au lieu d'utiliser un tableau, on peut donc se contenter de stocker le couple (u_{k-1}, u_{k-2}) en le mettant à jour au fur et à mesure : on obtient alors une complexité spatiale en $\mathrm{O}(1)$, la complexité temporelle restant inchangée.

```
let fibo_dyn_opt (n : int) : int =
  let couple = ref (0, 1) in
  for i = 2 to n + 1 do
    let a, b = !couple in
    couple := (b, a + b)
  done;
  fst !couple
```

On peut bien sûr appliquer cette technique pour les suites récurrentes doubles sans réaliser qu'il s'agit d'un cas particulier de programmation dynamique *bottom-up*.

Retour sur le parenthésage optimal : Comme on se rapproche ici d'une implémentation réelle, on prend une numérotation plus raisonnable : les matrices vont de A_0 à A_{n-1}, et $\mathrm{opt}(i,j)$ est le coût minimal pour le calcul de $\prod_{k=i}^{j-1} A_k$. On va être amené, pour calculer $\mathrm{opt}(0,n)$, à calculer les $\mathrm{opt}(i,j)$ pour différentes valeurs de i et j.

— *Approche top-down* : On crée un tableau bi-dimensionnel t pour stocker les $\mathrm{opt}(i,j)$. On écrit une fonction auxiliaire récursive `aux` dont le principe est le suivant, pour un appel `aux i j` :
  — Si $t_{i,j}$ n'est pas remplie :
    — On le calcule à l'aide de la formule

$$\texttt{aux i j} = \min_{i<k<j} (\texttt{aux i k} + \texttt{aux k j} + l_i l_k c_{j-1}).$$

    — On écrit le résultat dans $t_{i,j}$.
  — Dans tous les cas, on renvoie $t_{i,j}$.

— *Approche bottom-up* : On crée aussi un tableau t, mais cette fois, il faut faire attention à ce que les valeurs soient calculées dans le bon ordre, c'est-à-dire à ce que la case $t_{i,j}$ soit remplie avant qu'on ait besoin de son contenu. Ici, c'est possible en remplissant le tableau par $j-i$ croissants ; il y a quand même beaucoup plus de chances de faire une erreur d'indice quelque part que dans la version *top-down*.

On se convaincra assez aisément que les complexités tant spatiale que temporelle sont quadratiques dans les deux cas.

2.4 Stockage des résultats

Pour l'instant, on a toujours utilisé des tableaux, éventuellement bi-dimensionnels, pour stocker les résultats partiels. C'est possible le plus souvent, mais cela suppose que :

— La « récursion », qu'on écrive ou non une fonction récursive) se fait sur un ou des paramètres entiers.
— Ces paramètres sont plus petits pour les résultats partiels que pour le résultat final, ou au moins bornés et prévisibles.

Ces deux conditions ne sont pas toujours réunies, et on peut alors être amené à utiliser une solution plus générale : un dictionnaire.

Suite de Syracuse : Pour tout entier $a \in \mathbb{N}^*$, on définit la suite de Syracuse associée par

$$u_0(a) \coloneqq a \qquad \text{et} \qquad \forall n \in \mathbb{N}, \quad u_{n+1}(a) \coloneqq \begin{cases} \frac{u_n(a)}{2} & \text{si } u_n \text{ est pair} \\ 3u_n(a)+1 & \text{sinon.} \end{cases}$$

On admet que, pour toute valeur raisonnable de a, la suite finit par boucler sur $1, 4, 2$. On note $\mathrm{syr}(a)$ le plus petit n tel que $u_n = 1$, appelé *temps de vol* de a. Si l'on veut calculer *une* valeur de $\mathrm{syr}(p)$, on n'a guère d'autre choix que de procéder « bêtement » :

```
let rec syr (a : int) : int =
  if a = 1 then 0
  else if a mod 2 = 0 then 1 + syr (a / 2)
  else 1 + syr (3 * a + 1)
```

On pourrait éventuellement préférer une version itérative ou récursive terminale, mais ce n'est pas le but pour l'instant.

Dès que l'on calcule plusieurs valeurs, en revanche, on a toutes les chances de retomber lors d'un calcul intermédiaire sur une valeur déjà calculée : par exemple, en calculant $\mathrm{syr}(3)$, on suit la chaîne $3 \to 10 \to 5 \to 16 \to 8 \to 4 \to 2 \to 1$. Si l'on veut un peu plus tard calculer $\mathrm{syr}(5)$, il serait judicieux de ne pas tout recommencer. Supposons par exemple que l'on souhaite déterminer le maximum des $\mathrm{syr}(k)$ pour $1 \leqslant k \leqslant n$. Vu le comportement complexe de la suite, il est impossible de prévoir à l'avance quelles valeurs on va être amené à calculer : impossible ici de procéder en *bottom-up*. Pour la même raison, le cache sera nécessairement un dictionnaire : on ne peut pas borner les k pour lesquels on va calculer $\mathrm{syr}(k)$, et même si on le pouvait cette borne serait beaucoup trop grande. Morale de l'histoire : nous allons utiliser une fonction récursive mémoïsée utilisant un dictionnaire.

```
let max_syr (n : int) : int =
  let cache = Hashtbl.create 1000 in
  let rec syr (a : int) : int =
    if Hashtbl.mem cache a then Hashtbl.find cache a
    else if a = 1 then 0
    else begin
      let res =
        if a mod 2 = 0 then 1 + syr (a / 2)
        else 1 + syr (3 * a + 1) in
      Hashtbl.add cache a res;
      res
    end in
  let rec max_aux (k : int) : int =
    if k = 1 then 0
    else max (syr k) (max_aux (k - 1)) in
  max_aux n
```

