TP d’Algorithmique

n° 1

Table des matières

Exercice 1 : Somme des n premiers entiers 2

Rappel de l’énoncé 2

Analyse du travail à faire 2

Algorithmes 2

Programmes 3

Jeux d’essai 4

Exercice 2 : Calcul de la puissance 5

Rappel de l’énoncé 5

Analyse du travail à faire 5

Algorithmes 5

Programmes 6

Jeux d’essai 8

Remarques 9

Exercice : Calcul du PGCD 9

Rappel de l'énoncé 9

Analyse du travail à faire 9

Algorithmes 9

Programmes 10

Jeux d'essai 12

Remarques 13

**Remarque :**

Nous n'avons pas réussi à faire le calcul de la puissance de manière itérative en utilisant l'astuce donnée.

# 

# Exercice 1 : Somme des n premiers entiers

## Rappel de l’énoncé

Écrire une fonction récursive qui calcule les n premiers entiers. Déterminer d'abord une relation de récurrence, puis la ou les conditions d'arrêt.

- Relation : somme(n) = n + somme(n-1)

- Condition d'arrêt : n = 0

## Analyse du travail à faire

On pourra réutiliser la procédure écrite pour le calcul de la factorielle. Pour la version itérative, on fera une boucle de 1 à n qui ajoute le nombre index (i) au résultat.

## Algorithmes

**// Calcul de la somme des n premiers entiers positifs**

**// Entrée : n, le nombre limite > 0**

**// Sortie : La somme des entiers de 1 à n, ou -1 si le nombre n'est pas valide**

**// Pré-condition: /**

**// Post-Condition: /**

**Fonction** somme\_iter(n : **Entier**) **Retourne** **Entier**

**Début**

**Avec**

resultat : **Entier**

**Si** n < 0 **Alors** **Retourne** -1

resultat <- 0

**Pour** i **de** 0 **à** n **Faire**

resultat <- resultat + i

**Fin Faire**

**Retourne** resultat

**Fin** somme\_iter

**Fonction** somme\_recur(n : **Entier**) **Retourne** **Entier**

**Début**

**Si** n = 0 **Alors** **Retourne** 0

**Sinon** **Retourne** n + somme\_recur(n-1)

**Fin** **Si**

**Fin** somme\_recur

## Programmes

#include <iostream>

#include <ctime>

using namespace std;

int somme\_iter(int n)

{

if (n < 0) {

cout << "Le nombre doit être positif.";

return -1;

}

int resultat = 0;

int i;

for(i=0; i <= n; i++)

{

resultat = resultat + i;

}

return resultat;

}

int somme\_recur(int n)

{

if (n < 0) {

cout << "Le nombre doit être positif." << endl;

return -1;

}

if (n == 0)

return 0;

else

return n + somme\_recur(n-1);

}

int main()

{

int nbr;

cout << "Entrez un nombre : ";

cin >> nbr;

cout << "Somme des entiers de 1 à " << nbr << " : " << endl;

clock\_t temps\_debut = clock();

int i;

for (i = 0; i < 10000000; i++) {

somme\_recur(nbr);

}

cout << "Recursif : " << somme\_recur(nbr) << endl;

cout << "\t" << float(clock() - temps\_debut) / CLOCKS\_PER\_SEC << "s" << endl;

temps\_debut = clock();

for (i = 0; i < 10000000; i++) {

somme\_iter(nbr);

}

cout << "Itératif : " << somme\_iter(nbr) << endl;

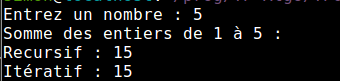
cout << "\t" << float(clock() - temps\_debut) / CLOCKS\_PER\_SEC << "s" << endl;

return 0;

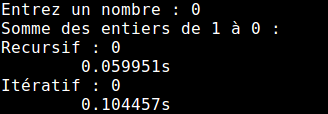
}

## Jeux d’essai

|  |  |  |
| --- | --- | --- |
| **Cas testé** | **Entrée** | **Résultat** |
| > 0 | 5 | 15 |
| < 0 | -3 | Le nombre doit être positif |
| = 0 | 0 | 0 |



Oncle D:Users:Junior:tp_dutinfo:TP01:Ex1_scr2.png



# Exercice 2 : Calcul de la puissance

## Rappel de l’énoncé

On se propose de calculer récursivement a^n. Utiliser l'astuce suivante :

- a^n =

- Si n est pair = a^(n/2) \* a^(n/2)

- Si n est impair = a \* a^( (n-1)/2) \* a^( (n-1) / 2)

Il est demandé que a^(n/2) ou a^((n-1)/2) ne soient exécutés qu'une seule fois pour

des raisons de performance.

