Exercices Supplémentaires sur la Complexité des Algorithmes

Exercice 1 : Analyser la complexité de chacun des algorithmes suivants en donnant une brève explication à votre réponse.

Algorithme	Complexité	Explication
Somme de deux vecteurs de n éléments	**O(n)**	**Une seule boucle qui se répète n fois**
Somme de deux matrices carrées (n, n)	**$O(n^2)$**	**Deux boucles imbriquées qui se répète chacune n fois**
Produit de deux matrices carrée (n, n)	**$O(n^3)$**	**Trois boucles imbriquées qui se répète chacune n fois** **Pour i = 1 à n faire** **Pour j = 1 à n faire** **C[i][j] = 0 ;** **Pour k = 1 à n faire** **C[i][j] = C[i][j] + A[i][k] * B[k][j]** **Fin k** **Fin j** **Fin i**
Calcul itératif du factoriel de n	**O(n)**	**On effectue la multiplication n fois**
Calcul itératif de la somme 1+ 2+ ….+ n	**O(n)**	**On effectue la somme n fois**
Insertion en tête d'une liste chainée simple de taille n	**O(1)**	**On ne parcourt pas la liste**
Insertion en tête d'un tableau de taille n	**O(n)**	**Décalage à droite des n éléments**
Insertion à la fin d'une liste chainée simple de taille n	**O(n)**	**Parcours des n éléments, ensuite insertion**
Insertion à la fin d'un tableau de taille n	**O(1)**	**Accès direct à (n+1)ème case et insertion**
Tester si un nombre est pair ou impair	**O(1)**	**Un test « Div 2»**

Exercice 2 : Analyser la complexité des algorithmes suivants :

`void exp1() {`
`  int i ;`
`  for (int i = 1 ; i <= 100 ; i++)`
`  cout<<i*2 ;`
`}`
	La boucle «for i» se répète un nombre fixe de fois, donc la complexité est O(1)

`void exp2(int n, int s) {`
`  int p = n ; s = 0 ;`
`  while (p >= 1) {`
`  int i = 1 ;`
`  while (i <= n) {`
`  s = s + p ; i++ ;`
`  }`
`  p = p /2 ;`
`  }`
`}`
	- La boucle «while p» : p = n, n/2, n/4.... jusqu'à $\frac{n}{2^p} = 1 \Rightarrow p = log_2(n)$
	- La boucle «while i» s'exécute n fois
	- Les deux boucles étant imbriquées, on aura $T(n) = O(nlog(n))$

`void exp3(int n) {`
`  for (int i = 1 ; i <= n ; i++)`
`  for (int j = 1 ; j <= i ; j++)`
`  exp1() ;`
`}`
	La boucle «for i» se répète n fois, où dans chaque itération la boucle « for j» se répète i fois, on aura donc (1+2+3+....+n) O(1) (de exp1()), ce qui donne une complexité de $O(n^2)$

`void exp4() {`
`  int i ;`
`for(int i = n ; i > 0 ; i=i/2) {`
`  for(int j = 1 ; j < n ; j=j*2) {`
`  for(int k = 0 ; k < n ; k=k+2) {`
`  //un nombre constant`
`  d'instructions`
`  }`
`  }`
`}`
	- La boucle «for i» : i = n, n/2, $n/2^2$, ... jusqu'à $n/2^p \approx 1$ (p : le nombre d'itérations), $\Rightarrow p \approx \log(n)$
	- La boucle «for j» : j= 1, 2, 2^2, ... jusqu'à $2^p = n$ (p : le nombre d'itérations), $\Rightarrow p = \log(n)$
	- La boucle «for k» se répète n/2 fois puisque le pas = 2
	- Les trois boucles étant imbriquées : $T(n) = \frac{n}{2}(log(n))^2 = O(n(\log(n))^2)$

`for (i=0; i<=n-1; i++){`
`  for (j=i+1; j<=n-1; j++){`
`  //un nombre constant`
`  d'instructions`
`  }`
`}`
	- La boucle «for i» se répète n fois, où dans chaque itération la boucle «for j» se répète (n-1)-(i+1) +1 = n-i-1, c.ad : lorsque i=0, on a (n-1) itérations ; i=1, (n-2) itérations, i=2, (n-3) itérations.... i= n-2, 1 itération. Le nombre total d'itérations = (n-1) + (n-2)+....3+2+1= $O(n^2)$

Exercice 3 : analyser la complexité des fonctions récursives suivantes:

```
int somme (int n) {
 if (n==0) return 0 ;
else return n + somme (n-1)
}
```

1) Formule de récurrence

$$T(n) = \begin{cases} O(1) & si\ n = 0 \\ 1 + T(n-1) & sinon \end{cases}$$

2) Résolution par la méthode de substitution et calcul de la complexité

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

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

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

.....

$\boxed{T(n) = p + T(n-p)}$ avec p le nombre d'itérations (1)

La dernière itération s'effectue lorsque la taille de la donnée = 0, c.ad n-p =0 $\Rightarrow$ p=n. On remplace dans la formule (1), on obtient :

$$T(n) = n + T(0) = n + 1, et\ donc\ T(n) = 0(n)$$

```
Algorithme F
Entrée : tableau T de taille n, les indices sont [1..n]
Début
  Si n == 1 alors afficher T[1]
  Sinon  début
            afficher(T[n]) ;
            F (Tableau T privé du dernier élément) ;
            F (Tableau T privé du premier élément) ;
          Fin
Fin
```

2) Formule de récurrence

$$T(n) = \begin{cases} O(1) & si\ n = 1 \\ 1 + 2T(n-1) & sinon \end{cases}$$

2) Résolution par la méthode de substitution et calcul de la complexité

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

$= 1 + 2[1 + 2T(n-2)] = 3 + 4T(n-2)$

$= 3 + 4[1 + 2T(n-3)] = 7 + 8T(n-3)$

$= 7 + 8[1 + 2T(n-4)] = 15 + 16T(n-4)$

.....

$\boxed{T(n) = 2^P - 1 + 2^P T(n-p)}$ avec p le nombre d'itérations (1)

La dernière itération est réalisée lorsque la taille de la donnée = 1, c.ad n-p =1 $\Rightarrow$ p=n-1. On remplace dans la formule (1), on obtient :

$T(n) = 2^{n-1} - 1 + 2^{n-1} * 1, et\ donc\ T(n) = 0(2^n)$

Exercice 4

Analyser la complexité de F au pire et au meilleur cas :

```
bool F(int T[], int n) {
  for (int i = 0; i < n-1; i++)
    if (T[i] == T[i+1])return true;
  return false;
}
```

Pire cas	Meilleur cas
Le nombre maximal d'opérations est réalisé lorsque la boucle « for i » s'exécute complètement, c.ad n-1 fois, dans ce cas T(n) = 0(n)	Le nombre minimal d'opérations est effectué lorsque le test T[i] == T[i+1] est « true » dès la première itération, dans ce cas T(n) = 0(1)