Exercice 1

- 1) Répondre par vrai ou faux en donnant une explication à votre réponse (et/ou un exemple):
 - a) L'analyse de la complexité permet d'estimer le coût en espace mémoire.
 - b) La représentation chaînée d'une liste de n éléments nécessite moins d'espace mémoire par rapport à une représentation contiguë avec une taille maximale n.
 - c) Un arbre binaire de recherche de taille n ne peut être dégénéré.
 - d) Un Tas de taille n peut être dégénéré.
 - e) La récursivité permet toujours d'avoir une complexité meilleure par rapport à l'itération.
 - f) La recherche séquentielle sur un tableau triée est meilleure en complexité -pire cas- par rapport à la recherche dichotomique.
 - g) Si une file F de 10 éléments est implémentée avec un tableau circulaire de taille 12, et les valeurs sont stockées dans F[2] jusqu'à F[11], l'insertion d'un nouvel élément se fait dans F[1].
- 2) Démontrer que si la complexité dans un arbre binaire dépend du nombre de niveaux, elle est entre n et log n.
- 3) Donner le principe du tri rapide et analyser sa complexité au pire cas.

Exercice 2

- 1) Ecrire les fonctions en C/C++ en donnant les déclarations nécessaires :
 - a) Afficher par ordre décroissant les valeurs d'une liste chaînée simple triée par ordre croissant (fonction récursive).
 - b) Retourner la valeur maximale dans un arbre binaire de recherche représenté par chaînage (itérative).
 - c) Afficher les valeurs d'un arbre binaire de recherche par ordre croissant (récursive).
- 2) On dispose d'un ensemble T de n valeurs $a_1, a_2, \dots, a_n$. On voudrait générer tous les sous ensembles possibles de T comme dans l'exemple : $T = \{1,2\}$ sousEnsT = $\{\}, \{1\}, \{2\}, \{1, 2\}$ (l'ordre n'est pas important). Compléter l'algorithme suivant (itératif) qui permet de générer ces sous ensembles en utilisant une pile et une file (on dispose aussi de l'opérateur $\cup$).

Algorithme GenererSousEns

Entrée : ensemble T de n valeurs $a_1, a_2, \dots, a_n$

Sortie : Tous les sous ensembles de T stockés dans une file F

Debut

```

initFile(F) ; initPile(P) ;
empiler(P, {} ) ;
empiler(P, a1) ;
.....
.....
    
```

CORRIGE TYPE

Exercice 1 (7 pts)

1) Vrai ou Faux **(3,75 pts)**

- a) Faux : elle permet d'estimer le temps d'exécution **2 x 0,25**
- b) Faux : elle prend plus d'espace à cause du pointeur **2 x 0,25**
- c) Faux : il peut être dégénéré car c'est un arbre binaire simple exp **2 x 0,25**
- d) Faux : il ne peut pas être dégénéré car c'est un arbre parfait (tous les niveaux sont remplis sauf éventuellement le dernier) **2 x 0,25**
- e) Faux : comme exemple le calcul des nombres de Fibonacci (version récursive : exponentielle, version itérative : linéaire) **0,25 + 0,5**
- f) Faux : recherche séquentielle est en $O(n)$, dichotomique en $O(\log n)$ **2 x 0,25**
- g) Faux : l'insertion se fait dans $F[0]$ car c'est circulaire. **2 x 0,25**

2) Démonstration **(1,5 pts)**

Un arbre binaire de n nœuds peut être soit dégénéré $\Rightarrow$ donc $h \leq n$ **0,5**

Soit dans la plus part des cas équilibré, c-a-d $n = 1 + 2 + 2^2 + \dots + 2^h = 2^{h+1} - 1$ **0,5**

$$n = 2^{h+1} \Rightarrow h = \log n - 1 \quad \mathbf{0,25}$$

$$\text{donc } h > \log n \quad \mathbf{0,25}$$

3) Le tri rapide **(1,75 pts)**

Principe 0,25

Le tableau est réarrangé en deux sous tableaux par rapport à la valeur du pivot (choisie aléatoirement ou bien par autre méthode) : les valeurs à droite $<$ pivot $\leq$ valeurs à gauche. On répète récursivement ce processus sur les deux sous tableaux.

