UNIVERSIDAD DE CONCEPCIÓN
FACULTAD DE INGENIERÍA
DEPARTAMENTO DE INFORMÁTICA

Tarea 3: Optimización

Integrantes:

- Francy Pilar Jélvez Jen.
  2021452656.
- Pedro Ignacio Palacios Rossi.
  2021429140.
- Diego Joaquín Andrés Venegas Anabalón.
  2021433473.

Docente: Javier Vidal Valenzuela

Fecha de Entrega: Viernes, 8 de Diciembre 2023.

1. Introducción

El propósito de esta tarea es desarrollar y poner en práctica nuestros conocimientos sobre la optimización de programas con métodos vistos en clase, tales como técnicas de loop unrolling, eliminación de referencias a memoria, optimización acumulativa, etc.

Vamos a analizar la optimización de estos programas haciendo testing de tiempos de ejecución, un método comúnmente usado para el análisis de algoritmos.

Las funciones a optimizar son de bajo nivel de complejidad, para poder analizar mejor los tiempos y ver como un código más complejo puede ser más óptimo que uno simple. Las funciones requeridas por esta tarea son combine(), una función que combina los punteros en un array con sus datas, fibonacci(), que calcula el valor de el nivel de fibonacci deseado, e invert(), que invierte los carácteres dentro de un string.

2. Desarrollo

2.1. Combine:

2.1.1 Combine1:

La versión sin optimizar tiene complejidad óptima O(n), sin embargo, cuenta con un amplio margen de optimización, ya que tiene varios problemas, como uso innecesario de funciones y de-referencias, además de que no tiene buenas cualidades para permitir paralelismo.

```
void combine1(vec_ptr v, data_t *dest) {
   int i;
   *dest = IDENT;
   for (i = 0; i < vec_length(v); i++) {
       data_t val;
       get_vec_element(v, i, &val);
       *dest = *dest OP val;
   }
}
```

2.1.2 Combine2:

Primero removemos el llamado a la función vec_length() de las iteraciones del ciclo for, ya que el procesador tiene que buscar en memoria las secciones de código pertenecientes a cada función por cada llamado a la misma.

```
    void combine2(vec_ptr v, data_t *dest){
        long i;
        long length = vec_length(v);
        *dest = IDENT;
        for(i = 0; i < length ; i++){
            data_t val;
            get_vec_element(v,i,&val);
            *dest = *dest OP val;
        }
    }
```

2.1.3 Combine3:

Luego removemos el llamado a la función get_vec_element() y lo reemplazamos con acceso directo al arreglo de la estructura, también eliminamos la declaración de la variable val.

```
void combine3(vec_ptr v, data_t *dest) {
    int i;
    long int length = vec_length(v);
    data_t *data = get_vec_start(v);
    *dest = 0;
    for (i = 0 ; i < length ; i++) {
        *dest = *dest OP data[i];
    }
}
```

2.1.4 Combine4:

En esta implementación se elimina el acceso a memoria generado por el uso del puntero *dest.

```
void combine4(vec_ptr v, data_t *dest) {
    long int i, length = vec_length(v);
    data_t *data = get_vec_start(v), acc = IDENT;
    for (i = 0; i < length; i++){
        acc = acc OP data[i];
    }
    *dest = acc;
}
```

2.1.5 Combine5:

En esta optimización realizamos un loop unrolling de cuatro operaciones por ciclo, de modo que por iteración realizamos el trabajo que deberían hacer cuatro de las anteriores.

```
void combine5(vec_ptr v, data_t *dest) {
  long int i, length = vec_length(v);
  data_t *data = get_vec_start(v), acc = IDENT;
  for (i = 0 ; i < length ; i += 4){
     acc = acc OP ((data[i] OP data[i+1]) OP (data[i+2] OP data[i+3]));
  }
  for ( ; i < length ; i++){
     acc = acc OP data[i];
  }
  *dest = acc;
}
```

2.2 Comparación entre combine1() y combine5()

En combine_test.c, se pueden medir los tiempos de ejecución de combine1() con combine5() según sus ciclos por segundo. El ejemplo siguiente es por 1.000.000 de elementos en vec_ptr;

```
Ingrese cantidad de elementos del arreglo:1000000

Ejecucion de combine1 desde 0 hasta 2735
Ciclos transcurridos:2741
Ejecucion de combine5 desde 2748 hasta 2749
Ciclos transcurridos:1
```

Como se puede ver, es mucho más eficiente combine5() comparado con combine1(), con 2740 ciclos menos que el anterior.

