Université Paris 8 - Vincennes à Saint-Denis

Licence informatique & vidéoludisme

Projet Final Algorithmique Avancée

En mathématiques, les nombres de Catalan C(n) sont une suite de nombres entiers utilisés dans divers problèmes en combinatoire. Ils permettent notamment de dénombrer le nombre d'arbres binaires possédant exactement n nœud internes, ou encore le nombre de manières de parenthéser correctement un mot de longueur n avec autant de parenthèses ouvrantes que fermantes, et une parenthèse ouvrante arrive toujours avant la parenthèse qui la referme.

Bastien Capiaux

Date de rendu: le 06/12/2023

Groupe : L3-A

Contents

1 Énoncé **2**

2 A quoi servent les fichiers ? **2**
2.1 Makefile et readme 2
2.1.1 La compilation 2
2.1.2 L'exécution 2
2.2 main.c 2
2.3 Fichiers - sources 2
2.3.1 catalan.c 2
2.3.2 parenthese.c 2
2.4 Fichiers - headers 3
2.4.1 catalan.h 3
2.4.2 parenthese.h 3

3 Contenus des fichiers **3**
3.1 Catalan 3
3.1.1 Header 3
3.1.2 Source 4
3.2 Parenthèse 8
3.2.1 Header 8
3.2.2 Source 9
3.3 Main 11

4 Test **12**
4.1 Catalan 12
4.2 Parenthèse 13

5 Conclusion **13**
5.1 Catalan 13
5.2 Parenthèse 13

1 Énoncé

Implémenter plusieurs algorithmes permettant de calculer les nombres C(n), et les comparer en temps d'exécution. Écrire un programme qui affiche toutes les manières de bien parenthéser un mot de longueur n, discuter de son efficacité.

2 A quoi servent les fichiers ?

2.1 Makefile et readme

Le Makefile permet d'exécuter des commandes simples afin de nous faciliter la vie. Grâce à "make clean" nous pouvons nettoyer le projet des fichiers temporaire et binaire. Grâce à "make dist" le fichier est directement compressé.
Dans le make file vous pourrez retrouver une ligne de commande pour nettoyer votre terminal, compiler le projet et l'exécuter sur la même ligne.

2.1.1 La compilation

```
$ make
```

2.1.2 L'exécution

```
$ ./prog
```

2.2 main.c

main.c est le fichier qui contient le main et qui va utiliser nos autres fichiers compris dans "sources" et "headers".

2.3 Fichiers - sources

2.3.1 catalan.c

catalan.c regroupe toutes les fonctions que le nombre de Catalan a besoin pour fonctionner.

2.3.2 parenthese.c

parenthese.c regroupe toutes les fonctions pour que l'on puisse parenthésé un mot de longueur n.

2.4 Fichiers - headers

2.4.1 catalan.h

Permet d'utiliser toutes les fonctions de "catalan.c" dans un autre fichiers tel qu'ici notre "main.c".

2.4.2 parenthese.h

Permet d'utiliser toutes les fonctions de "parenthese.c" dans un autre fichiers tel qu'ici notre "main.c".

3 Contenus des fichiers

3.1 Catalan

3.1.1 Header

```
struct Catalan_Func
{
    unsigned long long int (*recursive)(int);
    unsigned long long int (*iterative)(int);
    unsigned long long int (*factoriel)(int);
    unsigned long long int (*generative)(int);
    unsigned long long int (*combinatoire_interpreter)(int);
    unsigned long long int (*hypergeometric)(int);
    unsigned long long int (*infini)(int);
    unsigned long long int (*integral)(int);
    void (*tests)(unsigned long long int (*)(int));
};
extern const struct Catalan_Func Catalan;
```

Structure permettant de créer un objet Catalan permettant de lancer les fonctions qui lui sont attribuées.
"extern const struct Catalan_Func Catalan;" Sert a initialisé la structure ailleurs que dans le fichier header.

```c
unsigned long long int recursive(int N);
unsigned long long int iterative(int N);
unsigned long long int factoriel(int N);
unsigned long long int factoriel_calcule(int N);
unsigned long long int combinatoire_interpreter(int N);
unsigned long long int generative(int N);
unsigned long long int coef_bin(int n, int k);
unsigned long long int hypergeometric(int n);
unsigned long long int infini(int n);
unsigned long long int integral(int n);

float c_execution_time(unsigned long long int (*function_p)(int), int);
void c_execution_time_test(unsigned long long int (*function_p)(int));
void c_tests(unsigned long long int (*function_p)(int));
char *c_function_name(void *function_p);
```

Répertoire des fonctions présente dans "catalan.c".

