Questions de cours :

1. A quoi sert l'étude de la complexité des algorithmes ?
2. Comment gérer la récursivité pour évaluer la complexité des algorithmes?
3. Comment savoir si mon algorithme est optimal ?

Exercice 1 : (évaluation de complexité et D&C)

A. Donner la complexité (en temps) de l'algorithme suivant :

```
{début}
  K ⟵ 0
  I ⟵ 1
  {#1}
  TantQue I ≤ N {#2} Faire
  R ⟵ R+T[I]  {#3}
  Si R>1000 {#3'} Alors
  R ⟵ 2*R   {#3''}
  finsi
  I ⟵ I+I  {#4}
  finTanQue
{fin}
```

B. Utilisant la stratégie Diviser pour Régner (divide&conquer), donner l'algorithme de :

1. Supprimer les doublons avec la stratégie

Données : Une collection d'éléments S = {{s1, s2, …, sn}}, i.e. une liste.
Sortie : S sans éléments identiques.

2. Supprimer les doublons sans la stratégie divide&conquer

Données : Un tableau T.
Sortie : T sans éléments identiques.

Exercice 2 : (Le tris)

1. Le premier algorithme pour trier un tableau, le tri par sélection on recherche le plus petit élément parmi les n éléments et on l'échange avec le premier élément ; puis on recherche le plus petit élément parmi les n−1 derniers éléments de ce nouveau tableau et on l'´echange avec le deuxième élément; plus généralement, à la k-ième étape, on recherche le plus petit élément parmi les n−k +1 derniers éléments du tableau en cours et on l'échange avec le k-ième élément.
   - Donner un algorithme qui réalise le tri par sélection

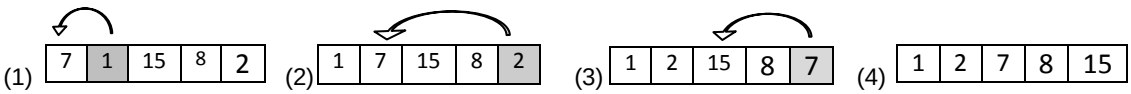


Figure : Comment Tri-Sélection trie le tableau [7, 1, 15, 8, 2]

2. Donner un algorithme employant la stratégie Diviser pour Régner (Divide and Conquer) pour réaliser le tri par fusion d'un tableau T de N entiers indicés de l à r.
   - Donner l'arbre d'exécution de cet algorithme sur la séquence suivante :
     (18 19 14 19 14 12 111 5)
     N.B : Le tableau contient les mêmes éléments mais rangés par ordre croissant.
3. Analyser la complexité en espace et en temps des algorithmes de tris par sélection et tris par Diviser pour Régner ?

Exercice 3 : (La recherche)

Etant donnée un tableau T de n éléments et un élément k.

1. Donner un algorithme de recherche séquentielle de premier indice i ou se trouve l'élément k si k est dans T, et sinon la réponse «k n'est pas dans T».
2. Donner un algorithme de recherche dichotomique utilisant la technique Diviser pour Régner pour rechercher un indice i ou se trouve l'élément k ou bien -1 (par convention si k n'est pas dans T).
   N.B : Les élément de tableau Test déjà ordonné
3. On suppose que la taille N du tableau T est une puissance de 2, N = 2n.
   a. Modifier l'algorithme précédant pour trouver le premier indice i où se trouve l'élément k.
   b. Ecrire un algorithme récursif pour trouver le premier indice i où se trouve l'élément k

4. Analyser la complexité en temps de ces deux algorithmes précédents, que tu peut conclu

Exercice 4 : (Valeur encadrée)

Donner un algorithme de type Diviser pour Régner qui cherche s'il existe un élément T[i] du tableau T[1..N] trié d'entiers, cet valeur se trouve entre deux bornes (l,r) tel que : $l \leq T[i] \leq r$

- Analysez sa complexité

Exemple Soit le tableau T[1..6] = [5,6,67,123,214] et les bornes l =50, r = 100. Avec ces données la valeur encadrée est : T[3] = 67.

