PARTIE 1:

Nous avons écrit l'ensembles des fonctions écrites grâce aux différentes aides fournis par le cours et internet.
L'ensemble du main et des tests seront fournies à la fin de cette partie.

Voici l'ensemble des bibliothèques et des define utilisé lors de cette partie 1:

```
#include<stdio.h>
#include<math.h>
#include<stdlib.h>
#include<time.h>
#include<assert.h>

#define N 100000
```

Exercice 1:
Voici le premier code écrit et commenté (fonction bitcommun) :

```
// Fonction pour compter les bits communs (méthode non optimisé)
int bitcommun(double x, double y) {
// Conversion des doubles en unsigned long long (pour manipuler les bits) (et les comparer)
unsigned long long x_bits = *(unsigned long long*)&x;
unsigned long long y_bits = *(unsigned long long*)&y;

// Variable pour stocker le nombre de bits communs
int common_bits = 0;

// Masque pour extraire chaque bit, on initialise la variable mask avec la valeur 1 sous
//la forme d'un entier non signé de 64 bits.
unsigned long long mask = 1ULL;

// Boucle à travers les 64 bits
for (int i = 0; i < 64; i++) {
// Comparaison bit à bit et incrémentation si les bits sont égaux
//(ici != car les bits sont comparés 1 à 1 après l'autre)
if ((x_bits & mask) != (y_bits & mask)) {
common_bits++;
}
mask «= 1; // Décalage du masque pour passer au bit suivant
}

return common_bits; // Retourne le nombre de bits communs
}
```

Exercice 2:
Voici le second code écrit et commenté, il s'agit d'une version optimisé (verifié dans l'exercice 3) du premier code (fonction bitcommmun_opt) :

```
// Fonction pour compter les bits communs (méthode optimisée)
int bitcommun_opt(double x, double y) {
//Comme pour la première méthode, convertit les double pour les manipuler.
unsigned long long int x_bits = *((unsigned long long int*)&x);
unsigned long long int y_bits = *((unsigned long long int*)&y);

//Réalise une opération XOR (^) entre les valeurs binaires stockées
//dans x_bits et y_bits et stocke le résultat dans la variable diff.
unsigned long long int diff = x_bits ŷ_bits;

int count = 0; //initialise le compteur de bits commun.

//La boucle while compte simplement le nombre de bits en commun en poursuivant diff
while (diff) {
count += diff & 1;
diff »= 1;
}

return count; // Retourne le nombre de bits communs
}
```

Exercice 3 :
Nous avons eu un petit problème de compréhension concernant la méthode de résolution de cette exercice. Nous avions compris qu'il fallait donner le temps en seconde de résolution de chaque fonction, afin de prouver que l'une est plus optimisée que l'autre. Nous avons pour cela utilisé la fonction clock().

Voici à présent le main global de la partie :

```
int main() {
srand(time(NULL)); //initalisation de la valeur srand à time(NULL)

double x = 1.41421356;
double y = 1.41427845;

double a[N], b[N], un=1;

double time_spent = 0.0; //calcul du temps passé

clock_t begin = clock();

for(int i=0; i< N; i++){
a[i] = (un*rand()*10000)/RAND_MAX;
b[i] = a[i] + (un*rand()*0.0001)/RAND_MAX;

int common_bits = bitcommun_opt(a[i], b[i]);
}
clock_t end = clock();

// calcule le temps écoulé en trouvant la différence (end - begin) et
// divisant la différence par CLOCKS_PER_SEC pour convertir en secondes
time_spent += (double)(end - begin) / CLOCKS_PER_SEC;

printf("Temps écoulé pour la boucle for (fonction simple) est de %f secondesn", time_spent);

double time_spent2 = 0.0;

clock_t begin2 = clock();

for(int i=0; i< N; i++){
a[i] = (un*rand()*10000)/RAND_MAX;
b[i] = a[i] + (un*rand()*0.0001)/RAND_MAX;

int common_bits_opt = bitcommun_opt(a[i], b[i]);
}
clock_t end2 = clock();

// calcule le temps écoulé en trouvant la différence (end - begin) et
// divisant la différence par CLOCKS_PER_SEC pour convertir en secondes
time_spent2 += (double)(end2 - begin2) / CLOCKS_PER_SEC;

printf("Temps écoulé pour la boucle for (fonction optimisée) est de %f secondesn", time_spent2);

// Appel de la fonction pour compter les bits communs (méthode 1)
int common_bits = bitcommun(x, y);

// Appel de la fonction pour compter les bits communs (méthode 2, optimisée)
int common_bits_opt = bitcommun_opt(x, y);

// Affichage du résultat
printf("Nombre de bits en commun entre %f et %f est %dn", x, y, common_bits);
printf("Nombre de bits en commun entre %f et %f est %d (optimisé)n", x, y, common_bits_opt);

//tests des fonctions :
assert(common_bits == common_bits_opt);
assert(bitcommun(x,y)==22);
assert(bitcommun_opt(x,y)==22);

return 0;
```

