Le paradigme « Diviser pour régner » consiste à transformer la résolution d'un problème de taille n en la résolution d'un ou de plusieurs problèmes problèmes de taille $\sim \frac{n}{k}$. Il permet d'améliorer grandement la complexité temporelle dans une majorité des cas. Ce principe a déjà été rencontré dans l'étude des arbres binaires.

I Deux exemples classiques

I.1 Exponentiation rapide

Calculer la puissance x^n d'un entier x peut se faire simplement avec la fonction récursive suivante :

```
let rec puissance x = function
   | 0 -> 1
   | 1 -> x
   | n -> x * puissance x (n - 1);;
```

Une analyse rapide de la complexité montre que cette fonction est de complexité $O(n)$.

Cependant, il est possible d'exploiter le fait que : $\left\{ \begin{array}{rcl} x^{2k} & = & (x^2)^k \\ x^{2k+1} & = & x \times (x^2)^k \end{array} \right.$ pour obtenir la fonction suivante :

```
let rec puissance_rapide x = function
   | 0 -> 1
   | 1 -> x
   | n when n mod 2 = 0 -> puissance_rapide (x * x) (n / 2)
   | n -> x * puissance_rapide (x * x) (n / 2);;
```

Le calcul de la complexité est alors le suivant :

- $C(0) = C(1) = O(1) = k_0$
- $C(n) = C\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + k_1$

On considérant l'écriture en binaire de n, il est facile de montrer qu'on arrive à une complexité de la forme $C(n) = k_0 + k_1 \times \lfloor \log_2(n) \rfloor = O(\log n)$.

Exercice 1

1. Combien de multiplications sont nécessaires pour calculer x^{15} avec l'algorithme d'exponentiation rapide ?
2. Est-il possible de faire mieux ?

I.2 Dichotomie

« Je pense à un nombre entre 1 et 100. Propose un nombre et je te dirais si c'est plus petit ou plus grand. » Un jeu simple qui cache un principe fondamental en informatique : la recherche d'éléments. Si on dispose d'un tableau quelconque, tester si un élément est dans le tableau ou non est un classique. On y répond facilement en temps linéaire :

```
let index x t =
   let i = ref (-1) in
   for j = 0 to Array.length t - 1 do
      if t.(j) = x then i := j
   done;
   !i;;
```

Cependant, si on suppose l'hypothèse supplémentaire que le tableau est trié, la recherche peut se faire de manière dichotomique : lorsqu'on cherche dans une partie du tableau, on regarde la valeur du milieu et selon

qu'elle soit plus grande ou plus petite que la valeur cherchée, on sait dans quelle partie on doit continuer la recherche.

```
let dichotomie x t =
   let imin = ref 0 and imax = ref (Array.length t) in
   while !imax - !imin > 1 do
      let imid = (!imin + !imax) / 2 in
      if x < t.(imid) then imax := imid
      else imin := imid
   done;
   if t.(!imin) = x then !imin else -1;;
```

L'analyse de complexité se fait sur la différence $n = imax - imin$. On fait alors un nombre logarithmique en n de passages dans la boucle `while`, et chaque passage est de complexité constante. La complexité totale est à nouveau $O(\log n)$.

Exercice 2

On suppose qu'un élément peut éventuellement apparaître plusieurs fois dans le tableau. Modifier la fonction précédente pour renvoyer l'indice de la première occurrence de x. Est-il possible de compter le nombre d'occurrence de x dans le tableau en temps logarithmique ?

II Complexité

Pour certaines relations de récurrence, une formule générale permet de calculer la complexité d'une fonction utilisant un principe « diviser pour régner ». L'énoncé est le suivant :

Théorème A : Master Theorem

Considérons la relation de récurrence :

$$C(n) = aC\left(\frac{n}{b}\right) + \Theta(n^c)$$

où a, b et c sont des réels positifs, avec $a > 1$ et $b > 1$. Alors :

(a) Si $c < \log_b a$: $C(n) = \Theta(n^{\log_b a})$.

(b) Si $c > \log_b a$: $C(n) = \Theta(n^c)$.

(c) Si $c = \log_b a$: $C(n) = \Theta(n^{\log_b a} \log n)$.

Remarque B

On peut se permettre d'omettre les parties entières dans les calculs d'ordres de grandeur de complexité. Il est cependant nécessaire de les laisser lorsqu'on calcule un nombre exact d'une certaine opération (par exemple nombre de multiplications).