3 Algorithmes gloutons

Les algorithmes gloutons (*greedy* en anglais) sont une classe d'algorithmes pour des problèmes d'optimisation, c'est-a-dire des problèmes pouvant se ramener à la minimisation d'une certaine fonction sous contraintes qui ont les propriétés suivantes :

— Ils construisent leur solution de manière graduelle. Autrement dit, ils décomposent le choix global à réaliser en une série de choix successifs *sur lesquels ils ne reviennent pas ultérieurement.*
— À chaque étape, ils font un choix *localement optimal.*

Ces algorithmes ont de nombreux avantages : ils sont souvent simples à concevoir et à implémenter, et ils s'exécutent rapidement. Cependant, il n'y a pas de raison en général pour qu'une solution construite en ne faisant que des choix locaux soit un optimum global. On a essentiellement trois situations possibles :

— Il existe un algorithme glouton pour trouver la solution optimale : cet algorithme est alors le plus souvent le meilleur pour le problème considéré, c'est-à-dire le plus efficace.
— Il existe un algorithme glouton donnant une solution sub-optimale, mais « pas trop mauvaise » : si le problème est trop difficile à résoudre exactement, cet algorithme peut être très utile en pratique.
— Les algorithmes gloutons donnent des solutions inacceptables : on se débrouille autrement !

3.1 Premier exemple : le rendu de monnaie

On appelle *système monétaire* un ensemble $S \coloneqq \{a_1, \dots, a_p\}$ d'entiers avec $1 = a_1 < a_2 < \cdots < a_p$: les a_i sont les valeurs des pièces ou billets disponibles (par exemple $1, 2, 5, 10, 20, 50, 100, 200$ centimes d'euro). On cherche une méthode permettant à un commerçant de rendre la monnaie à un client en utilisant le moins de pièces possible. Plus formellement, étant donné un entier n, on cherche à déterminer un p-uplet $(x_1, \dots, x_p)$ tel que

$$\sum_{i=1}^{p} x_i a_i = n \qquad \text{et} \qquad \sum_{i=1}^{p} x_i \text{ est minimal.}$$

L'algorithme glouton que nous allons considérer pour résoudre ce problème est le suivant : utiliser à chaque étape la plus grande pièce possible.

Exercice 5

⇨ 1. Écrire une fonction C ayant pour prototype :

```c
int *greedy_change(int system[], int p, int n)
```

— `system` est un tableau d'entiers codant le système monétaire.
— `n` est le nombre de pièces présentes dans le système.
— `target` est la somme à atteindre.

Les *préconditions* à cette fonction sont
— `system` est de longueur `p`.
— `system` est trié par ordre croissant.
— `target` est positif ou nul.

La fonction doit renvoyer un pointeur vers un tableau de taille `p` contenant en case i le nombre de pièces de valeur `system[i]` dans la solution gloutonne. S'il est impossible d'obtenir `n` avec le système donné, on le signalera au moyen d'une assertion.

2. Déterminer la complexité de cette fonction.

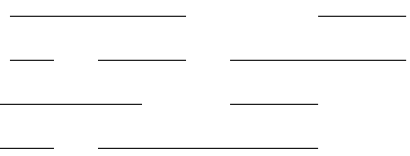
Pour un certain nombre de systèmes monétaires (dont le système $1, 2, 5, 10, 20, 50 \dots$), cet algorithme est optimal : ces systèmes sont dits *canoniques*. En revanche, en prenant le système $1, 7, 8$ et $n = 14$, cet algorithme renvoie $(8, 1, 1, 1, 1, 1, 1)$ alors que la solution optimale est $(7, 7)$. Dans ce cas, il faut utiliser une solution par programmation dynamique.

3.2 Exemple de solutions exactes : problèmes d'ordonnancement

La classe générale des problèmes d'*ordonnancement* (*scheduling* en anglais) est celle où l'on dispose d'un certain nombre de ressources (processeurs, employés, salles de cours) et où l'on a un certain nombre de tâches à accomplir. Typiquement, ces tâches ont une durée spécifiée, éventuellement un horaire de début et de fin, ou peut-être une *deadline*, et surtout elles monopolisent une ressource pendant qu'on les traite. On souhaite trouver une répartition optimale des tâches sur les ressources, pour une certaine définition de « optimale ».