## Analyse du travail à faire

Dans la version récursive, nous utiliserons une variable temporaire afin de ne calculer a^(n/2) ou a^((n-1)/2) qu'une seule fois.

La version itérative sans astuce sera un algorithme assez naïf qui multiplie un nombre par lui-même n fois.

## Algorithmes

**// Fonction calculant a à la puissance n**

**// Entrée : a, un réel et n, un entier > 0**

**// Sortie : le résultat de a^n, ou -1 si une erreur s'est produite**

**// Pré-condition : Aucune**

**// Post-condition : Aucune**

**Fonction** puissance\_recur(a : **Réel**, n : **Entier**) **Retourne** **Réel**

**Début**

**Avec** b : **Réel**

**Si** n < 0 **Alors** **Retourne** -1 **Fin** **Si**

**Si** n = 0 **Alors** **Retourne** 1

**Sinon**

**Si** n = 1 **Alors** **Retourne** a

**Sinon**

b <- puissance(a, n/2)

**Si** n mod 2 = 0 **Alors**

**Retourne** b \* b

**Sinon**

**Retourne** a \* b \* b

**Fin** **Si**

**Fin** **Si**

**Fin** **Si**

**Fin** puissance\_recur

**Fonction** puissance\_iter\_sansastuce(a : **Réel**, n : **Entier**) **Retourne** **Réel**

**Début**

**Avec** resultat : **Réel**

i : **Entier**

**Si** n < 0 **Alors** **Retourne** -1 **Fin** **Si**

resultat <- 1

**Pour** i **de** 1 **à** n **Faire**

resultat <- resultat \* a

**Fin** **Faire**

**Retourne** resultat

**Fin** puissance\_iter\_sansastuce

## Programmes

#include <iostream>

#include <ctime>

using namespace std;

double puiss\_recur(const double a, const int n)

{

double b;

if (n < 0)

return -1;

if (n == 0)

return 1;

else if(n == 1)

return a;

else {

b = puiss\_recur(a, n/2);

if(n%2 == 0)

return b\*b;

else

return a\*b\*b;

}

}

double puiss\_iter\_sansastuce(const double a, const int n)

{

double resultat = 1;

int i;

if (n < 0)

return -1;

for(i=0; i < n; i++)

{

resultat = resultat\*a;

}

return resultat;

}

int main()

{

double a;

int n;

cout << "Entrez le nombre : ";

cin >> a;

cout << "Entrez son exposant : ";

cin >> n;

if (puiss\_recur(a, n) == -1) {

cout << "L'exposant doit être positif." << endl;

return 1;

}

int i;

cout << a << " puissance " << n << " = " << endl;

clock\_t temps\_debut = clock();

for (i = 0; i < 1000000000; i++) {

puiss\_iter\_sansastuce(a, n);

}

cout << "Itératif : " << puiss\_iter\_sansastuce(a, n) << endl;

cout << "\t" << float(clock() - temps\_debut) / CLOCKS\_PER\_SEC << "s" << endl;

temps\_debut = clock();

for (i = 0; i < 1000000000; i++) {

puiss\_recur(a, n);

}

cout << "Récursif : " << puiss\_recur(a, n) << endl;

cout << "\t" << float(clock() - temps\_debut) / CLOCKS\_PER\_SEC << "s" << endl;

return 0;

}

## Jeux d’essai

|  |  |  |
| --- | --- | --- |
| **Cas testé** | **Entrée** | **Résultat** |
| Exposant > 0 | 10 et 2 | 100 |
| Exposant = 0 | -8 et 0 | 1 |
| Exposant < 0 | 12.5 et -4 | Erreur |

## Oncle D:Users:Junior:tp_dutinfo:TP01:Ex2_scr1.png

## Oncle D:Users:Junior:tp_dutinfo:TP01:Ex2_scr2.png

Oncle D:Users:Junior:tp_dutinfo:TP01:Ex2_scr3.png

## Remarques

On peut remarquer que dans un certain cas, l'algorithme récursif est un peu plus rapide que l'algorithme itératif. Ceci peut s'expliquer par le fait que l'algorithme récursif possède une optimisation que n'a pas l'algorithme itératif. De ce fait, il est plus rapide sur des petits calculs.

On remarque par contre que l'itératif est toujours plus rapide sur des calculs plus importants, ceci est dû à la « gourmandise » des algorithmes récursifs.

# Exercice : Calcul du PGCD

## Rappel de l'énoncé

Calculer le PGCD (Plus Grand Commun Diviseur) entre deux nombres en utilisant la récursivité.

