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

Exercice 1

1. On a besoin d'une multiplication pour calculer : x^2, x^4, x^8, x^12, x^14 puis x^15, soit 6 multiplications au total.
2. Si on calcule x^2, x^3, x^6, x^12 puis x^15, on n'a besoin que de 5 multiplications.

I.2 Dichotomie

Exercice 2

Au lieu de faire en sorte que x soit toujours compris dans $[t.(imin); t.(imax)[$, on modifie pour qu'il se situe dans $]t.(imin), t.(imax)]$. On continuera ainsi à baisser l'indice $imax$, même en cas d'égalité entre x et $t.(imid)$.

```
let dichotomie x t =
   let imin = ref (-1)
   and imax = ref ((Array.length t) - 1) 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.(!imax) = x then !imax else -1;;
```

Pour compter le nombre d'éléments, il suffit de trouver le plus grand indice et le plus petit indice des éléments égaux à x et de faire la différence. Cela se fait bien en temps logarithmique.

II Complexité

Exercice 3

Déterminer les complexités vérifiant les relations de récurrence suivantes :

1. $C(n) = \Theta(n^{\log_2(3)})$
2. $C(n) = \Theta(n^2 \log n)$
3. $C(n) = \Theta(n^3)$
4. $C(n) = \Theta(n^{\log_6(7)})$
5. $C(n) = \Theta(n^{1,5} \log n)$
6. $C(n) = \Theta(n^4)$
7. $C(n) = \Theta(\log \log n)$

Exercice 4

1. Si on suppose que n est une puissance de b, alors on a :
$$\begin{aligned} C(n) &= aC\left(\frac{n}{b}\right) + dn^c \\ &= a^2C\left(\frac{n}{b^2}\right) + ad\left(\frac{n}{b}\right)^c + dn^c \\ &= \dots \\ &= a^{\log_b n}C(1) + a^{\log_b n}d\left(\frac{n}{b^{\log_b n}}\right)^c + \dots + dn^c \\ &= \Theta(n^{\log_b a}) + d\sum_{i=0}^{\log_b n} a^i\left(\frac{n}{b^i}\right)^c \end{aligned}$$

2.
$$\begin{aligned} \sum_{i=0}^{\log_b n} a^i\left(\frac{n}{b^i}\right)^c &= n^c\sum_{i=0}^{\log_b n}\left(\frac{a}{b^c}\right)^i \\ &= n^c\sum_{i=0}^{\log_b n}\underbrace{\left(\frac{a}{b^{\log_b a}}\right)^i}_{1}b^{i\varepsilon} \\ &= n^c\frac{b^{\varepsilon(\log_b n+1)}-1}{b^\varepsilon-1} \\ &= n^{\log_b a}n^{-\varepsilon}\frac{n^\varepsilon b^\varepsilon-1}{b^\varepsilon-1} \\ &\leq n^{\log_b a}n^{-\varepsilon}\frac{n^\varepsilon b^\varepsilon}{b^\varepsilon-1} \qquad \text{reste positif, car } b>1 \\ &= n^{\log_b a}\frac{b^\varepsilon}{b^\varepsilon-1} \end{aligned}$$
On en déduit que $C(n) = O(n^{\log_b a})$. Or, d'après la relation de la question 1., on avait également que $C(n) = \Omega(n^{\log_b a})$. On en déduit le résultat.

3.
$$\begin{aligned} \sum_{i=0}^{\log_b n} a^i\left(\frac{n}{b^i}\right)^c &= n^c\sum_{i=0}^{\log_b n}\left(\frac{a}{b^c}\right)^i \\ &\leq n^c\sum_{i=0}^{+\infty}\left(\frac{a}{b^c}\right)^i \qquad \text{car } \frac{a}{b^c}<1 \text{ par hypothèse} \\ &= n^c\frac{1}{1-\frac{a}{b^c}} \\ &= n^c\frac{b^c}{b^c-a} \end{aligned}$$
On en déduit que $C(n) = O(n^c)$. Par l'énoncé du théorème, on avait également que $C(n) = \Omega(n^c)$, d'où le résultat.

4.
$$\begin{aligned} \sum_{i=0}^{\log_b n} a^i\left(\frac{n}{b^i}\right)^c &= n^c\sum_{i=0}^{\log_b n}\underbrace{\left(\frac{a}{b^{\log_b a}}\right)^i}_{1} \\ &= n^{\log_b a}(\log_b n+1) \end{aligned}$$
On a directement le résultat attendu (car le $\Theta(n^{\log_b a})$ est négligeable à côté de la somme).

