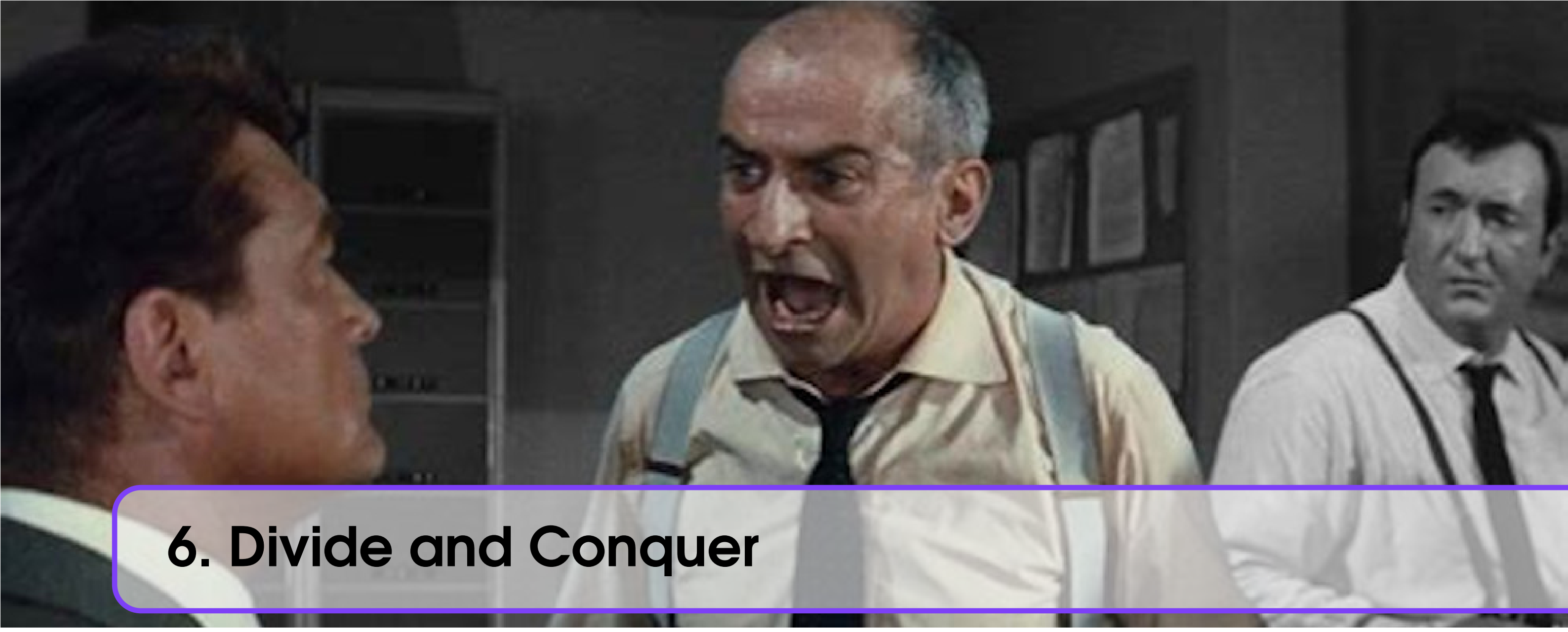
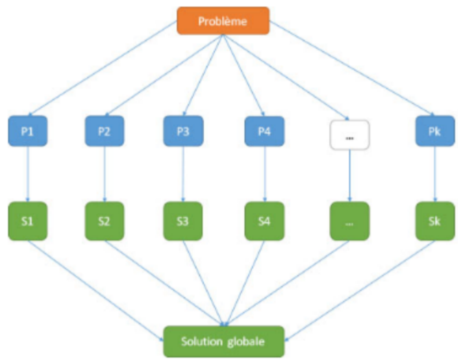


6. Divide and Conquer

Introduction

Utiliser un algorithme **diviser pour régner** (*divide and conquer* en Anglais, ce qui n'est pas tout à fait la même chose), c'est décomposer un problème en sous-problèmes.
Ces sous-problèmes sont au plus de même difficulté que le problème initial. Souvent ils sont de même complexité que le problème initial.