Analyse de la complexité au pire cas

- Formule de récurrence

Le pivot divise le tableau en deux sous tableaux de taille k et $n-k$ respectivement

$$T(n) = \begin{cases} O(1) & \text{si } n = 1 \\ T(k) + T(n-k) + O(n) & \text{(réarrangement) si } n > 1 \end{cases} \quad \mathbf{0,25} \quad \mathbf{0,5}$$

- Résolution

Au pire cas, $k = 1$ et $n-k = n-1$

$$T(n) = T(n-1) + T(1) + n \quad \mathbf{0,25}$$

$$T(n) = [T(n-2) + T(1) + n-1] + T(1) + n$$

$$T(n) = T(n-2) + 2T(1) + (n-1) + n$$

$$T(n) = [T(n-3) + T(1) + n-2] + 2T(1) + (n-1) + n$$

$$T(n) = T(n-3) + 3T(1) + (n-2) + (n-1) + n$$

En générale :

$$T(n) = T(n-k) + kT(1) + (n-k+1) + \dots + (n-1) + n \quad \mathbf{0,25}$$

Arrêt lorsque $n-k = 1 \Rightarrow k = n-1$

$$\text{Donc } T(n) = 1 + (n-1) + \sum_{i=0}^{k-1} n-i = O(n^2) \quad \mathbf{0,25}$$

Exercice 2 (13 pts)

1) Fonctions en C/C++

- a) Afficher par ordre décroissant les valeurs d'une liste chaînée simple triée par ordre croissant (fonction récursive) (2,75 pts)

```
struct bloc {           0,5
    int val ;
    bloc * suiv ;
}
void afficherD (bloc * q) {
    if (q != NULL) {           0,5
        afficherD (q->val) ;   1 (il faut que l'instruction soit à sa place)
        cout <<q->val ;       0,75 (il faut que l'instruction soit à sa place)
    }
}
```

- b) Retourner la valeur maximale dans un arbre binaire de recherche représenté par chaînage (itérative) (2,75 pts)

```
struct noeud {         0,5
    int val ;
    noeud * gauche, droite ;
}
int valMax (noeud * racine) {
    noeud * courant = racine ;           0,25
    if (courant != NULL) {               0,25
        while (courant -> droite != NULL) 0,75 (accepter aussi -> gauche car ça peut être relatif)
            courant = courant -> droite ;   0,5
    }
    return courant -> val ;               0,5
}
```

- c) Afficher les valeurs d'un arbre binaire de recherche par ordre croissant (récursive) (2,5 pts)

```
void afficheInfixe(noeud * racine) {
    if (racine != NULL) {           0,25
        afficheInfixe(racine->gauche) ; 0,75 (il faut que l'instruction soit à sa place)
        cout <<racine->val ;           0,75 (il faut que l'instruction soit à sa place)
        afficheInfixe(racine->droite) ; 0,75 (il faut que l'instruction soit à sa place)
    }
}
```

- 2) Générer les sous ensembles **(5 pts) (si autre méthode utilisant toujours pile et file, estimer le pourcentage de résolution du pb. Enlever 0,5 si le résultat n'est pas dans F)**

Algorithme GenererSousEns

Entrée : ensemble T de n valeurs $a_1, a_2, \dots, a_n$

Sortie : Tous les sous ensembles de T stockés dans une file F

Debut

```
initFile(F) ; initPile(P) ;  
empiler(P, {}) ;  
empiler(P, {a1});
```

```
pour chaque elmt  $a_i \in T - \{a_1\}$  faire (0,5)  
  tant que (non pileVide(P))faire (0,5)  
    x = depiler (p) ; (0,5)  
    enfiler (f , x) ; (0,5)  
    x = x  $\cup$   $a_i$  ; (0,5)  
    enfiler(f , x) ; (0,5)  
  fint tq
```

```
si  $a_i \neq a_n$  alors (0,5)  
  tant que (non fileVide(f))faire (0,5)  
    x = defiler (f) ; (0,5)  
    empiler (p, x) ; (0,5)  
  fint tq  
finSi
```

FinPour