Exercice 5

1. On crée un vecteur de la bonne taille, et on modifie les cases selon chacun des deux polynômes.

```
let somme p q =
   let m = Array.length p and n = Array.length q in
   let r = Array.make (max m n) 0 in
   for i = 0 to m - 1 do
      r.(i) <- p.(i)
   done;
   for i = 0 to n - 1 do
      r.(i) <- r.(i) + q.(i)
   done;
   r;;
```

```
let sous p q =
   let m = Array.length p and n = Array.length q in
   let r = Array.make (max m n) 0 in
   for i = 0 to m - 1 do
      r.(i) <- p.(i)
   done;
   for i = 0 to n - 1 do
      r.(i) <- r.(i) - q.(i)
   done;
   r;;
```

2. On crée un vecteur de la bonne taille. On modifie la case d'indice $i+j$ pour chaque indice i parcourant les termes du premier polynôme et j du deuxième polynôme :

```
let prod p q =
   let m = Array.length p and n = Array.length q in
   let r = Array.make (m + n - 1) 0 in
   for i = 0 to m - 1 do
      for j = 0 to n - 1 do
         r.(i + j) <- r.(i + j) + p.(i) * q.(j)
      done
   done;
   r;;
```

3. De par les deux boucles `for` imbriquées, la complexité est $O(\deg(P) \times \deg(Q))$.
4. On se contente de décaler les coefficients de k cases :

```
let prod_xk p k =
   let m = Array.length p in
   let r = Array.make (m + k) 0 in
   for i = 0 to m - 1 do
      r.(i + k) <- p.(i)
   done;
   r;;
```

5. On a $PQ = P_0Q_0 + (P_0Q_1 + P_1Q_0)X^k + P_1Q_1X^{2k}$ mais on remarque que $P_0Q_1 + P_1Q_0 = (P_0+P_1)(Q_0+Q_1) - P_0Q_0 - P_1Q_1$. Ainsi, on peut bien calculer PQ en 3 multiplications : P_0Q_0, P_1Q_1 et $(P_0+P_1)(Q_0+Q_1)$.
6. On propose l'algorithme suivant :

Variables : P et Q de degré n
début Kara
  si $n = 1$ **alors**
    Renvoyer $P \times Q$
  sinon
    $k \longleftarrow \dfrac{n}{2}$
    $P_0, P_1 \longleftarrow P[0:k], P[k:n]$
    $Q_0, Q_1 \longleftarrow Q[0:k], Q[k:n]$
    $R_0 \longleftarrow \text{Kara}(P_0, Q_0)$
    $R_1 \longleftarrow \text{Kara}(P_1, Q_1)$
    $R_2 \longleftarrow \text{Kara}(P_0 + P_1, Q_0 + Q_1)$
    $R_3 \longleftarrow R_2 - R_0 - R_1$
    $R \longleftarrow R_0 + \text{prod\_xk}(R_3, k) + \text{prod\_xk}(R_1, 2k)$
    Renvoyer R
  fin si
fin Kara

7. On remarque que les opérations effectuées, à part les appels à `kara` sont en temps linéaires en n (découpage du vecteur en 2, multiplication par X^k, addition). La formule de récurrence de la complexité est donc : $C(n) = 3C\left(\dfrac{n}{2}\right) + \Theta(n)$. En utilisant le théorème maître, on obtient $C(n) = \Theta(n^{\log_2(3)}) \simeq \Theta(n^{1,584})$.

8. Dans un premier temps, on augmente artificiellement la taille des polynômes pour obtenir des puissances de 2 (en rajoutant des 0). Ensuite, on applique l'algorithme décrit ci-dessus.

```
let agrandir p q =
   let m = Array.length p and n = Array.length q in
   let d = ref 1 in
   while !d < max m n do
      d := !d * 2
   done;
   let p' = Array.make !d 0 and q' = Array.make !d 0 in
   for i = 0 to m - 1 do
      p'.(i) <- p.(i)
   done;
   for i = 0 to n - 1 do
      q'.(i) <- q.(i)
   done;
   p',q';;

let separe p n k =
   let p0 = Array.make k 0 and p1 = Array.make k 0 in
   for i = 0 to k - 1 do
      p0.(i) <- p.(i);
      p1.(i) <- p.(i + k)
   done;
   p0, p1;;
```

```
let kara p q =
   let rec aux_kara p q = match Array.length p with
      | 1 -> [|p.(0) * q.(0)|]
      | n -> let k = n / 2 in
             let p0, p1 = separe p n k and
                 q0, q1 = separe q n k in
             let r0 = aux_kara p0 q0 and
                 r1 = aux_kara p1 q1 in
             let r2 = aux_kara (somme p0 p1) (somme q0 q1) in
             let r3 = sous r2 (somme r0 r1) in
             somme r0 (somme (prod_xk r3 k) (prod_xk r1 (2 * k)))
   in
   let p', q' = agrandir p q in
   aux_kara p' q';;
```

III Algorithmes de tris

III.1 Tris élémentaires

III.1.1 Tri par insertion

Exercice 6

1. On compare avec le premier élément pour savoir si on doit ou non insérer dans la queue :

```
let rec insertion x = function
   | [] -> [x]
   | y :: q when x < y -> x :: y ::q
   | y :: q -> y :: (insertion x q);;
```

2. On trie récursivement la queue et on y insère le premier élément :

```
let rec tri_insertion = function
   | [] -> []
   | x :: q -> insertion x (tri_insertion q);;
```

Exercice 7

Le meilleur des cas correspond à une liste déjà triée (les insertions se feront alors en temps constant). Le pire des cas correspond à une liste triée à l'envers (les insertions se feront en temps $O(k)$ où k est la taille de la liste partielle).

