Exercice 1 :

1) Somme (n : entier) : entier
-> Entrée : n (nombre limite)
-> Sortie : Somme renvoie la variable locale som qui stocke la somme des multiples
-> Variables locales : i, som : entier

```
Début
   i <- 0;
   som <- 0;
   Tant que i < n
      Si i mod 3 = 0
         som <- som + i;
      Sinon si i mod 5 = 0
         som <- som + i;
      Fin Si
      i <- i + 1;
   Fin Tant que
   Retourner som;
Fin
```

2) Il y a 4n opérations arithmétiques dans le pire des cas. La borne supérieure est O(n).

$\sum_{i=1}^{n} 3i + \sum_{i=1}^{n} 5i \quad \sum_{i=1}^{n} 1$ $\quad \to 3\frac{n(n+1)}{2} + 5\frac{n(n+1)}{2} - 15\frac{n(n+1)}{2}$

Exercice 2 :

A) Il y a n/2 + n mod 2 + 1 affectations. O(n)

B) Il y a n + 2 affectations. O(n)

C) Il y a 1 + n^2 + 2n affectations. O(n^2)

D) Il y a 1 + 2n + (n*(1-n)/2) = 1 + n*((n+5)/2) = 1 + (3/2)n + (n^2/2).
affectations. O(n^2)

E) Il y a 1 + min (m, n) affectations. O (min (m ,n))

F) Il y a 1 + max (m, n) affectations. O (max (m, n))

G) Il y a 2 + n + m affectations. O(n + m)

H) Il y a 2 + m*n + m/n affectations. O(n * m)

Les algorithmes ayant un coût linéaire sont A, B, E, F, G, H.
Les algorithmes ayant un coût quadratique sont C, D.

Exercice 3 :

1) Cet algorithme calcule 2^n - 1.

L'algo initialise r à 0. Puis le boucle ayant l'affectation r <- 2r +1
est répété n-1 fois. i est initialisé à 1 et sort de la boucle quand Il
est égal à n. Ce qu'on aura est u(n).

2) Cet algorithme calcule la même chose que l'algorithme précédent.
Sa complexité est O(2^n) si on considère que l'opération fondamentale
ets l'affectation. Si l'opération fondamentale est la condition ||
l'opération arithmétique alors sa complexité est O(n).

Exercice 4 :

1)

```
Minimum ( t : tab ) : entier
-> Entrée : t ( le tableau qu'on va examiner pour en déterminer son minimum )
-> Sortie : La fonction renvoie une variable locale min dans laquelle est stockée la plus petite valeur
du tableau
-> Variables locales :   i, min : entier;
Début
   min <- t[0];
   Pour i de 1 à t.n faire
      Si t[i] < min
         min <- t[i];
      Fin Si
   Fin Pour
   i <- i + 1;
   Retourner min;
Fin
```

2) Cet algorithme fait t.n comparaisons.

3) Il suffit de modifier "<" en le remplaçant par ">". (renommer "min" par "max" pour le sens)
OU on fait t[i]*(-1) à toute les valeurs du tableau et on réutilise .

4)

```
Extremum ( t : tab ) : entier
-> Entrée : t ( le tableau qu'on va examiner pour en déterminer son minimum )
-> Sortie : La fonction renvoie les variables locales min et max dans lesquelles
sont stockées la plus petite valeur du tableau et la plus grande valeur du tableau
respectivement.
-> Variables locales :   i, min, max : entier;
Début
   min <- t[0];
   max <- t[0];
   Pour i de 1 à t.n faire
      Si t[i] < min
         min <- t[i];
      Sinon si t[i] > max
         max <- t[i];
      Fin Si
```

```
        i <- i + 1;
    Fin Pour
    Retourner min;
    Retourner max;
Fin
```

Dans le meilleur des cas, on a t.n comparaisons.
Dans le pire des cas, on a t.n*2 comparaisons.

5)
Extremum_Rem (t : tab) : entier
-> Entrée : t (le tableau qu'on va examiner pour en déterminer son minimum)
-> Sortie : La fonction renvoie les variables locales min et max dans lesquelles sont stockées la plus petite valeur du tableau et la plus grande valeur du tableau respectivement.
-> Variables locales :   i, min, max : entier;

```
Début
    min <- t[0];
    max <- t[0];
    Pour i de 1 à t.n faire
        Si t[i] <= t[i+1]
            Si t[i] < min
             min <- t[i];
            Sinon si t[i+1] > max
             max <- t[i+1];
            Fin Si
        Sinon
            Si t[i+1] < min
                min <- t[i+1];
            Sinon si t[i] > max
                max <- t[i];
            Fin Si
        Fin Si
        i <- i + 2;
    Fin Pour
    Retourner min;
    Retourner max;
Fin
```

Le complexité de cet algorithme est 3n/2

6)
Maximum_deux (t : tab) : entier
-> Entrée : t (le tableau qu'on va examiner pour en déterminer ses deux maximums)
-> Sorties : La fonction retourne les variables locales max1 et max2 dans lesquelles est stockées la première plus grande valeur du tableau et la deuxième plus grande valeur du tableau respectivement.
-> Variables locales : i, max1, max2 : entier;

```
Début
    max1 <- t[0];
    max2 <- t[0];
    Pour i de 1 à t.n faire
```

```
    Si t[i] <= t[i+1]
      Si t[i] > max2
        max2 <- t[i];
      Si t[i+1] > max1
        max1 <- t[i+1];
      Fin Si
    Sinon
      Si t[i+1] > max2
        max2 <- t[i+1];
      Si t[i] > max1
        max1 <- t[i];
      Fin Si
    Fin Si
    i <- i + 2;
  Fin Pour
  Retourner max1;
  Retourner max2;
Fin
```

La complexité de cet algorithmes est de 3n/2.

7) 8)

Maximum_huit (t : tab) : entier

-> Entrée : t (le tableau de 8 éléments qu'on va examiner pour en déterminer ses deux maximums)

-> Sorties : La fonction retourne les variables locales max1 et max2 dans lesquelles est stockées la première plus grande valeur du tableau et la deuxième plus grande valeur du tableau respectivement.

-> Variables locales : i, max1, max2 : entier;

```
Début
  Si ( t[0] < t[1] )
    max1 <- t[1];
    max2 <- t[0];
  Sinon
    max1 <- t[0];
    max2 <- t[1];
  Fin Si
  Pour i de 2 à t.n faire
    Si t[i] <= t[i+1]
      Si t[i] > max2
        max2 <- t[i];
      Si t[i+1] > max1
        max1 <- t[i+1];
      Fin Si
    Sinon
      Si t[i+1] > max2
        max2 <- t[i+1];
      Si t[i] > max1
        max1 <- t[i];
      Fin Si
    Fin Si
    i <- i + 2;
```

```
    Fin Pour
    Retourner max1;
    Retourner max2;
Fin
```

La complexité de cet algorithme est 1 + 3*((n-1)/2)