3.1.2 Source

```c
const struct Catalan_Func Catalan = {
    .recursive = recursive,
    .iterative = iterative,
    .generative = generative,
    .factoriel = factoriel,
    .combinatoire_interpreter = combinatoire_interpreter,
    .hypergeometric = hypergeometric,
    .infini = infini,
    .integral = integral,
    .tests = c_tests
};
```

Initialisation de "extern const struct Catalan_Func Catalan;"

```c
// recursion
unsigned long long int recursive(int N) {
    if (N <= 1)
        return 1;
    unsigned long long int somme = 0;
    int index;
    for (index = 0; index < N; index += 1)
        somme += recursive(index) * recursive(N - index - 1);
    return somme;
}
unsigned long long int iterative(int n) {
    unsigned long long int *nombresCatalan = malloc((sizeof *nombresCatalan) * (n + 1));
    unsigned long long int resultat;
    int indexcourant, indexPP;
    nombresCatalan[0] = nombresCatalan[1] = 1;
    for (indexcourant = 2; indexcourant <= n; indexcourant += 1) {
        nombresCatalan[indexcourant] = 0;
        for (indexPP = 0; indexPP < indexcourant; indexPP += 1) {
            nombresCatalan[indexcourant] += nombresCatalan[indexPP] * nombresCatalan
            [indexcourant - indexPP - 1];
        }
    }
    resultat = nombresCatalan[n];
    free(nombresCatalan);
    return resultat;
}
```

Voici 2 implémentations de la méthode de récursion une naïvement "recursive" et l'autre "iterative" pour voir s'il y a une grosse différence de calcul.

```c
// Explicite
unsigned long long int factoriel(int N) {
    if (N < 0) return 0;
    if (N == 0) return 1;
    return (unsigned long long int)(factoriel_calcule(2 * N) / ((double)factoriel_calcule(N +
    1) * factoriel_calcule(N)));
}
unsigned long long int factoriel_calcule(int N) {
    if (N <= 1) return 1;
    else return (unsigned long long int)N * factoriel_calcule(N - 1);
}
unsigned long long int combinatoire_interpreter(int N) {
    unsigned long long int result = 1;
    int i;
    for (i = 0; i < N; ++i) {
        result *= (2 * N - i);
        result /= (i + 1);
    }
    return result / (N + 1);
}
```

Voici 2 implémentations de la méthode explicite une se nommant "factoriel" et

l'autre "combinatoire_interpreter".

```cpp
// Generative
unsigned long long int generative(int N) {
    unsigned long long int result = 1;
    int i;
    for (i = 1; i <= N; ++i) {
        result *= (1 - sqrt(1 - 4.0 / (i * i)));
    }
    return result / 2.0;
}
```

Ceci est l'implémentation de la méthode générative que j'ai nommée "generative".

```cpp
// Coef Binomial utile pour la suite
unsigned long long int coef_bin(int n, int k) {
    if (k == 0 || k == n) {
        return 1;
    } else {
        return coef_bin(n - 1, k - 1) * n / k;
    }
}
// Expression intégrale
unsigned long long int integral(int n) {
    float result = 1.0;
    for (int k = 1; k <= n; ++k) {
        result *= (2.0 * k - 1) / (n + k);
    }
    return (unsigned long long int) result;
}

// Hypergeometric
unsigned long long int hypergeometric(int n) {
    return coef_bin(2 * n, n) / (n + 1);
}

// Série infinie
unsigned long long int infini(int n) {
    unsigned long long int result = 0;
    int k;
    for (k = 0; k <= n; k++) {
        result += coef_bin(n, k) * coef_bin(n, k) / (k + 1);
    }
    return result;
}
```