3.2.1 Interval Scheduling

— *Entrées* : Un ensemble non vide d'intervalles non vides $\mathcal{S} \coloneqq \{]a_i, b_i[: 0 \leqslant i \leqslant n-1\}$. On parlera de requête numéro i pour l'intervalle $]a_i, b_i[$, et l'on notera $s(i) = a_i$ et $f(i) = b_i$ (*start* et *finish*).
— *Sortie* : Le plus grand sous-ensemble de $\mathcal{S}$, au sens de la cardinalité, constitué d'intervalles deux à deux disjoints. On parlera de *requêtes compatibles*.
— *Algorithme glouton* : À chaque étape, on choisit une requête compatible avec celles déjà choisies suivant une certaine règle, et l'on élimine les requêtes devenues incompatibles.



Une instance du problème Interval Scheduling.

La figure de gauche présente une solution non optimale et celle de droite présente une solution optimale.

Plusieurs choix de règles sont envisageables :

1. Prendre la requête commençant le plus tôt.
2. Prendre la requête la plus courte, c'est-à-dire minimisant $b_i - a_i$.
3. Prendre la requête ayant le moins de conflits avec des requêtes non encore choisies.
4. Prendre la requête se terminant en premier.

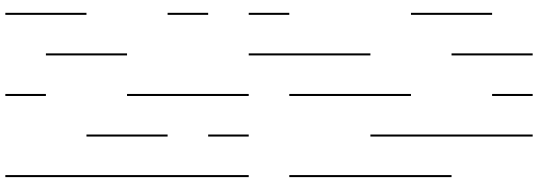
Exercice 6

⇨ 1. Montrer que les trois premières règles ne fournissent pas, en général, une solution optimale.
2. *Optimalité de l'algorithme glouton* : On note $G \coloneqq (i_1, \dots, i_m)$ les requêtes choisies par l'algorithme glouton suivant la règle 4, classées dans l'ordre croissant (de leurs horaires de fin ou de début, cela revient au même). On considère un autre ensemble $H \coloneqq (j_1, \dots, j_p)$ de requêtes compatibles, également classées, avec $p \geqslant m$.
   (a) Montrer par récurrence sur $r \leqslant m$ la propriété $\mathcal{P}_r$: « $f(i_r) \leqslant f(j_r)$ ».
   (b) Conclure.
3. *Complexité de l'algorithme glouton* : Justifier que l'on peut implémenter l'algorithme glouton avec la règle 4 en temps $\mathrm{O}(n \log n)$.

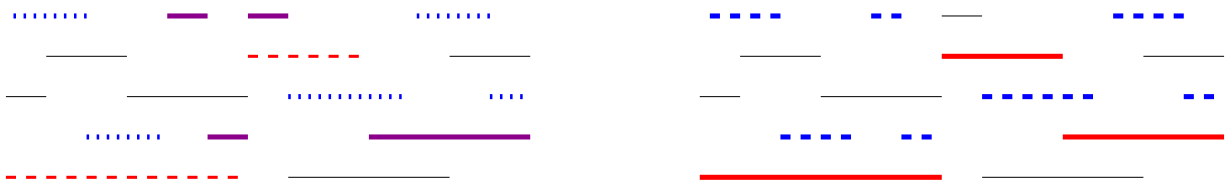
3.2.2 Interval Partitioning

— *Entrées* : Un ensemble non vide d'intervalles non vides $\mathcal{S} \coloneqq \{[a_i, b_i[: 0 \leqslant i \leqslant n-1\}$. On parlera de requête numéro i pour l'intervalle $]a_i, b_i[$, et l'on notera $s(i) = a_i$ et $f(i) = b_i$ (*start* et *finish*).
— *Sortie* : Une partition de $\mathcal{S}$ en k groupes de requêtes, chacun des groupes devant être compatibles et k devant être minimal.

Essentiellement, la question posée est : « étant données ces n conférences ayant chacune un horaire de début et de fin fixé, de combien de salles a-t-on besoin ? ».



Une instance du problème IntervalPartitioning.



À gauche, une solution non optimale du problème d'interval partitioning et à droite, une solution optimale.

Définition 3.1

Une *clique d'intervalles* est un ensemble d'intervalles deux à deux non disjoints, c'est-à-dire un ensemble $\{I_1, \dots, I_k\}$ tel que $I_i \cap I_j \neq \varnothing$ pour tous $1 \leqslant i, j \leqslant k$. On note $\chi(\mathcal{S})$ le cardinal de la plus grande clique de $\mathcal{S}$.

Proposition 3.2

Tout ordonnancement valide d'un ensemble $\mathcal{S}$ de requêtes utilise au moins $\chi(\mathcal{S})$ ressources.