Exercice 5 : (somme d'un tableau)

Etant donné un tableau T de n éléments entiers.

- Donner un algorithme qui utilise la technique Diviser pour Régner qui cherche la somme des éléments de ce tableau T
- Analysez sa complexité

Exercice 6 : (NP-complet)

Dans un graphe on appelle un sous-ensemble S de sommets *stable* s'il n'existe pas d'arête du graphe qui relie deux sommets de S.

Étant donné un graphe non-orienté G = (V, E) (de N sommets) représenté par une matrice d'adjacence M[i, j] (de taille N×N) et un entier M on cherche un ensemble stable de C sommets. Il faut trouver un algorithme retour-arrière qui résout ce problème.

On va appeler une solution partielle un tableau d'entiers [i1, . . . , ik] dans l'ordre croissant tel que les sommets [vi1, . . . , vik] forment un stable.

1. Écrivez une fonction booléenne test(B, k,M,N) qui teste est-ce que le tableau B[1..k] est une solution partielle pour un graphe représenté par un tableau (matrice d'adjacence) M[N,N].

2. Comment trouver une solution partielle de taille 0 ? Comment à partir d'une solution partielle de taille k passer à ses extensions de taille k + 1 ? Comment dire est-ce qu'on a déjà trouvé le stable de taille C?

3. Écrivez un algorithme retour-arrière de recherche d'un stable de taille C.

4. Estimez la complexité de votre algorithme.

Correction

La réponse de questions de cours :

1. L'objectif d'étudier la complexité des algorithme est d'évaluer la performance des algorithme en terme de temps et d'espace mémoire.
2. On peut gérer la récursivité par l'utilisation de la pile (structure de donnée dynamique).
3. L'algorithme est optimale ssi leur valeur de complexité égale a la valeur de la complexité de la borne inferieur.

Exercice 1 : (évaluation de complexité)

A. **Temps(n)=ΔT1+(ΔT2+ΔT3+ΔT4+ΔT5+ΔT6)*n+ΔT2**

$=c^{te}+tit*n= c^{te}+ c^{te}*n$

$\lim_{n\to\infty}$ Temps(n)=lim $c^{te}+n*c^{te}/ n*c^{te}$ =1 la complexité est asymptotiquement linéaire dans le pire des cas

B.

1. Supprimer les doublons avec la stratégie divide&conquer

Un premier réflexe serait de tester simplement la différence. En réalité, il faudrait les trier pour arriver à ce que les doublons se 'rejoignent'.

L'initialisation de l'algorithme est classique :

$SD(\emptyset) = \emptyset$

$SD(C.\emptyset) = C$

$|L| \geq 2 \rightarrow SD(L) = \Delta(SD(L_{impaire}), SD(L_{paire}))$

Regardons maintenant la fonction propre à cet algorithme, à savoir Δ :

$\Delta(\emptyset, L2) = L2$

$\Delta(L1, \emptyset) = L1$

$c1 < d1 \rightarrow c1.\Delta(c2, d1.d2)$ On trie la liste. c1 est le plus petit, on le met en tête, on trie le reste.

$c1 = d1 \rightarrow c1.\Delta(c2, d2)$ On a trouvé notre doublon : on enlève un des deux, et on continue.

$c1 > d1 \rightarrow d1.\Delta(c1.c2, d2)$ Comme dans la première clause, d1 est le plus petit, on trie le reste.

Le principe à retenir est que lorsqu'on a une liste, le résultat aura de grandes chances d'être récursif avec comme cas d'arrêt une des deux listes vides. On pourra alors rendre l'autre liste, qui aura été triée.

2. Supprimer les doublons sans la stratégie divide&conquer

La première question à se poser : a-t-on besoin de trier le tableau pour ça ?
Si non, on affirme pouvoir tout faire en une passe. Ici, ce n'est à priori pas le cas.
On a trié le tableau. On part du début avec un pointeur j. On avance jusqu'à trouver un élément différent des précédents. On l'échange avec i+1 et on sauvegarde la position où l'élément différent a été trouvé, afin de reprendre les recherches à ce point.

a a a b b c d d. On a trouvé un élément différent. On le permute avec i+1, on reprend la recherche.
a b a a b c d d
a b c a b b d d
a b c d b b a d, on est arrivé au bout, fin.