Voilà l'implémentation des méthodes intégrale, hypergéométrique et séries infinies se nommant respectivement "integral", "hypergeometric" et "infini"

```c
float c_execution_time(unsigned long long int (*function_p)(int), int N) {
    long start = clock();
    function_p(N);
    return (float)(clock() - start) / CLOCKS_PER_SEC;
}
void c_execution_time_test(unsigned long long int (*function_p)(int)) {
    float execut_time;
    float overall_execution_time = 0.f;
    float arret_time = 0.5f;
    float arret_time_s = 60.0f;
    float index;
    float max_index = 100000;
    float percent;
    for (index = 0; index <= max_index; index += 1) {
        execut_time = c_execution_time(function_p, index);
        if (execut_time > arret_time) {

            printf("index_execution_time ( \033[1;31m%0.f\033[0m ) : \033[33m%f\033[39ms > \033[33m%.2f\033[39ms -> arret du
            test.\n", index-1, execut_time, arret_time);
            percent = ((index-1)/max_index)*100;
            printf("%s | 0 -> \033[1;31m%0.f\033[0m ( \033[1;31m%.3f%%\033[0m ) | overall_execution_time: \033[33m%f\033[39ms.
            \n",c_function_name(function_p), index-1, percent, overall_execution_time);
            return;
        }
        overall_execution_time += execut_time;
        if (overall_execution_time >= arret_time_s)
        {
            printf("overall_execution_time ( \033[1;31m%0.f\033[0m ): \033[33m%f\033[39ms > \033[33m%.0f\033[39ms -> arret du
            test.\n", index-1, overall_execution_time, arret_time_s);
            percent = ((index-1)/max_index)*100;
            printf("%s | 0 -> \033[1;31m%0.f\033[0m ( \033[1;31m%.3f%%\033[0m )  | overall_execution_time: \033[33m%f\033[39ms.
            \n", c_function_name(function_p), index-1, percent, overall_execution_time);
            return;
        }
    }
    percent = ((index-1)/max_index)*100;
    printf("%s | 0 -> \033[1;32m%0.f\033[0m ( \033[1;32m%.3f%%\033[0m )  | overall_execution_time: \033[33m%f\033[39ms.\n",
    c_function_name(function_p), index-1, percent, overall_execution_time);
}

void c_tests(unsigned long long int (*function_p)(int)) {
    printf("\nExecution time test %s:\n", c_function_name(function_p));
    c_execution_time_test(function_p);
}
```

Série de fonctions de test pour Catalan :

- c_test : Permet de lancer les tests.
- c_execution_time : donne le temps d'exécution d'une fonction.
- c_execution_time_test : permet de faire le test plusieurs fois sur une même fonction avec différentes valeurs et renvoie le temps total d'exécutions.
- c_fonction_name : renvoie le nom de la fonction demander.

3.2 Parenthèse

3.2.1 Header

```c
struct Parenthese_Func {
    void (*recursive)(int);
    // void (*iterative)(int);
    void (*tests)(void (*)(int));
};
extern const struct Parenthese_Func Parenthese;
```

Structure permettant de créer un objet Parenthese permettant de lancer les fonctions qui lui sont attribuées.
"extern const struct Parenthese_Func Parenthese;" Sert a initialisé la structure ailleurs que dans le fichier header.

```c
// struct Parenthese_struct;
// typedef struct Parenthese_struct{
//     char *str;
//     int left;
//     int right;
// }Parenthese_struct;

// extern Parenthese_struct *queue;


void parenthese_rec(int);
void generateParenthesesRec(char [], int, int, int );
// void generateParenthesesIte(int);

float p_execution_time(void (*function_p)(int), int N);
void p_execution_time_test(void (*function_p)(int));
void p_tests(void (*function_p)(int));
char *p_function_name(void *function_p);
```

Répertoire des fonctions présente dans "parenthese.c".

3.2.2 Source

```c
const struct Parenthese_Func Parenthese = {
    .recursive = parenthese_rec,
    // .iterative = generateParenthesesIte,
    .tests = p_tests
};
```

Définition de Parenthese pour appeler nos fonctions dans le "main.c".

```c
void generateParenthesesRec(char current[], int left, int right, const int N) {
    if (left + right == 2 * N) {
        printf("%s\n", current);
        return;
    }
    if (left < N) {
        current[left + right] = '(';
        generateParenthesesRec(current, left + 1, right, N);
    }
    if (right < left) {
        current[left + right] = ')';
        generateParenthesesRec(current, left, right + 1, N);
    }
}
void parenthese_rec(const int N) {
    char current[2 * N + 1];  // +1 for the null terminator
    current[2 * N] = '\0';    // null terminator
    generateParenthesesRec(current, 0, 0, N);
}
```