PARTIE 2 :
Par manque de temps la partie 2 n'est que le code commenté :

```
#include <stdio.h>
#include <math.h>
//Q.Bonus
double Horner(double *p, int n, double val) {
double result = p[0]; // Initialisation du résultat avec le premier coefficient
for (int i = 1; i <= n; i++) { // Boucle pour chaque coefficient du polynôme
result = result * val + p[i]; // Mise à jour du résultat selon la méthode de Horner
}
return result; // Retourne le résultat final
}

//Q1 (On a pas reussi d'utilise avec la fonction "Horner",mais on a trouve les resultats)
// Définition de la fonction "suite" avec trois paramètres : X0, P, Y
double suite(double X0, double P, double Y) {
double Xn = X0; // Initialise Xn avec la valeur initiale X0
double Xn1; // Déclare une variable pour stocker le terme suivant
int max_iterations = 1000; // Limite le nombre d'itérations
int iteration = 0; // Initialise le compteur d'itérations

// Démarre une boucle "for" pour effectuer les itérations
for (iteration = 0; iteration < max_iterations; iteration++) {
// Calcule le prochain terme de la suite en utilisant la formule donnée
Xn1 = (4 * Xn * Xn * Xn - (3 * P - 2) * Xn * Xn - P * Xn - 2 * P) /
(5 * Xn * Xn - (4 * P - 3) * Xn - 2 * P - 2);

// Affiche le terme actuel et le numéro d'itération correspondant avec une précision de 10 chiffres
après la virgule
printf("Terme %d : %.10lfn", iteration, Xn1);

// Vérifie si la différence entre le terme actuel et le terme précédent est inférieure à Y
if (fabs(Xn1 - Xn) < Y) {
// Si la condition est satisfaite, la convergence est atteinte, la fonction renvoie le dernier terme
calculé
return Xn1;
}

// Met à jour Xn avec le terme actuel pour la prochaine itération
Xn = Xn1;
}

// Si la convergence n'est pas atteinte après les itérations maximales, renvoie le dernier terme calculé
return Xn1;
}

int main() {
double X0, P, Y; // Déclare des variables pour les valeurs d'entrée

// Demande à l'utilisateur d'entrer la valeur initiale X0 et la stocke dans X0
printf("Entrez la valeur initiale X0 : ");
scanf("%lf", &X0);

// Demande à l'utilisateur d'entrer la valeur de P et la stocke dans P
printf("Entrez la valeur de P : ");
scanf("%lf", &P);

// Demande à l'utilisateur d'entrer la valeur de Y et la stocke dans Y
printf("Entrez la valeur de Y : ");
scanf("%lf", &Y);

// Appelle la fonction "suite" avec les valeurs d'entrée et stocke le résultat dans Xn
double Xn = suite(X0, P, Y);

// Affiche le terme final de la suite avec une précision de 15 chiffres après la virgule
printf("Terme final : %.15en", Xn);
```

return 0; // Indique que le programme s'est exécuté avec succès
}

/*
Q2.

x0 = 2, α = 0,3, ε = 10$\hat{}$(-14): Convergence rapide vers une limite finie est un type de convergence.
Explication : La faible valeur de α (0.3) favorise la convergence. De plus, la faible tolérance ε (10$\hat{}$(-14)) garantit une convergence rapide vers une limite finie.

x0 = 5, α = 3,5, ε = 10$\hat{}$(-14): Convergence lente ou divergence est le type de convergence.
Explication : La valeur de α (3.5) est élevée, suggérant une divergence potentielle. En raison de la grande valeur de α, la divergence peut se produire même si x0 est proche de la solution.

x0 = 0,5, α = 3,5 et ε = 10$\hat{}$(-14):
Le type de convergence est la divergence, également connue sous le nom de convergence lente.
Explication : La valeur de α (3.5) est élevée, comme dans le cas précédent, suggérant une divergence potentielle. Il est peu probable que la divergence soit stoppée malgré une petite tolérance ε.

x0 = -3,0, α = 1,0 et ε = 10$\hat{}$(-14):

Convergence rapide vers une limite finie.
La valeur de α est égale à 1.0, ce qui est critique mais avantageux pour la convergence. Une convergence rapide vers une limite finie est assurée par la faible tolérance ε (10$\hat{}$(-14).

x0 = 2.0, α = 1.0, ε = 10$\hat{}$(-14):

Convergence rapide vers une limite finie.
La valeur de α est égale à 1.0, ce qui est critique mais favorable à la convergence. La petite tolérance ε (10$\hat{}$(-14)) garantit une convergence rapide vers une limite finie.