Astuce :

- si a = b, alors a (ou b)

- si a > b, alors c'est le PGCD entre a-b et b

- sinon, c'est le PGCD entre a et b-a

## Analyse du travail à faire

L'algorithme récursif aura pour condition d'arrêt a = b, et se rappellera en changeant le paramètre a ou b par a-b ou b-a (comme indiqué dans l'astuce).

L'algorithme itératif pourra être réalisé avec deux variables temporaires dont on changera la valeur en fonction de a et b.

## Algorithmes

**// Fonction calculant le PGCD de deux nombres donnés**

**// Entrée : les nombres a et b, nécessairement > 0**

**// Sortie : le PGCD de a et b, ou -1 si une erreur est survenue**

**// Pré-condition : /**

**// Post-condition : /**

**Fonction** pgcd\_recur(a : **Entier**, b : **Entier**) **Retourne** **Entier**

**Début**

**Si** a<=0 **ou** b<=0 **Alors** **Retourne** -1 **Fin** **Si**

**Si** a=1 **ou** b=1 **Retourne** 1 **Fin** **Si**

**Si** a=b **Alors** **Retourne** a

**Sinon**

**Si** a > b **Alors** **Retourne** pgcd\_recur(a-b, b)

**Sinon** **Retourne** pgcd\_recur(a, b-a)

**Fin** **Si**

**Fin** **Si**

**Fin** pgcd\_recur

**Fonction** pgcd\_iter(a : **Entier**, b : **Entier**) **Retourne** **Entier**

**Début**

**Si** a<=0 **ou** b<=0 **Alors** **Retourne** -1 **Fin** **Si**

**Tant** **que** (a!=b) **ou** (a!=1 **et** b!=1) **Faire**

**Si** a > b **Alors**

a <- a-b

**Sinon**

b <- b - a

**Fin** **Si**

**Fin** **Faire**

**Si** a = b **Alors**

**Retourne** a

**Fin** **Si**

**Si** a=1 **ou** b=1 **Alors**

**Retourne** 1

**Fin** **Si**

**Fin** pgcd\_iter

## Programmes

#include <iostream>

#include <ctime>

using namespace std;

int pgcd\_recur(const int a, const int b)

{

if (a <= 0 || b <= 0)

return -1;

if (a == 1 || b == 1)

return 1;

if(a == b)

return a;

else if(a > b)

return pgcd\_recur(a-b, b);

else

return pgcd\_recur(a, b-a);

}

int pgcd\_iter(const int a, const int b)

{

int tempa=a;

int tempb=b;

if (a <= 0 || b <= 0)

return -1;

while((tempa != tempb) && (tempa != 1 && tempb != 1))

{

if (tempa > tempb)

tempa = tempa-tempb;

else

tempb = tempb-tempa;

}

if (tempa == tempb)

return tempa;

else if (a==1 || b==1)

return 1;

else

return -1;

}

int main()

{

int a;

int b;

cout<<"Entrez le premier nombre : ";

cin>>a;

cout<<"Entrez le deuxième nombre : ";

cin>>b;

int i;

cout << "PGCD de " << a << " et " << b << endl;

if (pgcd\_recur(a, b)) {

cout << "Veuillez entrer des nombres positifs." << endl;

return 1;

}

clock\_t temps\_debut = clock();

for (i = 0; i < 10000000; i++) {

pgcd\_iter(a,b);

}

cout << "Itératif : " << pgcd\_iter(a,b) << endl;

cout << "\t" << float(clock() - temps\_debut) / CLOCKS\_PER\_SEC << "s" << endl;

temps\_debut = clock();

for (i = 0; i < 10000000; i++) {

pgcd\_recur(a,b);

}

cout << "Récursif : " << pgcd\_recur(a, b) << endl;

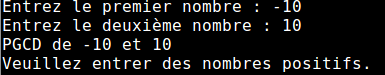
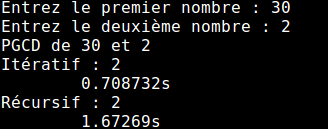
cout << "\t" << float(clock() - temps\_debut) / CLOCKS\_PER\_SEC << "s" << endl;

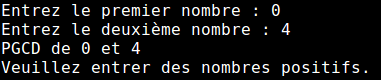
return 0;

}

## Jeux d'essai

|  |  |  |
| --- | --- | --- |
| **Cas testé** | **Entrée** | **Résultat** |
| a et b > 0 | a=30 et b=2 | 2 |
| a < 0 et b > 0 | a=-10 et b=10 | Erreur |
| a = 0 et b > 0 | a=0 et b=4 | Erreur |





## Remarques

Encore une fois, on remarque que la version itérative est plus rapide que la version récursive. Cette fois-ci, la version récursive n'a pas d'amélioration, donc elle est plus lente dans tous les cas.