Permet d'afficher le bon parenthèsage d'un mot de longueur n avec C(n) print. (Se qui est très long a afficher).

```c
float p_execution_time(void(*function_p)(int), int N) {
    long start = clock();
    function_p(N);
    return (float)(clock() - start) / CLOCKS_PER_SEC;
}
void p_execution_time_test(void(*function_p)(int)) {
    float execut_time;
    float overall_execution_time = 0.f;
    float arret_time = 2.0f;
    unsigned long long int index;
    for (index = 1; index <= 50; index += 1)
    {
        execut_time = p_execution_time(function_p, index);
        printf("%s ( %llu ) : \033[33m%f\033[39ms\n\n", p_function_name(function_p), index, execut_time);
        if (execut_time > arret_time)
        {
            printf("( %llu ) : \033[33m%f\033[39m supérieur à \033[33m%f\033[39ms -> arret du test.\n", index,
            execut_time, arret_time);
            printf("%s | overall_execution_time: \033[33m%f\033[39ms.\n",p_function_name(function_p),
            overall_execution_time);
            return;
        }
        overall_execution_time += execut_time;
    }
    printf("%s | overall_execution_time: \033[33m%f\033[39ms.\n",p_function_name(function_p),
    overall_execution_time);
}

void p_tests(void(*function_p)(int)) {
    printf("\nExecution time test:\n");
    p_execution_time_test(function_p);
}

char *p_function_name(void *function_p) {
    if (function_p == parenthese_rec)
        return "Parenthese -> recursive";
    // else if (function_p == generateParenthesesIte)
    //  return "Parenthese -> iterative";
    else
        return "unknown_function";
}
```

Série de fonctions de test pour Parenthese:

- p_test : Permet de lancer les tests.
- p_execution_time : donne le temps d'exécution d'une fonction.
- p_execution_time_test : permet de faire le test plusieurs fois sur une même fonction avec différentes valeurs et renvoie le temps total d'exécutions.
- p_fonction_name : renvoie le nom de la fonction demander.

3.3 Main

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

#include "headers/catalan.h"
#include "headers/parenthese.h"

int main() {
    Parenthese.tests(Parenthese.recursive);
    printf("\n");
    Catalan.tests(Catalan.recursive);
    printf("\n");
    Catalan.tests(Catalan.iterative);
    printf("\n");
    Catalan.tests(Catalan.factoriel);
    printf("\n");
    Catalan.tests(Catalan.combinatoire_interpreter);
    printf("\n");
    Catalan.tests(Catalan.generative);
    printf("\n");
    Catalan.tests(Catalan.hypergeometric);
    printf("\n");
    Catalan.tests(Catalan.infini);
    printf("\n");
    Catalan.tests(Catalan.integral);
    return 0;
}
```

Le fichier "main.c" Prends le const struct de Catalan et Parenthese et applique la fonction de test sur ces fonctions.

4 Test

4.1 Catalan

Ceci est le résultat d'un test de 0 à 100 000 éléments sur nos fonctions de calcul de nos nombres de Catalan.

Voici donc les résultats classés par ordre croissants du plus bas vers le meilleur temps :

1. récursive
2. infini
3. itérative
4. factoriel
5. hypergéométrique
6. combinatoire interpreter
7. intégral
8. générative

4.2 Parenthèse

Ceci est le résultat d'un test de 0 à 50 éléments sur notre fonction récursive nous permettant d'afficher toutes les manières de bien parenthésé un mot.

5 Conclusion

5.1 Catalan

Nous avons pu comparer de nombreuses fonctions et/ou méthodes de calculs et nous avons pu voir que la méthode générative est l'une voir la meilleure fonction pour trouver les nombres de catalan. Nous pouvons remarquer que les 2 fonctions qui ont les meilleurs résultats n'utilisent pas le coefficient binomial et la récursion ce qui doit montrer que ma fonction binomiale soit n'est pas assez rapide ou trop mal optimisé pour suivre la cadence.
J'aurais voulu essayer d'autre manière d'écrire le coefficient binomial pour comparer les résultats.

5.2 Parenthèse

Nous pouvons voire que notre fonction basique est très lente mais fonctionnelle. Le problème de cette fonction est que nous devons afficher toutes les façons de parenthésés ce qui demande énormément de calcule.