Vocabulaire :

Diviser : étape consistant à séparer le problème initial en sous-problèmes.
Régner : étape consistant à résoudre les sous-problèmes.
Combiner : étape de résolution du problème initial.

Récursif ou itératif ?
La récursivité est assez pratique pour décomposer le problème et est généralement une bonne solution.
Par contre une pile d'appels récursifs trop importante peut amener un `stack overflow` (dépassement de pile).

Parallélisation des calculs
Une fois les sous-problèmes définis, on peut les traiter simultanément en parallélisant le traitement, c'est-à-dire en les traitant par des processeurs différents, voire des machines différentes.

6.1 L'algorithme de recherche par dichotomie

Le premier algorithme du type **Diviser pour régner** est apparu dans l'Antiquité (Babylone vers -200), il s'agit de la recherche **dichotomique**. Historiquement il avait été proposé qu'on nomme plutôt **decrease and conquer** ces algorithmes à un seul sous-problème.

6.1.1 Spécification

Rôle :	Trouver une valeur dans un tableau.
Entrées :	Un tableau T de longueur n. Une valeur v.
Préconditions :	Les valeurs de T sont comparables. Le tableau est **trié** dans l'ordre croissant. La valeur v est comparable aux valeurs de T.
Sortie :	Un nombre entier *indice* entre -1 et $n-1$.
Postconditions :	Si la valeur v n'est pas dans le tableau T, alors *indice* vaut -1. Sinon, *indice* est un nombre entier tel que $0 \leq indice \leq n-1$ et $T[indice] = v$.

6.1.2 L'algorithme

L'algorithme de recherche :

Entrées : un tableau T de longueur n rangé dans l'ordre croissant une valeur v

Sorties : un entier i entre -1 et $n-1$

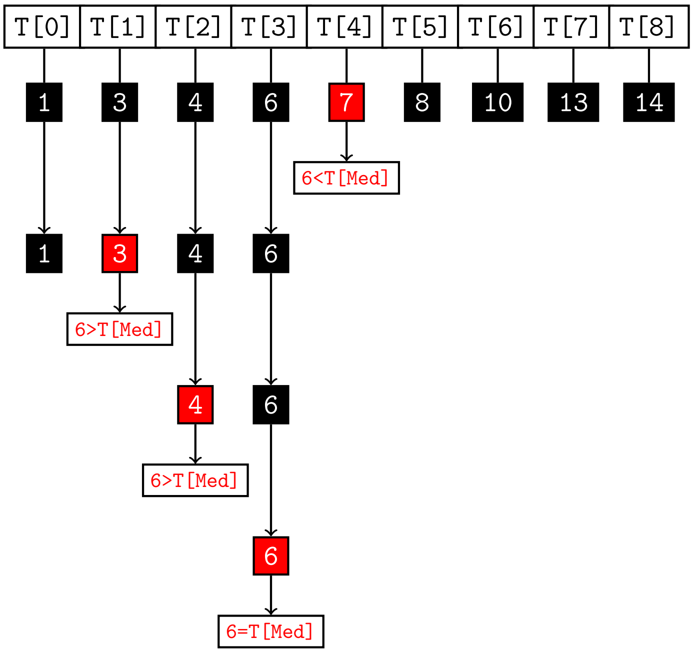
$min \leftarrow 0$;
$med \leftarrow 0$;
$max \leftarrow n-1$;
tant que $min < max$ **faire**
  $med \leftarrow (min+max)//2$;
  si $T[med] < v$ **alors**
    $min \leftarrow med+1$;
  sinon
    si $T[med] > v$ **alors**
      $max \leftarrow med-1$;
    sinon
      $min \leftarrow med$;
      $max \leftarrow med$;
    fin si
  fin si
fin tq
si $T[min] == v$ **alors**
  $indice \leftarrow min$;
sinon
  $indice \leftarrow -1$;
fin si

La version en langage Python :

```python
mini = 0
med = 0
maxi = n-1
while mini < maxi:
        med = (mini + maxi)//2
        if T[med]< v :
                # v est dans la partie supérieure du tableau
                mini = med +1
        elif T[med] > v :
                # v est dans la partie inférieure du tableau
                maxi = med -1
        else :
                maxi = med
                mini = med
if T[mini]==v :
        indice = mini
else:
        indice = -1
```

6.1.3 Un exemple

Dans le tableau `T = [1, 3, 4, 6, 7, 8, 10, 13, 14]`, nous recherchons le nombre 6.