Exercice 3

Déterminer les complexités vérifiant les relations de récurrence suivantes :

1. $C(n) = 3C\left(\frac{n}{2}\right) + \Theta(n)$
2. $C(n) = 16C\left(\frac{n}{4}\right) + \Theta(n^2)$
3. $C(n) = 5C\left(\frac{n}{3}\right) + \Theta(n^3)$
4. $C(n) = 7C\left(\frac{n}{6}\right) + \Theta(n \log n)$
5. $C(n) = 49C\left(\frac{n}{25}\right) + \Theta(n^{1,5} \log n)$

6. $C(n) = C(n-1) + \Theta(n^3)$
7. $C(n) = C(\sqrt{n}) + \Theta(1)$

Exercice 4

On cherche à montrer le théorème précédent.

1. On suppose que $C(n) = aC\left(\frac{n}{b}\right) + dn^c$, où d est un réel positif. Montrer que $C(n) = d\sum_{i=0}^{\log_b n} a^i \left(\frac{n}{b^i}\right)^c + \Theta(n^{\log_b a})$.
2. On suppose que $c = \log_b a - \varepsilon$, avec $\varepsilon > 0$. Montrer que $\sum_{i=0}^{\log_b n} a^i \left(\frac{n}{b^i}\right)^c \leq n^{\log_b a} \frac{b^\varepsilon}{b^\varepsilon - 1}$.
   En déduire le point (a) du théorème.
3. On suppose que $c > \log_b a$. Montrer que $\sum_{i=0}^{\log_b n} a^i \left(\frac{n}{b^i}\right)^c \leq n^c \frac{b^c}{b^c - a}$. En déduire le point (b) du théorème.
4. Montrer le point (c) du théorème.

Exercice 5

L'algorithme de Karatsuba est un algorithme permettant le calcul du produit de deux polynômes. Pour la suite de l'exercice, on considèrera qu'un polynôme s'implémente sous la forme du tableau de ses coefficients, c'est-à-dire que le polynôme $P = \sum_{i=0}^{n} a_i X^i$ s'implémente par le tableau $[|a_0; \ldots; a_n|]$.

```
type poly = int array;;
```

1. Écrire une fonction `somme : poly -> poly -> poly` qui calcule la somme de deux polynômes. Écrire de même une fonction `sous` qui calcule la soustraction.
2. Écrire une fonction naïve `prod : poly -> poly -> poly` qui calcule le produit de deux polynômes de degré quelconque.
3. Déterminer la complexité de la fonction précédente en fonction du degré des polynômes.
4. Écrire une fonction `prod_xk : poly -> int -> poly` qui prend en argument un polynôme P et un entier k et calcule $X^k \times P$. Cette fonction doit être de complexité $O(\deg(P) + k)$.
5. On suppose que $P = P_0 + P_1 \times X^k$ et $Q = Q_0 + Q_1 \times X^k$. Déterminer un moyen de calculer PQ en effectuant 3 multiplications de polynômes non triviaux (on supposera que la multiplication par X^k n'est pas coûteuse en temps). Il sera nécessaire d'effectuer des additions supplémentaires.
6. En déduire un algorithme récursif de type « Diviser pour régner » permettant de calculer le produit de deux polynômes. On pourra supposer que les deux polynômes sont de même degré $n = 2^k - 1$.
7. Déterminer la complexité de cette algorithme.
8. Écrire une fonction `kara : poly -> poly -> poly` implémentant cet algorithme.

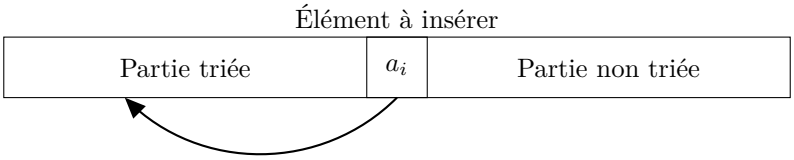
III Algorithmes de tris

III.1 Tris élémentaires

III.1.1 Tri par insertion

Pour imaginer le fonctionnement du tri par insertion (*insertion sort*), imaginons un joueur de jeu de cartes, à qui les cartes sont données petit à petit. Pour gagner du temps, le joueur trie ses 3 premières cartes, puis rajoute les suivantes dans la partie de son jeu qui est déjà triée. Il lui suffit alors de chercher l'emplacement pour rajouter une nouvelle carte, et de l'y insérer.