On considère l'algorithme (glouton) suivant :
— Trier les requêtes dans un certain ordre (précisé ultérieurement).
— Programmer la requête r_1 sur la ressource 1.
— $d \leftarrow 1$ (d contiendra toujours le nombre de ressources allouées jusqu'à présent).
— Pour i de 2 à n :
  — Chercher le plus petit $k \leqslant d$ tel que l'on puisse programmer la requête i sur la ressource k.
  — Si l'en existe un :
    — programmer la requête i sur la ressource k.
  — Sinon :
    — programmer r_i sur $d+1$;
    — $d \leftarrow d+1$.

Exercice 7

⇨ 1. (a) Montrer que cet algorithme renvoie une solution compatible.
   (b) Exhiber une instance et un ordre de traitement des requêtes pour lesquels la solution renvoyée n'est pas

optimale.

(c) On suppose désormais que le tri des requêtes se fait *par ordre de début croissant*. Montrer que la solution renvoyée est optimale.

2. Montrer que l'on peut implémenter cet algorithme, ou une variante qui lui équivaut, avec une complexité temporelle en $O(n \log n)$.

3.2.3 Retard minimum

— *Problème* : On considère un ensemble de n requêtes, ayant chacune une durée d_i et un horaire de fin souhaité (une *deadline* pour faire court) h_i. Si la requête i se termine finalement au temps t, son *retard* est $l_i := \max(0, t - h_i)$. On ne dispose que d'une ressource, les requêtes doivent donc être programmées pour s'exécuter pendant des intervalles de temps disjoints, en commençant à l'instant $t = 0$. Le but est de minimiser le retard maximal

$$l = \max_{1 \leqslant i \leqslant n} l_i.$$

— *Algorithme glouton* : On programme les requêtes par *deadlines* croissantes et sans temps mort.

Proposition 3.3

L'algorithme glouton donne une solution optimale.

Exercice 8

⇨ On considère une instance du problème de taille n, et pour une solution $x_1, \ldots, x_n$ on note $s(x_i)$ l'horaire de début de x_i, $d(x_i)$ la durée de x_i et $h(x_i)$ l'horaire de fin souhaité pour x_i. On supposera toujours que $s(x_1) \leqslant s(x_2) \ldots \leqslant d(x_n)$. Autrement dit, on numérote les requêtes dans l'ordre dans lequel on les a planifiées.

1. Une solution $x = (x_1, \ldots, x_n)$ est dite *sans temps mort* si $s(x_i) + d(x_i) = s(x_{i+1})$ pour $1 \leqslant i < n$. Montrer que l'on peut se limiter à considérer des solutions sans temps mort. Une solution optimale est-elle nécessairement sans temps mort ?
2. On considère une solution sans temps mort x, et l'on suppose qu'il existe $i \in [1 \ldots n-2]$ tel que $h(x_i) > h(x_{i+1})$. Montrer que la solution y (sans temps mort) obtenue en inversant les requêtes x_i et x_{i+1} vérifie $l(y) \leqslant l(x)$.
3. Conclure.

4 Backtracking

4.1 L'exemple des n dames

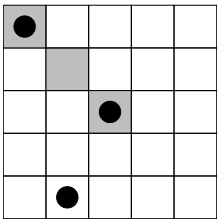
On considère un échiquier de taille $n \times n$ et l'on souhaite y placer n dames de manière à ce qu'aucune d'entre elles ne soit en prise, c'est-à-dire sur la même ligne, colonne ou diagonale qu'une autre dame.

Exercice 9

⇨ 1. Combien y a-t-il de manières de placer n dames sur un échiquier $n \times n$, en imposant qu'il n'y ait pas plus d'une dame par case ? Est-il raisonnable d'essayer toutes ces possibilités pour $n = 4$? $n = 8$? $n = 20$?
2. Même question en imposant qu'il n'y ait qu'une seule dame par colonne.
3. Même question en imposant de plus qu'il n'y ait qu'une seule dame par ligne et par colonne.

4.1.1 Principe de la méthode

Le principe du *backtracking* (ou *retour sur trace*) s'applique à des problèmes de recherche exhaustive pour lesquels on peut facilement déterminer si une solution partielle a une chance de pouvoir donner une solution complète du problème. Ici, une solution partielle dans laquelle on a déjà placé k dames peut être éliminée si l'une des dames est « en prise », c'est-à-dire sur la même ligne, colonne ou diagonale qu'une autre dame déjà placée.

