M1.L4 : Série d'exercices sur les algorithmes / récursivité [Solution]

1 Que font ces algorithmes?

algo1 : a) sortie : la somme des nombres des n premiers nombres impairs (qui vaut n^2); b) 36; c) $O(n)$; d) pas récursif

algo2 : a) sortie : n^2 ; b) 36; c) $O(1)$; d) pas récursif

algo3 : a) sortie : la somme des nombres des n premiers nombres impairs (qui vaut n^2); b) 36; c) $O(n)$; d) récursif

algo4 : a) sortie : si n est impair, la somme des nombres des $(n + 1)/2$ premiers nombres impairs; si n est pair, la somme des $n/2$ premiers nombres pairs; b) 12; c) $O(n)$; d) récursif

f) Un seul de ces algorithmes fonctionne correctement : lequel? algo7; les autres ne s'arrêtent jamais.

g) Et celui qui fonctionne a un gros défaut : lequel? il recalcule plusieurs fois la même chose inutilement et prend également un temps exponentiel pour s'exécuter.

h) Question subsidiaire : que calcule-t-il? si l'entrée est n, la sortie est n (tout ça pour ça...).

2 Au temps des Egyptiens, troisième partie

La version récursive de l'algorithme est donnée par :

multRecursif
entrée : a, b deux entiers naturels non nuls sortie : a × b

Si b = 1
  Sortir : a
 	
Si ((b mod 2) = 1) // b est impair
  Sortir : a + **multRecursif(**2.a , b/2**)**
 	
Sortir : multRecursif(2.a , b/2 **)**

Pour obtenir cet algorithme il faut remarquer que la version itérative est pilotée par l'opérande **b**. Cela détermine le critère d'arrêt sur **b**.

Ensuite le test est aussi fait sur **b** pour différencier les deux cas (pair, impair).

Pour identifier les paramètres des 2 appels récursifs on peut s'inspirer de l'algorithme itératif en l'exécutant pour la valeur 3 de b (et en gardant une variable **a** pour l'autre opérande). Dans ce scénario **b** prend ensuite les valeurs 2 et 1, et cela couvre les 3 cas principaux.

3 Au temps des Grecs

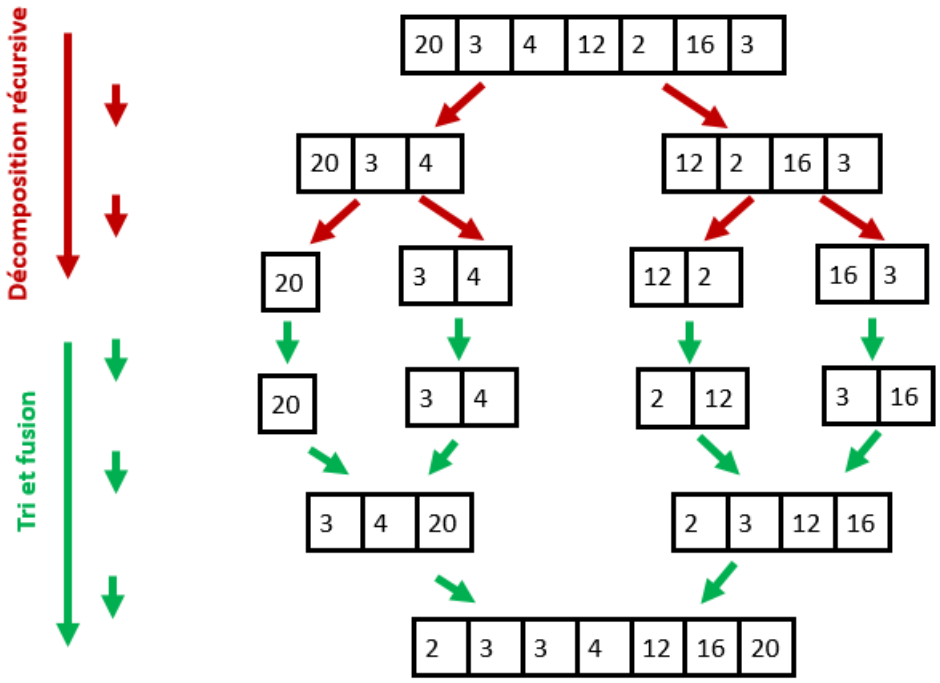
Algorithme d'Euclide –

Voici une version où a et b sont toujours non nuls.

pgcdRecursif
entrée : a, b deux entiers naturels non nuls sortie : pgcd (a,b)
Si a ≥ b reste ← a mod b **Si** reste = 0 **Sortir** : b **Sortir : pgcdRecursif (b, reste)** **Sinon** reste ← b mod a **Si** reste = 0 **Sortir** : a **Sortir : pgcdRecursif (a, reste)**

4 Tri fusion

a) le tri fusion procède d'abord à la décomposition de la liste jusqu'à atteindre une taille de 1 ou de 2 éléments ; ensuite il recompose la liste en combinant les sous-listes triées :



b) peu importe que la seconde liste soit déjà triée ou pas il y aura les même étapes de décomposition et de recomposition.

La seule différence porte sur le nombre des test de swap dans la phase de tri avec deux éléments puis dans la recomposition mais le coût global reste du même ordre de complexité en O(nlog(n)).

5 Taille de liste : le retour

La version récursive peut s'écrire : **TailleDichotomiqueRec(L, a, b)**

TailleDichotomique
entrée : L, a, b sortie : le nombre d'éléments de L
Si $a \geq b - 1$ **Sortir** :a $c \longleftarrow a + \lfloor \frac{b-a}{2} \rfloor$ **Si a_element(L, c)** **sortir** : **TailleDichotomiqueRec(L, c, b)** **Sinon** **sortir** : **TailleDichotomiqueRec(L, a, c)**

Rappel : cet algorithme cherche la taille entre a inclus et b exclu.