Schématiquement, chaque étape du tri ressemble à :



Exercice 6

1. Écrire une fonction récursive `insertion : 'a -> 'a list -> 'a list` qui prend en argument une liste ℓ **triée** et un élément x et renvoie une nouvelle liste où l'on a inséré l'élément x à la bonne place dans la liste.
2. En déduire une fonction `tri_insertion : 'a list -> 'a list` qui prend en argument une liste ℓ et renvoie une liste triée des éléments de ℓ.

Proposition C

Le tri par insertion effectue de l'ordre de $\dfrac{n^2}{2}$ comparaisons dans le pire des cas.

Preuve

Dans le pire des cas, une insertion effectue k comparaisons où k est la taille de la liste où on fait l'insertion. Le nombre total de comparaisons est alors $\displaystyle\sum_{k=1}^{n-1} k = \frac{n(n-1)}{2} \sim \frac{n^2}{2}$.

Proposition D

Le tri par insertion effectue $n-1$ comparaisons dans le meilleur des cas.

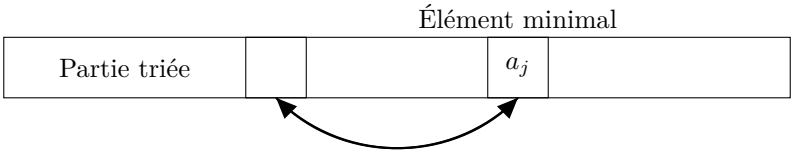
Exercice 7

Déterminer des listes de taille n représentant le meilleur et le pire des cas.

III.1.2 Tri par sélection

Pour comprendre le fonctionnement de l'algorithme de tri par sélection (*selection sort*), imaginons une photo de classe. Le photographe veut que tout le monde soit bien visible, et place chaque personne une par une. Naturellement, il commence par les plus petits, qu'il place devant, et à chaque étape, il choisit le prochain élève le plus petit pour le placer.

Schématiquement, chaque étape du tri ressemble à :



Exercice 8

On considère la fonction usuelle permettant de permuter des éléments dans un tableau :

```
let swap t i j =
   let x = t.(i) in
   t.(i) <- t.(j); t.(j) <- x;;
```

1. Écrire une fonction `minimum : 'a array -> int -> int` qui prend en argument un tableau t et un indice i et renvoie l'indice de l'élément minimum dans le sous-tableau commençant à l'indice i.
2. En déduire une fonction `tri_selection : 'a array -> unit` qui trie un tableau.

Proposition E

Le tri par sélection effectue de l'ordre de $\frac{n^2}{2}$ comparaisons et $n-1$ permutations dans tous les cas.

Preuve

La première boucle est de taille n, et la seconde de taille $n-i-1$. Le nombre de comparaisons est donc :
$\sum_{i=0}^{n-2}(n-i-1) = \frac{n(n-1)}{2} \sim \frac{n^2}{2}$. Il y a une permutation après chaque passage dans la première boucle, soit $n-1$.

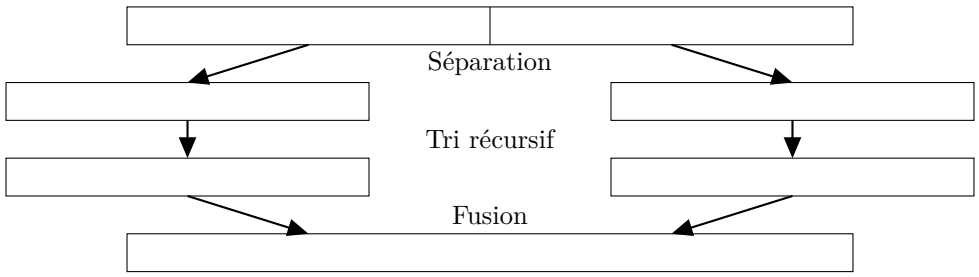
III.2 Tri efficaces

Pour améliorer les complexités quadratiques des algorithmes précédents, on utilise le paradigme « Diviser pour régner ».

III.2.1 Tri fusion

Pour illustrer le principe du tri fusion (*merge sort*), imaginons qu'un enseignant doive trier un tas de copies par ordre croissant de notes. Chance pour lui, deux élèves sont à la traîne pour quitter la salle de classe, et il les sollicite pour ce travail. Il donne la moitié du tas à l'un, l'autre moitié à l'autre, et leur demande de trier chacun sa partie. Une fois les deux parties triées, il est facile de recomposer le tas final en comparant à chaque fois le plus petit des deux. On peut imaginer un procédé similaire avec quatre élèves disponibles, et ainsi de suite...