III.1.2 Tri par sélection

Exercice 8

1. Une recherche classique de minimum.

```
let minimum t i =
   let m = ref i in
   for j = i + 1 to Array.length t - 1 do
      if t.(j) < t.(!m) then m := j
   done;
   !m;;
```

2. On cherche le minimum qu'on place en première position, et on continue sur le sous-tableau :

```
let tri_selection t =
   for i = 0 to Array.length t - 2 do
      let m = minimum t i in
      swap t i m
   done;;
```

III.2 Tri efficaces

III.2.1 Tri fusion

Exercice 9

1. On alterne les éléments. On fait un appel récursif si la liste a plus que 2 éléments :

```
let rec separation = function
   | [] -> [], []
   | [x] -> [x], []
   | x :: y :: q -> let l1, l2 = separation q in
                    x :: l1, y :: l2;;
```

2. On compare les deux premiers éléments pour savoir lequel doit être au début de la fusion.

```
let rec fusion l1 l2 = match l1, l2 with
   | [], _ -> l2
   | _, [] -> l1
   | x :: q, y :: r when x < y -> x :: (fusion q l2)
   | x :: q, y :: r -> y :: (fusion l1 r);;
```

3. On sépare, on trie récursivement les deux moitiés, puis on fusionne.

```
let rec tri_fusion = function
   | [] -> []
   | [x] -> [x]
   | l -> let l1, l2 = separation l in
          fusion (tri_fusion l1) (tri_fusion l2);;
```

4. Dans un premier temps, remarquons que `separation` et `fusion` ont une complexité linéaire (on fait un seul appel récursif, avec 1 ou 2 éléments en moins). On en déduit que la formule de récurrence de la complexité du tri fusion est :

$$C(n) = 2C\left(\frac{n}{2}\right) + O(n)$$

Le théorème maître nous permet de trouver que la complexité est $O(n \log n)$.
Cette complexité ne change pas en dehors des puissances de 2, car on peut minorer et majorer par les puissances de 2 adjacentes.
Cette complexité est la même dans tous les cas, car les appels récursifs se font toujours sur des objets de mêmes tailles.

5. La formule de complexité spatiale est $S(n) = S\left(\frac{n}{2}\right) + O(n)$. En effet, lors d'un appel récursif, on a besoin d'un espace linéaire (pour garder en mémoire ℓ_1, ℓ_2 et leur version triées), mais comme on fait les appels récursifs l'un après l'autre, l'espace utilisé pour l'un peut ensuite être utilisé pour l'autre. On en déduit un espace $O(n)$ pour le tri-fusion.

Exercice 10

Pour la version linéaire, il suffit de compter les éléments, du plus petit au plus grand, jusqu'à la moitié de la taille de l'ensemble. On garde en mémoire deux indices i et j pour savoir à quel endroit de chaque tableau. On garde également en mémoire les deux dernières valeurs (pour pouvoir faire une demi-somme si nécessaire).

```
let mediane t1 t2 =
   let m = Array.length t1 and n = Array.length t2 in
   let i = ref 0 and j = ref 0 in
   let a = ref 0 and b = ref 0 in
   while !i + !j <= (m + n) / 2 do
      a := !b;
      if t1.(!i) < t2.(!j) then
         (b := t1.(!i); incr i)
      else
         (b := t2.(!j); incr j);
   done;
   if (m + n) mod 2 = 1 then !b else (!a + !b) / 2;;
```

La complexité est bien linéaire, car la somme $i + j$ augmente de 1 à chaque passage.

Pour obtenir une complexité logarithmique, on écrit une fonction qui trouve le k-ème élément le plus petit dans l'union des deux tableaux. On procède comme suit :

- Si l'un est vide, on renvoie l'élément k du deuxième.
- Sinon, on détermine les indices et les valeurs des médianes des deux tableaux. On distingue alors selon la valeur de k par rapport à la somme des deux indices. Par exemple, si k est plus grand que la somme des deux indices et si la médiane du premier tableau est plus grande que celle du deuxième, alors le k-ème élément ne se trouve pas dans la première moitié du deuxième tableau. On raisonne de même dans les 3 autres cas. On utilise la fonction `sub` pour découper les tableaux.