`Med = (0+8)//2 = 4` on regarde `T[Med]=T[4]=7`

on ne garde que la partie gauche du tableau

`Med = (0+3)//2 = 1` on regarde `T[Med]=T[1]=3`

on ne garde que la partie droite du tableau

`Med = (2+3)//2 = 2` on regarde `T[Med]=T[2]=4`

on ne garde que la partie droite du tableau

`Med = (3+3)//2 = 3` on regarde `T[Med]=T[3]=6`

on a trouvé le nombre 6 dans le tableau

6.1.4 Efficacité

Nous avions vu en première un algorithme simple de recherche :

Entrées : un tableau T de longueur n
une valeur v
Sorties : un entier i entre -1 et $n-1$

$i \leftarrow -1$;
$k \leftarrow 0$;
tant que ($k <$taille du tableau T) ET ($i = -1$) **faire**
  si $T[k] = v$ **alors**
    $i \leftarrow k$;
  sinon
    $k \leftarrow k+1$;
  fin si
fin tq

Pour la complexité de cet algorithme, il faut prévoir le pire cas, autrement dit le nombre recherchée v n'est pas dans le tableau. Cet algorithme va alors tester toutes les n cases du tableau.

L'algorithme de recherche ***simple*** est un algorithme de **complexité linéaire**.

On le note $O(n)$

En ce qui concerne l'algorithme de recherche dichotomique, comme à chaque étape on ne garde que la moitié du tableau restant, on procède par divisions successives par 2. Le lien entre la longueur n du tableau et son écriture binaire apparaît (même si ce n'est pas vraiment évident).

L'algorithme de recherche ***dichotomique*** est un algorithme de **complexité logarithmique**.

On le note $O(\log(n))$

On peut ainsi comparer l'efficacité des deux algorithmes :

n	Algorithme de recherche *classique*	Algorithme de recherche *dichotomique*
$n = 10$	Il y a de l'ordre d'une dizaine d'opérations	Comme $10_{10} = 1010_2$, on a : $\log_2(10) = 4$. Il y a de l'ordre de 4 opérations
$n = 100$	Il y a de l'ordre d'une centaine d'opérations	Comme $100_{10} = 1100100_2$, on a : $\log_2(100) = 7$. Il y a de l'ordre de 7 opérations
$n = 1\ 000$	Il y a de l'ordre d'un millier d'opérations	Comme $1000_{10} = 1111101000_2$, on a : $\log_2(1000) = 10$. Il y a de l'ordre de 10 opérations
$n = 1\ 000\ 000$	Il y a de l'ordre d'un million d'opérations	Comme $1000000_{10} = 11110100001001000000_2$, on a : $\log_2(1000000) = 1000$. Il y a de l'ordre de 1000 opérations
...	...	...

6.1.5 Récursivité

exercice 6.1 Réécrire l'algorithme de recherche dichotomique par récursivité. ■

6.2 Algorithmes de tri

6.2.1 Rappel : Tri par sélection

A chaque étape, on a à gauche, les éléments déjà triés et à droite les éléments non encore triés.
Le tri par sélection consiste à sélectionner dans la liste des valeurs non encore triées (à droite), la valeur la plus petite, puis à la positionner comme la nouvelle valeur la plus grande dans la liste des valeurs déjà triées (à gauche). Par exemple, on veut trier le tableau `T = [70, 50, 30, 10, 20, 40, 60]`. On écrira en rouge les cases triées, en bleu le minimum des cases non encore triées.