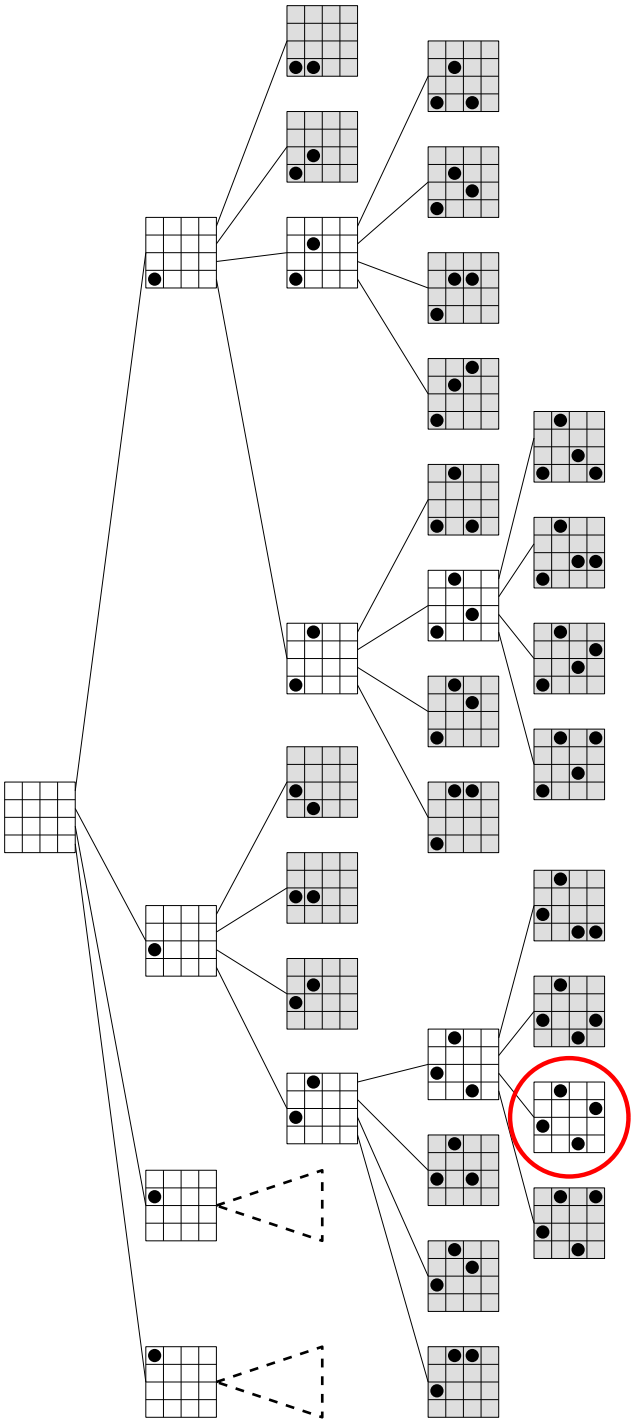


Une solution partielle à rejeter.

Une bonne manière de raisonner est en termes d'arbres : un nœud interne de profondeur k correspond au placement des k premières dames (de manière correcte ou non), ses enfants aux différentes manières de rajouter la $(k+1)$-ème dame. Les feuilles sont les solutions complètes (n dames placées), dont seulement certaines sont correctes. L'idée du

backtracking est de réaliser un parcours en profondeur de cet arbre tout en l'*élaguant*, c'est-à-dire en coupant une branche dès que l'on sait qu'elle ne peut contenir de solution correcte.

— On sait qu'une solution correcte contient exactement une dame sur chaque colonne : un *candidat* sera une liste $(l_0, \ldots, l_{k-1})$, où les l_i sont dans $[\![0, n[\![$ et indiquent sur quelle ligne se trouve la dame de la colonne i. La position de la figure ci-dessus, par exemple, serait codée $(4, 0, 2)$.
— Un candidat sera dit *complet* lorsque $k = n$ (toutes les dames sont placées), *partiel* ou *incomplet* sinon.
— Les enfants d'un candidat partiel $(l_0, \ldots, l_{k-1})$ sont les candidats de la forme $(l_0, \ldots, l_{k-1}, x)$ avec $0 \leqslant x < n$.
— Une *extension* d'un candidat $(l_0, \ldots, l_{k-1})$ est un candidat de la forme $(l_0, \ldots, l_{k-1}, x_1, \ldots, x_p)$.
— Un candidat est *rejetable* si aucune solution correcte ne l'étend.
— La manière la plus simple de procéder ici est de rejeter un candidat si et seulement si il contient deux dames qui se menacent mutuellement.



Arbre exploré pour le problème des 4 dames. Seule la moitié a été représentée,
l'autre est symétrique. La solution est entourée (il y en a une autre dans la partie symétrique),
les candidats complets ou non rejetés sont grisés.

Exercice 10

⇨ 1. Compter le nombre de nœuds internes l'arbre de la figure ci-dessus, et en déduire son nombre total de nœuds.
2. Si l'on utilisait une règle de rejet triviale où on ne rejette jamais un candidat partiel, mais uniquement un

candidat complet qui n'est pas solution, combien de nœuds l'arbre posséderait-il ?

3. Si l'on utilisait une règle de rejet parfaite, c'est-à-dire si on est capable de détecter magiquement si un candidat est rejetable, combien de nœuds l'arbre posséderait-il ?