Cet algorithme a deux avantages :

- il s'écrit d'une façon très courte et intuitive
- comme on s'est payé le tri, il est mieux de pouvoir rendre par la même occasion un résultat trié.

Algo(T, i, j, k) = Algo(merge_sort(T), 1, 2, n)
j ≠ n+1 ^ T[i] = T[j] → Algo(T, i, j, k) = Algo(T, i, j+1, n) rien trouvé de différent : on continue
^ T[i] ≠ T[j] → Algo(T, i, j, k) = Algo(T[i+1↔j], i+1, j+1, n) différent : permute, augmente i
j = n + 1 → i (point à partir duquel le tableau est fini)

Exercice 2 : (Le tris)

1. TRI-SELECTION(tableau T, entier N)
Données : Un tableau T de N éléments comparables
Résultat : Le tableau T contient les mèmes éléments mais rangés par ordre croissant
Indice entier i, j, min
for i = 1 to N - 1 do
min = i // On initialise l'indice du minimum des N i + 1 derniers éléments
for j = i + 1 to N do
if (T[j] < T[min]) then
min = j // On actualise l'indice du minimum en cours
échange (T; i; min) // min est l'indice du plus petit élément comp ris entre les indices i et N (inclus) et on permute les éléments d'incices i et min
/* Le tableau T contient les i plus petits éléments du tableau initial rangés aux places 1 … i */

2.

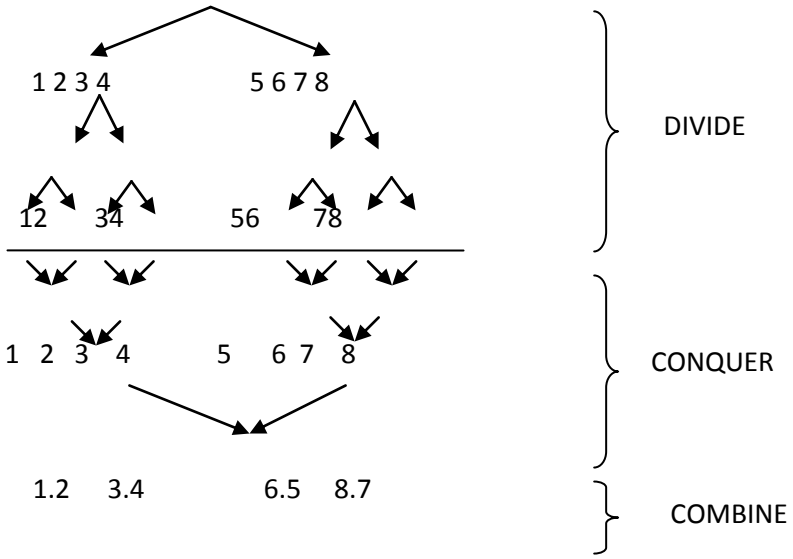
TRI-FUSION (tableau T, entier N; l; r)
Données : Un tableau T de N entiers indicés de l _a r
Résultat : Le tableau T contient les memes éléments mais rangés par ordre croissant
Indice entier m
if l < r **then**
m = [(l + r)/2] // On calcule l'indice du milieu du tableau
TRI-FUSION(T; l;m)
TRI-FUSION(T;m+1,r)
fusion(T; l; r;m)

fusion (tableau T, entier l; r;m)
Données : Un tableau T d'entiers indicés de l à r, et tel que l ≤ m < r et que les sous-tableaux T[l … m] et T[m + 1 … r] soient ordonnés
Résultat : Le tableau T contenant les mêmes éléments mais rangés par ordre croissant
Indice entier i, j, k, n1, n2
Var tableau d'entiers L;R
n1 = m - l + 1
n2 = r - m
/* On utilise des tableaux temporaires L;R et une allocation par blocs L = T[l … m] et R = T[m + 1 … r] */
for i = 1 to n1 **do**
L[i] = T[l + i - 1]
for j = 1 to n2 **do**
R[j] = T[m + j]
i = 1

```
j = 1
L[n1 + 1] = ∞ // On marque la fin du tableau gauche
R[n2 + 1] = ∞ // On marque la fin du tableau droit
for k = l to r do
        if (L[i] _ R[j]) then
              T[k] = L[i]
              i = i + 1
        else
             T[k] = R[j]
             j = j + 1
```