70	50	30	10	20	40	60	10 est le minimum des cases non triées
10	70	50	30	20	40	60	on décale 10 afin que ce soit la plus grande valeur triée
10	70	50	30	20	40	60	20 est le minimum des cases non triées
10	20	70	50	30	40	60	on décale 20 afin que ce soit la plus grande valeur triée
10	20	70	50	30	40	60	30 est le minimum des cases non triées
10	20	30	70	50	40	60	on décale 30 afin que ce soit la plus grande valeur triée
10	20	30	70	50	40	60	40 est le minimum des cases non triées
10	20	30	40	70	50	60	on décale 40 afin que ce soit la plus grande valeur triée
10	20	30	40	70	50	60	50 est le minimum des cases non triées
10	20	30	40	50	70	60	on décale 50 afin que ce soit la plus grande valeur triée
10	20	30	40	50	70	60	60 est le minimum des cases non triées
10	20	30	40	50	60	70	on décale 60 afin que ce soit la plus grande valeur triée
10	20	30	40	50	60	70	70 est le minimum des cases non triées
10	20	30	40	50	60	70	on décale 70 afin que ce soit la plus grande valeur triée

L'algorithme doit faire 2 choses. Chercher le minimum parmi les valeurs non triées puis l'échanger avec la première valeur non encore triée.

L'algorithme de tri par sélection :

pour i **allant de** 0 **à** $n-1$ **faire**
  $indice\_min \leftarrow i$;
  pour j **allant de** $i+1$ **à** $n-1$ **faire**
    si $T[indice\_min] > T[j]$ **alors**
      $indice\_min \leftarrow j$;
    fin si
  fin pour
  $temp \leftarrow T[i]$;
  $T[i] \leftarrow T[indice\_min]$;
  $T[indice\_min] \leftarrow temp$;
fin pour

La version en langage Python :

```python
for i in range(0,n) :
    indice_min = i
    # on cherche le minimum à partir de T[i]
    for j in range(i+1,n) :
        if T[indice_min] > T[j] :
            indice_min = j
    # on échange
    T[indice_min], T[i] = T[i], T[indice_min]
```

L'algorithme de tri ***par sélection*** est un algorithme de **complexité quadratique**.

On le note $O(n^2)$

6.2.2 Rappel : Tri par insertion

A chaque étape, on a à gauche, les éléments déjà triés et à droite les éléments non encore triés.
Le tri par insertion consiste à prendre le premier élément non encore trié puis à le placer à sa place dans les éléments triés. Par exemple, on veut trier le tableau `T = [70, 50, 30, 10, 20, 40, 60]`. On écrira en rouge les cases triées, en bleu le premier élément non encore triées.

70	50	30	10	20	40	60	70 est le premier élément non triées
70	50	30	10	20	40	60	on place 70 dans à sa place dans les éléments triés
70	50	30	10	20	40	60	50 est le premier élément non triées
50	70	30	10	20	40	60	on place 50 dans à sa place dans les éléments triés
50	70	30	10	20	40	60	30 est le premier élément non triées
30	50	70	10	20	40	60	on place 30 dans à sa place dans les éléments triés
30	50	70	10	20	40	60	10 est le premier élément non triées
10	30	50	70	20	40	60	on place 10 dans à sa place dans les éléments triés
10	30	50	70	20	40	60	20 est le premier élément non triées
10	20	30	50	70	40	60	on place 20 dans à sa place dans les éléments triés
10	20	30	50	70	40	60	40 est le premier élément non triées
10	20	30	40	50	70	60	on place 40 dans à sa place dans les éléments triés
10	20	30	40	50	70	60	60 est le premier élément non triées
10	20	30	40	50	60	70	on place 60 dans à sa place dans les éléments triés

L'algorithme doit prendre à chaque fois le premier élément non trié et ensuite le décaler vers la gauche, case par case, jusqu'à ce qu'il soit trié.