4.1.2 Programmation *ad hoc*

— Pour programmer une solution au problème, il faut déjà choisir une représentation des candidats. On peut par exemple utiliser un tableau de taille k pour un candidat dans lequel k dames ont été placées, la case i indiquant la ligne dans laquelle se trouve la dame de la colonne i. Il s'agira donc d'un `int array` dont les valeurs seront comprises entre 0 et $n-1$.
— Un candidat est alors complet si et seulement si sa longueur est n.
— Il nous faut une fonction de rejet : on va tester si la dernière dame ajoutée est en conflit avec l'une des autres. On maintiendra comme invariant le fait que si le candidat `c` est de longueur k, alors il n'y a pas de conflit entre les dames situées dans les colonnes $0, \ldots, k-2$.

Exercice 11

⇨ Écrire une fonction `conflit_derniere` qui prend en entrée un candidat de longueur k et renvoie un booléen indiquant si la dernière dame du candidat est en conflit avec l'une des autres.

```
conflit_derniere : int array -> bool
```

Pour ajouter une dame à un candidat partiel, on peut utiliser la fonction suivante :

```
let etend (c : int array) (x : int) : int array =
  Array.append c [| x |]
```

Remarque

⇨ Cette fonction renvoie un *nouveau tableau*, dont la taille est un plus celle de `candidat`, et qui contient la valeur `x` dans sa dernière case ainsi qu'une copie de `candidat` dans les cases précédentes.

Ensuite, il faut décider quel problème précis on souhaite résoudre. Il y a quatre variantes principales :
— Déterminer s'il existe au moins une solution.
— Compter le nombre de solutions.
— Exhiber une solution s'il en existe une et signaler qu'il n'en existe pas dans le cas contraire.
— Déterminer la liste des solutions.

La version la plus simple à programmer est sans doute celle comptant les solutions :

```
let compte_n_reines (n : int) : int =
  let rec compte_extensions (c : int array) : int =
    if conflit_derniere c then
      0
    else if Array.length c = n then
      1
    else
      let s = ref 0 in
      for i = 0 to n - 1 do
        s := !s + compte_extensions (etend c i)
      done;
      !s
  in
  compte_extensions [||]
```

```
# compte_n_reines 10;;
- : int = 724 (* instantané *)
# compte_n_reines 12;;
- : int = 14200 (* quelques secondes *)
```

Exercice 12

⇨ 1. Écrire une fonction `liste_n_reines` renvoyant la liste des solutions au problème des n reines.
2. Écrire une fonction `existe_n_reines` renvoyant un booléen indiquant s'il existe au moins une solution au problème des n reines (on s'arrêtera dès que possible).
3. Écrire une fonction `solution_n_reines` renvoyant `Some c`, où `c` est une solution, s'il existe au moins une solution, `None` sinon.

```
liste_n_reines : int -> int array list
existe_n_reines : int -> bool
```

```
solution_n_reines : int -> int array option
```

4.1.3 Programmation générique

Il est possible d'abstraire le principe de la méthode pour pouvoir l'appliquer sans modification à différents problèmes. Pour ce faire, on définit deux types :

```
type 'a reponse =
  | Refus
  | Accepte of 'a
  | Partiel of 'a

type 'a probleme = {
  accepte : 'a -> 'a reponse;
  enfants : 'a -> 'a list;
  initiale : 'a;
}
```

- Le type `'a` correspond à un candidat : il dépendra du problème considéré. Pour notre solution au problème des n reines, il serait égal à `int array`.
- Un problème est spécifié par la donnée :
  - d'un type `'a` pour les candidats.
  - d'un candidat initial (qui serait `[||]` pour les n reines).
  - d'une fonction `enfants` prenant en entrée un candidat `c` et renvoyant la liste de ses enfants.
  - d'une fonction `accepte`, qui prend en entrée un candidat `c` et qui a la spécification suivante :
    - si `accepte c` vaut `Refus`, on est sûr que `c` ne peut être étendu en une solution.
    - si `accepte c` vaut `Accepte s`, alors `s` est l'unique solution qui étend au sens large `c`.
    - si `accepte c` vaut `Partiel c'`, alors `c'` est un candidat partiel qui étend au sens large `c`.

    Le plus souvent, `s` et `c'` seront égaux à `c`. Cependant, il est parfois intéressant de combiner le *backtracking* avec des phases déductives. Par exemple, si l'on programme la résolution d'un Sudoku, la fonction `accepte` pourra commencer par calculer une grille `c'` à partir de `c` en utilisant des méthodes déductives, puis renvoyer `Refus`, `Accepte c'` ou `Partiel c'` suivant que `c'` est visiblement impossible, complète et correcte ou incomplète et pour l'instant correcte.