- L'arbre d'exécution de la séquence :

18_1 19_2 14_3 19_4 14_5 12_6 111_7 5_8



3.

TRIS-SELECTION

complexité en temps : le temps d'exécution t(n) du programme dépendra en général de la taille n des données d'entrée ; on distinguera :

- la complexité moyenne, c'est-à-dire la valeur moyenne des t(n) : elle est en général difficile à évaluer, car il faut commencer par décider quelle donnée est « moyenne » ce qui demande un recours aux probabilités ;
- la complexité pour le pire des cas, c'est-à-dire pour la donnée d'entrée donnant le calcul le plus long, soit t(n) maximal ;
- la complexité pour le meilleur des cas, c'est-à-dire pour la donnée d'entrée correspondant au calcul le plus court, soit t(n) minimal.

L'algorithme TRI-SELECTION trie les éléments du tableau dans le tableau lui-mème et n'a besoin d'aucun espace supplémentaire. Il est donc très économe en espace car il fonctionne en espace constant : on trie le tableau sur place, sans avoir besoin de stocker une partie du tableau dans un tableau auxiliaire. Par contre il n'est pas économe en temps

DIVISER POUR REGNER

La complexité en temps est **n log n** , l'espace est **linéaire** dans tous les cas.

Complexité	Espace	Temps		
		pire	moyenne	meilleure
Tri sélection	Constant	N^2	N^2	N^2
Tri fusion	Linéaire	n log n	n log n	n log n

Exercice 3 : (La recherche)

1. RECHERCHE (tableau T, élément k)

Données : Un tableau T de n éléments et un élément k

Résultat : Le premier indice i où se trouve l'élément k si k est dans T, et sinon la réponse « k n'est pas dans T »

```
Indice entier i
 i = 1
 while ((i ≤n) ^ (T[i] ≠ k))  do
       i = i + 1
```

```
if (i _ n) then
        afficher T[i] = k
    else
       afficher k n'est pas dans T
```

2 .

RECHERCHEDICHO (tableau T, entier k)
Données : Un tableau T[1 : : :N] d'entiers déjà ordonné et un entier k
Résultat : Un indice i où se trouve l'élément k, ou bien -1(par convention si k n'est pas dans T)

```
Indice entier i; l; r
    l = 1
    r = N
    i = [(l + r)/2]
    while ((k ≠ T[i]) ^ (l ≤ r)) do
       if (k < T[i]) then
            r = i - 1
       else
            l = i + 1
        i = [(l + r)/2]
    if (k == T[i]) then
    retourner i
    else
    retourner -1
```

Recherche séquentielle :

La complexité en temps de RECHERCHE est linéaire (de l'ordre de n), puisqu'il faudra au pire parcourir tout le tableau.

D&C :

Soit t(n) le nombre d'opérations effectuées dans cet algorithme sur un tableau de taille n : t(n) satisfait l'équation de récurrence t(n) = t(n=2)+1, comme t(1) = 1, on en déduit t(n) = O(log n). On remarquera que la complexité en temps est réduite de linéaire (O(n)) à logarithmique (O(log n)) entre la recherche séquentielle et la recherche dichotomique.

Exercice 4 : (Valeur encadrée)

Coupons le tableau en deux moitiés : A[s..m] et A[m + 1..f]. Les trois cas ci-dessous correspondent à trois positions de l'élément de milieu A[m] par rapport à l'intervalle [a, b].