L'algorithme de tri par insertion :

pour i **allant de** 1 **à** $n-1$ **faire**
  $j \leftarrow i$;
  tant que $j > 0$ **et** $T[j-1] > T[j]$ **faire**
    $temp \leftarrow T[j-1]$;
    $T[j-1] \leftarrow T[j]$;
    $T[j] \leftarrow temp$;
    $j \leftarrow j-1$;
  fin tq
fin pour

La version en langage Python :

```python
for i in range(1,n) :
    #on prend un nouvel indice
    j = i
    while (j>0) and (T[j-1] > T[j]) :
        # tant qu'à gauche, il y a un élément plus grand
        # on les échange
        T[j], T[j-1] = T[j-1], T[j]
        j = j - 1
```

L'algorithme de tri ***par insertion*** est un algorithme de **complexité quadratique**.

On le note $O(n^2)$

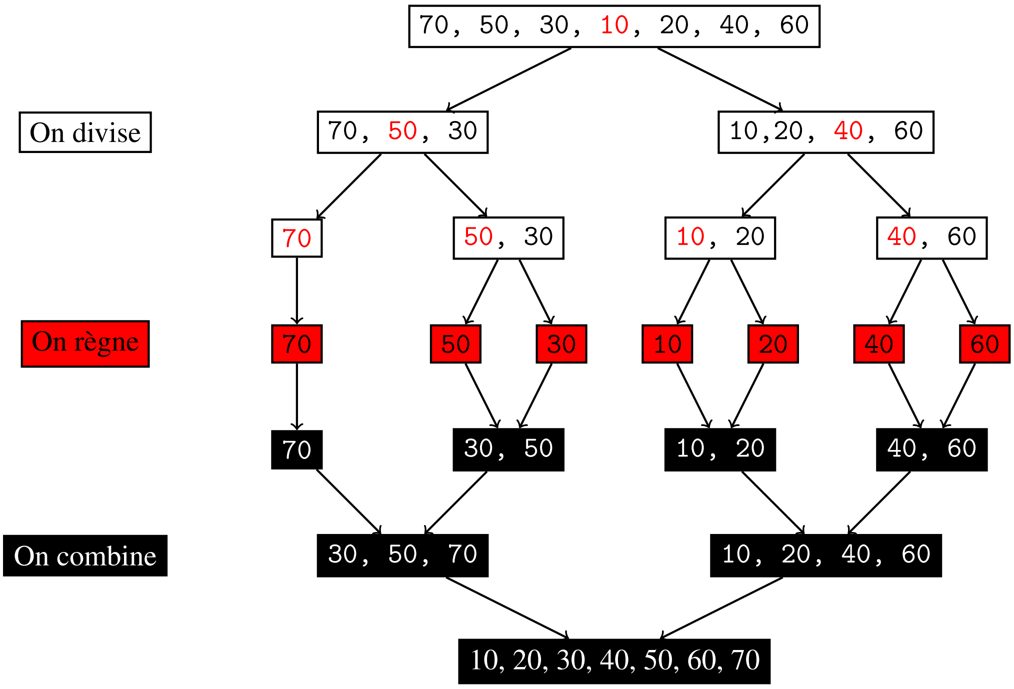
6.2.3 Le tri fusion

Le principe est le suivant :

Diviser : On divise le tableau en deux parties (de dimensions assez proches) et on répète le procédé de manière récursive.

Régner : Une fois le problème suffisamment subdivisé, il ne reste que des sous-tableaux à un seul élément.

Combiner : On effectue les tâches de comparaison en remontant au fur et à mesure vers le tableau composant tous les éléments.



La phase Combiner est, dans le cas du **tri fusion**, appelée **fusion**. Dans le schéma précédent, la première phase de comparaison (sur deux éléments) ne posent pas de problème. Ce sont bien les phases de comparaison suivantes qui sont plus difficiles à comprendre. Par exemple, comment fusionner les listes triées `[70]` et `[30, 50]` ?

Étape 1 : on considère les deux plus petits éléments de chacune des listes (70 et 30) et on les compare, on obtient alors le plus petit élément de la liste fusionnée : 30.

Étape 1bis : on a donc : `[30]` et il reste : `[70]` et `[50]`.