Exercices 13

⇨ Écrire une fonction `n_reines` prenant en entrée un entier n et renvoyant un `int array problem` codant le problème des n reines. On pourra réutiliser les fonctions `etend` et `conflit_derniere`.

```
n_reines : int -> int array problem
```

⇨
1. Écrire une fonction `enumere_solutions` qui prend en entrée un `'a probleme` et renvoie la `'a list` de ses solutions.
2. Écrire une fonction `cherche_solution` qui prend en entrée un `'a probleme` et renvoie `Some s`, où `s` est une solution, s'il en existe une et `None` sinon.

```
enumere_solutions : 'a probleme -> 'a liste
cherche_solution : 'a probleme -> 'a option
```

4.2 Remarques générales

4.2.1 Choix des enfants

Pour l'instant, on a fixé une fois pour toutes l'ordre dans lequel les choix seront faits (on place systématiquement les dames de gauche à droite) : c'est le plus simple mais rarement le plus efficace. Imaginons par exemple qu'on résolve une grille de Sudoku, sans aucun raisonnement déductif :

1234	23	123	2
123	23	123	4
23	4	2	1
2	1	24	3

Une grille de Sudoku avec pour chaque case les enfants qui ne seront pas immédiatement rejetés.

Si l'on remplit les cases dans l'ordre « naturel » (de gauche à droite et de haut en bas), on va commencer par essayer de placer un 1 dans la case en haut à gauche et mettre longtemps avant de remarquer qu'aucune solution n'étend ce candidat. On essaiera ensuite 2, puis 3, et finalement 4 (qui est la bonne valeur). Il aurait été bien plus intéressant de détecter les cases ayant peu d'enfants potentiellement valides (il y a ici trois cases avec un seul enfant acceptable, mais même des cases donnant deux branches sont préférables à une qui en donne quatre).

Bien sûr, détecter ce genre de choses va conduire à passer plus de temps dans chaque nœud de l'arbre, mais en échange l'arbre aura moins de nœuds (potentiellement *beaucoup moins*) : on y gagne souvent, en particulier pour les nœuds peu profonds.

4.2.2 Candidat mutable

Dans ce que nous avons présenté, les enfants d'un candidat sont des objets complètement indépendants du candidat. Il est parfois plus efficace de n'avoir qu'un seul candidat, que l'on mute :

— soit en ajoutant une valeur pour passer d'un nœud à un enfant.

— soit en modifiant une valeur pour passer d'un enfant au suivant.

Il faut faire très attention dans ce cas à ne pas oublier le troisième type de modification :

— enlever une valeur quand on passe d'un nœud à son père (parce que l'exploration du nœud est terminée).

Le problème du Sudoku consiste à remplir une grille 9×9 en utilisant les chiffres de 1 à 9 eet en obéissant aux contraintes suivantes : chaque ligne, chaque colonne et chaque sous-groupe de 3×3 doit contenir exactement une seule occurence de chaque chiffre. Le problème est en général posé sous la forme d'une grille où certaines cases sont déjà remplies. La figure suivante contient un exemple de problème (à gauche), ainsi qu'une solution (à droite). Les sous-groupes 3×3 y sont délimités en gras.

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

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

Nous allons résoudre le problème du Sudoku par backtracking avec une implémentation en C. Pour représenter une grille partiellement remplie de Sudoku, on va utiliser un tableau c de 81 cases dans lequel on va stocker les valeurs des différentes cases en « *row major order* », c'est-à-dire en y plaçant les 9 valeurs présentes dans la première ligne, puis les valeurs de la seconde ligne, jusqu'à la 9ème et dernière ligne. Lorsqu'une case est vide, on utilisera la valeur 0. Pour faciliter l'implémentation, on va numéroter les lignes de i de 0 à 8 ainsi que les colonnes j de 0 à 8. Une case aura un index $k \in [\![0, 81[\![$ dans le tableau c. Enfin, les 9 groupes de la grille seront numérotées de 0 à 8 en « row major order ». On commence par écrire des fonctions permettant d'obtenir la ligne, la colonne ainsi que la zone d'une case indexée par k dans le tableau c.

```c
int row(int k) {
    return k / 9;
}

int column(int k) {
    return k % 9;
}

int group(int k) {
    return 3 * (row(k) / 3) + column(k) / 3;
}
```

On en déduit facilement une fonction `same_zone` permettant de savoir si deux cases indexées respectivement par k_1 et k_2 dans c sont dans la même ligne, dans la même colonne ou dans le même groupe.

```c
bool same_zone(int k1, int k2) {
    return row(k1) == row(k2) ||
           column(k1) == column(k2) ||
           group(k1) == group(k2);
}
```

Pour interrompre prématurément notre recherche d'une solution, on écrit une fonction booléenne `check` qui vérifie si la case d'indice k contient une valeur qui est en conflit avec une autre case. Pour cela, on parcourt toutes les cases du tableau.

```
bool check(int c[81], int k) {
    for (int p = 0; p < 81; ++p) {
        if (k != p && same_zone(k, p) && c[k] == c[p]) {
            return false;
        }
    }
    return true;
}
```

