F1

a) Un type est un ensemble de valeurs et d'opérations sur celles-ci. Un type abstrait est un type accessible uniquement à travers une interface.

b) (1) : - push(x) : ajoute un élément sur le dessus de la pile
- pop() : retire et retourne l'élément de dessus de pile

(2) : - enqueue(x) : ajoute un élément en arrière de la file
- dequeue() : retire et retourne l'élément en tête de file

(3) : - add(x) : ajoute l'élément x à la file de priorité
- deleteMin() : retire et retourne l'élément de priorité minimale

F2

Tri	temps		mémoire
	moyen	pire cas	
rapide	$O(n \log n)$	$O(n^2)$	$O(\log n)$
tas	$O(n \log n)$	$O(n \log n)$	$O(1)$
fusion	$O(n \log n)$	$O(n \log n)$	$O(n)$
insertion	$O(n^2)$	$O(n^2)$	$O(1)$
sélection	$O(n^2)$	$O(n^2)$	$O(1)$

F3

```
classe Node :
    element
    next

    Node(e):
        element = e
        next = null

    setNext(n):
        next = n

classe LSCC :
    last = null
    size = 0

    LSCC():
        last = null
        size = 0

    addLast(n):
        if (size == 0):
            last = n
            last.setNext(last)
            size = 1
        else:
            tmp = last.next
            last.setNext(n)
            n.setNext(tmp)
            last = n
            size ++
```

F3 (suite)

suite classe LSCC

```
removeFirst():
    if (size == 0):
        Erreur("removeFirst sur LSCC vide")
    if (size == 1):
        node = last
        last = null
        size = 0
        return node.element
    else:
        node = last.next
        last.setNext(last.next.next)
        size--
        return node.element
```

```
classe FIFOconcat:
    liste = LSCC()

    enqueue(x):
        liste.addLast(Node(x))

    dequeue():
        return liste.removeFirst()

    size():
        return liste.size

    concat(P):
        tmp = last.next
        last.setNext(P.last.next)
        P.last.setNext(tmp)
        last = P.last
        liste.size = liste.size + P.liste.size
        P = null
```

F4

```
classe Tas:
    t = [] # les valeurs non initialisées sont +inf
    t[0] = -inf
    size = 0

insert(x):
    size++
    swim(size, x)

deleteMin():
    root = t[1]
    leaf = t[size]
    if (size > 1):
    |   sink(1, leaf)
    t[size] = +inf
    size--
    return root

swim(i, x):
    while (x < t[i]):
    |   p = ⌊i/2⌋
    |   t[i] = t[p]
    |   i = p
    t[i] = x

sink(i, x):
    lc = 2*i
    rc = 2*i+1
    while (x > t[lc] || x > t[rc]):
    |   if (t[rc] < t[lc]):
    |   |   t[i] = t[rc]
    |   |   i = rc
    |   else:
    |   |   t[i] = t[lc]
    |   |   i = lc
    |   lc = 2*i
    |   rc = 2*i+1
```

F4 (suite)

```
classe FileEgalitaire:
    mintas = Tas()
    maxtas = Tas()
    taille = 0

    insert(x):
        if (x ≥ mintas.getMin()):
            if (mintas.size ≤ maxtas.size):
                mintas.insert(x)
            else:
                maxtas.insert(-1 * mintas.deleteMin())
                mintas.insert(x)
        else if (maxtas.size ≤ mintas.size):
            maxtas.insert(x)
        else:
            mintas.insert(-1 * maxtas.deleteMin())
            maxtas.insert(x)
        taille ++

    deleteMedian():
        if (taille > 0):
            taille --
            if (mintas.size ≥ maxtas.size):
                return mintas.deleteMin()
            return -1 * maxtas.deleteMin()
        else:
            Erreur("deleteMedian sur tas vide!")
```

F5

a) Algo tricho(A, x, g=0, d=n): (n = A.length)

```
if (d-g == 0):
    return false
if (d-g == 1):
    if (A[g] == x):
        return true
    return false
p = ⌊(2g+d)/3⌋ ; y = A[p] ;
q = ⌊(g+2d)/3⌋ ; z = A[q] ;
if (y == x || z == x):
    return true
if (x < y):
    return tricho(A, x, g, p)
if (x > z):
    return tricho(A, x, q, d)
return tricho(A, x, p, q)
```

b) À chaque appel de tricho, on fait au maximum deux accès au tableau (si $d-g > 1$). Puis, on fait un seul appel récursif avec la longueur divisée en 3. Donc, le nombre d'accès est $\sim 2\log_3 n$ au maximum. Pour la recherche dichotomique, on a un seul pivot et on divise par 2 à chaque appel, donc on fait $\sim \log_2 n$ accès en pire cas. Pour n assez grand, la recherche trichotomique fait donc moins d'accès.

F5 (suite)

c) C'est vrai. Une fois que le tas est bâti en $O(n)$, on doit appeller $n-1$ fois sink (le plus gros du travail). En pire cas, à chaque fois on doit couler la nouvelle racine jusqu'au dernier niveau. Dans un tas binaire, on fait deux comparaisons par niveau et dans un tas ternaire, trois.

Pour n assez grand, $2\log_2 n > 3\log_3 n$. □

(exemple, $n=10000$: $2\log_2(10000) \approx 26{,}6 > 3\log_3(10000) \approx 25{,}2$)