Étape 2 : on répète l'étape 1 : on compare donc 70 et 50. on obtient donc le deuxième élément le plus petit : 50.

Étape 2bis : on a donc : `[30,50]` et il reste `[70]` et `[]`.

Étape 3 : Comme la seconde liste est vide, on ajoute à la liste fusionnée les éléments restants de la première liste : `[30, 50, 70]`.

Complexité :

Comme on divise la taille du tableau par 2 à chaque appel récursif, on fait $\log(n)$ appels.
Lors de la phase de combinaison, on parcourt les listes de gauche et de droite, donc en temps n à chaque appel récursif. D'où une complexité quasi linéaire en $O(n\log(n))$.

L'algorithme de tri ***fusion*** est un algorithme de

On le note $O(n\log(n))$

exercice 6.2
1. Programmer une fonction `fusion(tab_gauche, tab_droite)` qui prend en argument 2 tableaux d'éléments triés et renvoie un tableau trié contenant tous les éléments des deux tableaux.
2. Programmer une fonction `tri_fusion(tab)` qui prend en argument un tableau d'entiers et qui renvoie un tableau trié contenant tous les éléments du tableau.

6.3 Quelques exercices

exercice 6.3 — Minimum et maximum. L'objectif est d'écrire une fonction `min_et_max(T)` renvoyant le couple `(minimum, maximum)` du tableau `T` à l'aide d'un algorithme du type *diviser pour régner*.
1. Faire des essais à la main avec des cartes.
2. Proposer un algorithme « classique » puis un algorithme du type diviser pour régner.
3. Coder vos deux algorithmes en Python puis tester-les avec le tableau
   `L = [9, 1, 6, 5, 15, 11, 12, 14, 13, 4, 3, 8, 7, 10, 2]`.
4. Étudier la complexité de ces deux algorithmes et comparer-les.

exercice 6.4 — Exponentiation rapide et nombre de Fermat. Voici une méthode diviser pour régner pour calculer x^n :

$$puissance(x,n) = \begin{cases} x & \text{si } n = 1 \\ puissance(x^2, n/2) & \text{si } n \text{ est pair} \\ x \times puissance(x^2, (n-1)/2) & \text{si } n \text{ est impair} \end{cases}$$

1. Coder cet algorithme en Python puis tester-le.
2. Étudier la complexité de cette algorithme et comparer-le avec un algorithme classique.
3. Les nombres de Fermat ont été proposé par Pierre de Fermat comme étant des nombres premiers. Ils s'obtiennent ainsi :
   $$F_n = 2^{2^n} + 1$$
   Bien que F_1, F_2, F_3, F_4 sont effectivement premiers, F_5 ne l'est pas.
   On ignore actuellement si les nombres de Fermat au-delà de F_{33} sont premiers ou non, mais on conjecture que seuls F_1, F_2, F_3 et F_4 le sont.
   Calculer F_4 et F_5.

exercice 6.5 — `T[i]=i`. On dispose d'un tableau T composé d'entiers ordonnés de manière ascendante.
1. Proposer un algorithme du type *diviser pour régner* qui permet de déterminer l'entier i tel que $T[i] = i$.
2. Tester votre algorithme à la main sur le tableau suivant
   `T = [0, 2, 2, 5, 8, 12, 13, 15, 19, 26, 27]`.
3. Écrire en Python votre algorithme et tester-le.

exercice 6.6 — Plus grande somme. On donne un tableau contenant des nombres positifs et négatifs. On veut trouver la somme maximale d'éléments consécutifs de ce tableau.
Par exemple pour le tableau `T = [-2, -5, 6, -2, -3, 1, 5, -6]`, la somme maximale est 7, correspondant aux nombres en rouge.
1. Proposer un algorithme « classique » puis un algorithme du type diviser pour régner.
2. Coder vos deux algorithmes en Python puis tester-les avec le tableau
   `T = [-2, -5, 6, -2, -3, 1, 5, -6]`.
3. Étudier la complexité de ces deux algorithmes et comparer-les.