Dès lors, selon la parité de la somme des tailles, on peut déterminer la médiane :

```
let rec keme t1 t2 k = match Array.length t1, Array.length t2 with
   | 0, _ -> t2.(k)
   | _, 0 -> t1.(k)
   | m, n -> let i1, i2 = m / 2, n / 2 in
             let m1, m2 = t1.(i1), t2.(i2) in
             begin match i1 + i2 < k, m1 > m2 with
                | true, true -> let s2 = Array.sub t2 (i2 + 1) (n - i2 - 1) in
                                keme t1 s2 (k - i2 - 1)
                | true, false -> let s1 = Array.sub t1 (i1 + 1) (m - i1 - 1) in
                                 keme s1 t2 (k - i1 - 1)
                | false, true -> let s1 = Array.sub t1 0 i1 in
                                 keme s1 t2 k
                | false, false -> let s2 = Array.sub t2 0 i2 in
                                  keme t1 s2 k
             end;;
```

```
let mediane t1 t2 =
   let n = Array.length t1 + Array.length t2 in
   if n mod 2 = 1 then
      keme t1 t2 (n / 2)
   else
      (keme t1 t2 (n / 2) + keme t1 t2 (n / 2 - 1)) / 2;;
```

À chaque appel récursif, l'un des deux tableaux a sa taille réduite de moitié. On en déduit la complexité logarithmique.

Pour la version améliorée, on sépare les deux tableaux en deux parties (aux indices i et j respectivement), et on les regroupe en mettant les deux parties gauches ensembles et les deux parties droites ensembles. On doit s'assurer que les deux parties sont de même tailles, c'est-à-dire que $i + j = m - i + n - j$ (à 1 près si on a un nombre impair d'éléments). Si on s'assure de plus que le maximum de la partie gauche est plus petit que le minimum de la partie droite, alors on peut calculer efficacement la médiane. Pour cette dernière condition, il suffira de s'assurer que $t1.(i-1) \leq t2.(j)$ et $t2.(j-1) \leq t1.(i)$.

```
let rec mediane t1 t2 =
   let m = Array.length t1 and n = Array.length t2 in
   if m > n then mediane t2 t1 else begin
   let imin, imax, milieu = ref 0, ref m, (m + n + 1) / 2 in
   let i, j = ref (m / 2), ref ((n + 1) / 2) in
   while !imin <= !imax &&
         ((!i < m && t2.(!j - 1) > t1.(!i)) ||
          (!i > 0 && t1.(!i - 1) > t2.(!j))) do
      if !i < m && t2.(!j - 1) > t1.(!i) then imin := !i + 1
      else imax := !i - 1;
      i := (!imin + !imax) / 2;
      j := milieu - !i;
   done;
   let max_g = (match !i, !j with
      | 0, _ -> t2.(!j - 1)
      | _, 0 -> t1.(!i - 1)
      | _ -> max t1.(!i - 1) t2.(!j - 1)) in
   let min_d = (match !i, !j with
      | _ when !i = m -> t2.(!j)
      | _ when !j = n -> t1.(!i)
      | _ -> min t1.(!i) t2.(!j)) in
   if (m + n) mod 2 = 1 then max_g else (max_g + min_d) / 2
   end;;
```

Pour la complexité, on remarque qu'il s'agit d'une dichotomie sur l'indice i, qui parcourt les indices du tableau le plus petit. On a bien la complexité attendue.

III.2.2 Tri rapide

Exercice 11

1. On modifie les indices. On renvoie l'indice i à la fin de la fonction.

```
let partition t debut fin =
   let p = t.(debut) and i = ref debut in
   for j = debut + 1 to fin - 1 do
      if t.(j) < p then
         (incr i; swap t !i j)
   done;
   swap t debut !i;
   !i;;
```

2. Pour la fonction de tri rapide : on partitionne, puis on trie récursivement les deux moitiés. On utilise une fonction auxiliaire qui prend en argument les indices des sous-tableaux à trier (pour éviter de recréer des tableaux).

```
let tri_rapide t =
   let rec aux_tri t debut fin =
      if fin - debut > 1 then
         let milieu = partition t debut fin in
         aux_tri t debut milieu;
         aux_tri t (milieu + 1) fin in
   aux_tri t 0 (Array.length t);;
```

Exercice 12

On se contente de modifier le premier élément du tableau avant de lancer le tri :

```
let tri_rapide_alea t =
   let rec aux_tri t debut fin =
      if fin - debut > 1 then
         let k = Random.int (fin - debut) in
         swap t debut (debut + k);
         let milieu = partition t debut fin in
         aux_tri t debut milieu;
         aux_tri t (milieu + 1) fin in
   aux_tri t 0 (Array.length t);;
```