NTYAM Kévin - EL OUECHRINE Mohamed Amine.

# Rapport MOCA - Semaine 10.

## Profiling de Pascal:

```
                     Call graph (explanation follows)


granularity: each sample hit covers 4 byte(s) for 0.35% of 2.83 seconds

index % time    self  children    called     name
                                                 <spontaneous>
[1]    100.0    0.02    2.81                 main [1]
                0.08    2.73 7085610/7085610     comb [2]
-----------------------------------------------
                0.08    2.73 7085610/7085610     main [1]
[2]     99.3    0.08    2.73 7085610         comb [2]
                2.73    0.00 21256830/21256830     fact [3]
-----------------------------------------------
                2.73    0.00 21256830/21256830     comb [2]
[3]     96.5    2.73    0.00 21256830         fact [3]
-----------------------------------------------
```

Le profiling révèle un goulot d'étranglement majeur dans la fonction fact , qui consomme 98,4% du temps d'exécution avec 41 millions d'appels depuis comb. Cela indique une mauvaise approche algorithmique

## Ameliorations effectuee:

```
double num(int p, int n) {
    double result = 1.0;
    for(int i = 0; i < p; i++) {
        result *= (n - i);
    }
    return result;
}

double comb(int p, int n) {
    if(p == 0 || p == n) return 1;
    if(p == 1) return n;

    return num(p, n) / fact(p);
}
```

1. Ajout d'une nouvelle fonction num qui calcule le numérateur n(n-1)...(n-p+1) de manière itérative sans calculer toutes les factorielles.
2. Ajout des cas particulier pour ces ca (p=0, p=n, p=1).

## Après Améliorations:

```
granularity: each sample hit covers 4 byte(s) for 0.62% of 1.61 seconds

index % time    self  children    called     name
                                                 <spontaneous>
[1]    100.0    0.04    1.57                 main [1]
                0.07    1.50 7085610/7085610     comb [2]
-----------------------------------------------
                0.07    1.50 7085610/7085610     main [1]
[2]     97.5    0.07    1.50 7085610         comb [2]
                0.82    0.00 6699200/6699200     num [3]
                0.68    0.00 6699200/6699200     fact [4]
-----------------------------------------------
                0.82    0.00 6699200/6699200     comb [2]
[3]     50.9    0.82    0.00 6699200         num [3]
-----------------------------------------------
                0.68    0.00 6699200/6699200     comb [2]
[4]     41.9    0.68    0.00 6699200         fact [4]
-----------------------------------------------
```

Le profiling montre que le nombre d'appels à fact a été divisé par 3, passant de 21,2 millions à 6,7 millions. Cette réduction est cohérente avec les modifications apportées :

**Avant** : La fonction comb utilisait 3 appels à fact par combinaison (n!, k!, (n-k)!).

**Après** : L'optimisation via num(p, n) calcule directement n·(n-1)·...·(n-p+1), ne nécessitant plus qu'un seul appel à fact(p) (pour le dénominateur p!).

Cette simplification explique la division par 3 des appels, confirmant l'efficacité de l'approche itérative.

## Exercice 2:

On va appliquer les notions apprise sur cet outil sur notre projet:

Après avoir lancé gprof sur notre ./analyseur a eu ca :

```
granularity: each sample hit covers 4 byte(s) for 0.05% of 18.75 seconds

index % time    self  children    called     name
                                                 <spontaneous>
[1]    100.0    0.00   18.75                 main [1]
               18.68    0.00  221715/221715      incWord [2]
                0.00    0.02  221760/221760      insertDico [6]
                0.01    0.00      46/138         displayWord [3]
                0.00    0.01       1/1           displayDico [7]
                0.00    0.01       1/2           displayNodes [4]
                0.01    0.00  221761/221761      next_word [8]
```

Résultats Principaux:

**Temps d'exécution global**

```
granularity: each sample hit covers 4 byte(s) for 0.05% of 18.75 seconds
```

## Analyse des résultats:

La fonction incword du fichier word.c c'est révèle **très** coûteuse qui a été appelé **221 715 fois**, monopolise presque tout le temps CPU.

### Analyse de la fonction incword:

**Défauts clés** :

1. **Fuites mémoire potentielles**: On fait un double malloc ou 1 de ces 2 est inutile .
2. **Performances dégradées** :complexité O(n) par insertion et O(n*n) tout a la fin de l'insertion , parce qu'au début, c'est négligeable d'insérer à la queue, mais ça peut poser des problèmes avec de longs textes.

### Améliorations

J'ai modifié l'en-tête de la fonction incword pour qu'elle renvoie systématiquement la queue (dernier élément) de la liste. En effet, la queue est mise à jour à chaque itération, contrairement à la tête qui restait fixe.

Cette correction résout le problème de la version précédente, où la tête et la queue pointaient accidentellement vers le même maillon, causant des incohérences dans les modifications.

## Résultats

Le nb d'appel des fonction est reste le meme c'est qui logique parceque c'etait pas le but de changer la logique de notre algorithme.

On est passee de 18s pour executer un grand txt a quelques ms a l'execution

```
granularity: each sample hit covers 4 byte(s) for 33.33% of 0.03 seconds

index % time    self  children    called     name
                0.01    0.00      46/138         main [2]
                0.02    0.00      92/138         displayNodes [3]
[1]    100.0    0.03    0.00     138         displayWord [1]
```