À gauche : A[m] < a. Dans ce cas-là tous les éléments de la moitié gauche du tableau sont aussi < a, et ne peuvent pas appartenir à l'intervalle demandé [a, b]. On va chercher la valeur encadrée dans la moitié droite seulement.

Dedans :a ≤ A[m] ≤ b. On a déjà trouvé une valeur encadrée A[m]. On retourne son indice.

À droite :b < A[m]. Ce cas est symétrique au premier, la recherche peut être limitée à la moitié gauche du tableau.

Cette analyse mène à la fonction récursive suivante chercher(s,f) qui renvoie l'indice de la valeur encadrée dans le tableau A[s..f] (ou ⊥ s'il n'y en a pas. On suppose que le tableau A, et les bornes a,b sont des variables globales.

```
fonction chercher(s,f)
    // cas de base
    si (s=f)
        si (A[s] ∈ [a; b])
                retourner s
        sinon retourner ⊥
     // on divise pour régner
    m=(s+f)/2
    si A[m] < a  ind= chercher(m+1,f)
    sinon si a ≤ A[m] ≤ b   ind=m
          sinon   ind=chercher(s,m)
    retourner ind
```

2. On a réduit un problème de taille n à un seul problème de la taille n/2 et des petits calculs en O(1), donc la complexité satisfait la récurrence :

T(n) = T(n/2) + O(1).

T(n) = O(log n).

Exercice 5 : (somme d'un tableau)

```
fonction Sum(B,i,j)
    si i=j
        retourner B[i]
    m= (i+j) div 2
    gauche=Sum(B,i,m)
    droite=Sum(B,m+1,j)
    retourner (gauche+droite)
```

On a la récurrence suivante sur la complexité de cet algo :
$T(n) = 2 * T(n/2) + O(1)$
l'exposant est $k = \log_2 2 = 1$, que la perturbation $O(1) = O(n0)$
est petite, et donc : $T(n) = O(n)$.

Exercice 6 : (NP-complet)

Pour le test on suppose que B[k] est un tableau croissant de k entiers entre 0 et N. Il reste à tester l'absence d'arêtes. Dans la fonction ci-dessus je suppose que B,M,N sont des variables globales.

```
Boolean function test(k)
        pour i de 1 à k-1
            pour j de i+1 à k
                si M[B[i],B[j]]= 1
                    retourner faux
        retourner vrai
```

–La seule solution partielle de taille 0 est le tableau vide.

–Si on a une solution partielle B[1..k] pour l'étendre il faut choisir un nouveau sommet v supérieur à B[k] (attention, pour k=0 cette contrainte disparaît) ; et l'ajouter à la place B[k+1]. Il faut que le B ainsi obtenu passe le test ci-dessus.

–Une solution partielle B[1..k] est un stable recherché si k = C.

Ceci mène à l'algorithme retour-arrière suivant

```
Fonction TrouverStable(k)
                    si k=C
                        imprimer B ; arrêter
                    si k=0 alors début =1
                    sinon début= B[k]+1
                    pour v de début à N
                        B[k+1]=v
                        si test(k+1)
                            TrouverStable(k+1)
```

Le programme principal appelle
TrouverStable(k)

Au pire cas et algorithme essaye tous les $2^{|V|}$

Exercice 1 : (08 points)

A- Etant donné un tableau T de n entiers.

1. Donner un algorithme de tri par sélection.
2. En utilisant la stratégie **Diviser pour Régner, donner un algorithme de** tri par fusion de tableau T.

- Donner l'arbre d'exécution de cet algorithme sur la séquence suivante :
  (18 19 14 19 14 12 111 5).
  N.B : Le tableau contient les mêmes éléments mais rangés par ordre croissant.

3. Analyser la complexité en espace et en temps des algorithmes de tris par sélection et tris par Diviser pour Régner ? .

B- Dans l'objectif de chercher un élément dans le tableau T (Résultat de la partie A).

1. Proposer deux algorithmes (séquentielle et dichotomique) en indiquant la position de l'élément recherché.
2. Etudier la complexité en temps des deux algorithmes. (quel est le meilleur algorithme et pourquoi).