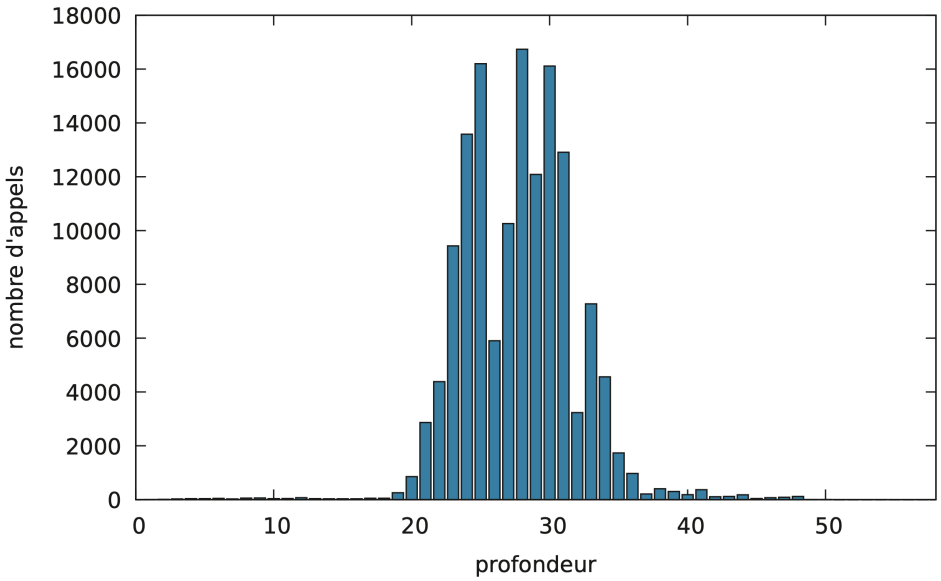
On peut désormais écrire notre fonction `solve` à qui on passe un candidat c qui ne contient aucune case en conflit avec une autre et qui renvoie `true` s'il est possible de compléter le candidat en une solution au problème (avec cette solution contenue dans le tableau c) et `false` sinon.

```
bool solve(int c[81]) {
    for (int k = 0; k < 81; k++) {
        if (c[k] == 0) {
            for (int v = 1; v <= 9; v++) {
                c[k] = v;
                if (check(c, k) && solve(c)) {
                    return true;
                }
            }
            c[k] = 0;
            return false;
        }
    }
    return true;
}
```

Un tel code résout notre problème de Sudoku en un dixième de seconde. En testant de nombreux problèmes de Sudoku, on se rend compte qu'il faut en moyenne un tiers de seconde pour résoudre une grille.

L'analyse de la complexité d'un tel programme est difficile mais on peut faire quelques observations empiriques. On peut voir var exemple que la résolution de notre problème nécesssite 142 256 appels à `solve`. C'est très peu par rapport à l'espace total de recherche. En effet, on est parti d'une grille avec 58 cases vides, avec 9 valeurs pour chacune des cases. La taille de l'espace de recherche est donc de $9^{58} \approx 2.22 \times 10^{55}$. On comprend en particulier qu'une recherche brutale, testant systématiquement chaque n-uplet de valeurs possibles pour les cases vides n'a aucune chance de terminer dans un délai raisonnable. Encore une fois, c'est l'interruption prématurée de la recherche, dès lors que la valeur placée est incompatible avec les valeurs déjà présentes, qui fait la force de notre algorithme.

On comprend un peu mieux ce qui se passe si on observe comment se répartissent les appels à `solve` selon la profondeur dans la pile d'appels. Ici, la profondeur varie entre 0 pour le tout premier appel et 58 pour une grille qui est déjà pleine.



On peut constater qu'il y a peu d'appels à des profondeurs faibles, ce qui était attendu. En effet, à la profondeur 0, les seuls appels possibles sont pour remplir la première case vide. Il y a donc au plus 9 appels possibles. Plus généralement, pour les faibles profondeur, il y a peu d'appels possibles car la combinatoire du problème est encore trop faible. On voit aussi peu d'appels dans des profondeurs élevées car il y a peu de remplissages de grilles où l'on se rend compte vers la fin qu'il n'est pas possible de le compléter en une solution. C'est grâce à cela que notre algorithme de backtracking est beaucoup plus efficace qu'une recherche par force brute.

Remarquons enfin que même si il devait tourner très longtemps, notre programme ne consomme que très peu de mémoire. En particulier, il ne fait aucune allocation sur le tas. Il n'y a de plus aucun risque de faire déborder la pile d'appels car il n'y a jamais plus de 58 appels imbriqués.

Exercice 14

⇨ Écrire une fonction `int count_solutions(int c[81])` qui compte le nombre de manières de compléter une grille en une solution.

4.2.3 Complexité

Il est très délicat d'estimer la complexité d'un algorithme de *backtracking* : en règle générale, il n'est pas difficile de donner un majorant de la taille de l'arbre, mais l'élagage réalisé va dépendre fortement de l'instance et de l'algorithme précis (choix des enfants, ordre d'exploration, qualité de la fonction d'acceptation/rejet, etc). Même une estimation dans le pire cas est délicate : rien ne dit qu'on puisse trouver une instance pour laquelle on est amené à explorer tout l'arbre ou presque, sans pouvoir réaliser d'élagage.