Schématiquement, le principe est le suivant :



Exercice 9

1. Écrire une fonction `separation : 'a list -> 'a list * 'a list` qui renvoie deux listes de tailles différentes d'au plus 1, dont l'union est la liste donnée en argument.
2. Écrire une fonction récursive `fusion : 'a list -> 'a list -> 'a list` qui prend en argument deux listes triés, et renvoie une liste triée contenant les éléments des deux listes.
3. En déduire une fonction récursive `tri_fusion : 'a list -> 'a list` effectuant le tri d'une liste.
4. Déterminer la complexité temporelle de la fonction `tri_fusion` pour des tableaux de taille une puissance de 2. Cet ordre de grandeur change-t-il en dehors des puissances de 2 ? Y a-t-il une différence entre le meilleur des cas, le cas moyen et le pire des cas ?
5. Déterminer la complexité spatiale du `tri_fusion`.

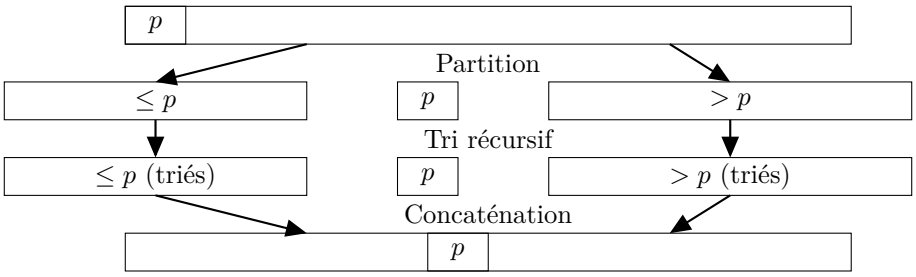
Exercice 10

(Difficile) Écrire une fonction qui calcule la médiane des valeurs de deux tableaux d'entiers non vides triés de taille respective m et n en complexité $O(m+n)$. Améliorer cette complexité en $O(\log(m+n))$ puis $O(\log(\min(m,n)))$.

III.2.2 Tri rapide

Le tri rapide (*quick sort*) utilise, comme le tri fusion, le paradigme diviser pour régner. Le principe est le suivant : on désigne un élément du tableau comme étant un *pivot*. On partitionne le tableau en trois parties : les éléments inférieurs au pivot, le pivot lui-même, et les éléments supérieurs. On trie ensuite les deux parties contenant les éléments non-pivot récursivement, qu'on concatène pour obtenir un tableau trié. Par défaut, le pivot sera le premier élément du tableau.

Schématiquement, on a :



Exercice 11

1. Reprendre la fonction `partition` vue dans l'exercice 7 du chapitre 5, et la modifier pour rajouter en argument des indices `debut` et `fin`, pour qu'elle partitionne le sous-tableau entre les indices `debut` (compris) et `fin` (non compris) par rapport à l'élément d'indice `debut`, et renvoie l'indice où cet élément a été placé.
2. En déduire une fonction récursive (ou utilisant une fonction auxiliaire récursive) de signature `tri_rapide : 'a array -> unit` qui effectue le tri d'un tableau.

Proposition F

L'algorithme de tri rapide effectue le tri d'un tableau de taille n en complexité $O(n^2)$ dans le pire des cas.

Preuve

Si on suppose qu'à chaque partitionnement, l'un des deux sous-tableaux soit vide, la formule de récurrence vérifiée par la complexité est : $C(n) = C(n-1) + O(n)$, soit $C(n) = O(n^2)$.

Remarque G

Ce cas se produit lorsque le tableau est déjà trié (dans un sens ou dans l'autre).

Proposition H

L'algorithme de tri rapide effectue le tri d'un tableau de taille n en complexité $O(n \log n)$ dans le meilleur des cas.

Preuve

Si on suppose qu'à chaque partitionnement, les deux tableaux sont de même taille, alors la formule de récurrence devient : $C(n) = 2C\left(\frac{n}{2}\right) + O(n)$, soit $C(n) = O(n \log n)$ d'après le théorème maître.

Proposition I

(*Admise*) L'algorithme de tri rapide effectue le tri d'un tableau de taille n en complexité $O(n \log n)$ en moyenne.

Exercice 12

Pour éviter des cas défavorables, on peut choisir le pivot aléatoirement dans le tableau. Modifier la fonction de partition pour implémenter cette fonction. On rappelle que `Random.int k` renvoie un entier choisi aléatoirement entre 0 et $k-1$.