2.3 Combine6:

La versión anterior le permite hacer múltiples operaciones por ciclo al procesador, sin embargo, no permite paralelismo absoluto ya que algunas de las operaciones dependen de otras, en esta versión tenemos 4 valores independientes con los que vamos operando de manera paralela en cada ciclo.

```
void combine6(vec_ptr v, data_t *dest) {
    int i;
    long int length = vec_length(v);
    data_t *data = get_vec_start(v);
    data_t resultado1 = IDENT;
    data_t resultado2 = IDENT;
    data_t resultado3 = IDENT;
    data_t resultado4 = IDENT;
    for (i = 0 ; i < length ; i += 4) {
        resultado1 = resultado1 OP data[i];
        resultado2 = resultado2 OP data[i+1];
        resultado3 = resultado3 OP data[i+2];
        resultado4 = resultado4 OP data[i+3];
    }
    for ( ; i < length ; i++) {
        resultado1 = resultado1 OP data[i];
    }
    *dest = resultado1 OP resultado2 OP resultado3 OP resultado4;
}
```

2.2. Fibonacci

2.2.1 No optimizado

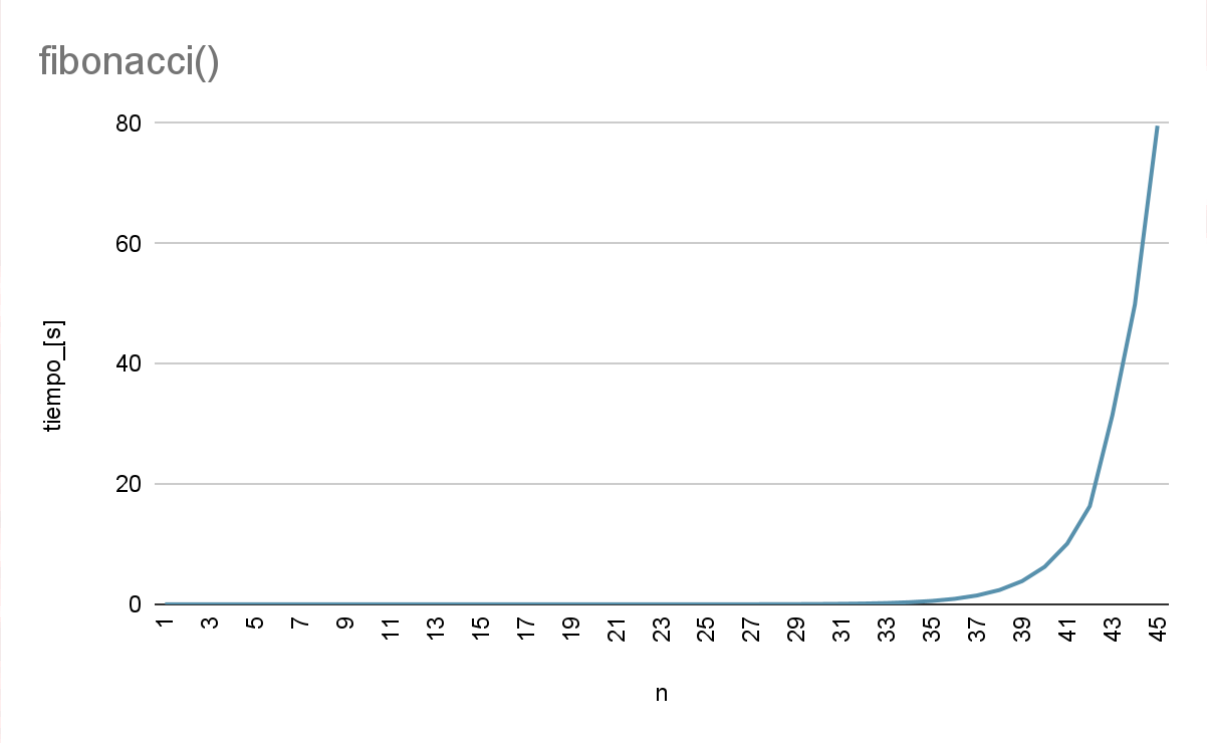
Primero analizaremos el código sin optimizar del enunciado que retorna la n-ésima posición de la secuencia de fibonacci. Veremos que es una función recursiva con múltiples problemas graves de optimización.

```
long long int fibonacci(long long int *n) {
    if (*n == 1) return 0;
    if (*n == 2) return 1;
    long long int m = *n - 1;
    long long int p = *n - 2;
    return fibonacci(&m) + fibonacci(&p);
}
```

Para empezar, la recursividad causa que tengamos múltiples instancias de la función con sus valores propios para “n”, ”m” y “p”, además, muchas de estas instancias van a estar duplicadas, por lo que son redundantes. También vemos que la función trabaja con una referencia al valor “n” lo que es innecesario, ya que solo se utiliza el valor almacenado en “n” sin modificarlo, y es más tardío, ya que cada vez que usamos el valor, tenemos que acceder a memoria. Por último, notamos que cada llamado a fibonacci(n) con *n > 2, se ejecutan 2 llamados más, lo que lleva a una cantidad exponencial de llamados y lleva la complejidad de la función a O(2^n).

2.2.1.1 Experimentación:

Hicimos una experimentación para medir el tiempo promedio de ejecución de la función fibonacci respecto la posición de la secuencia que se pide, para cada valor se promedia un mínimo de 20 experimentos. Estos son los resultados:



Vemos que se correlaciona con la hipótesis de complejidad exponencial.

2.2.2 Optimizado

Mejoramos la implementación anterior aplicando las optimizaciones pedidas, como resultado tenemos la siguiente función:

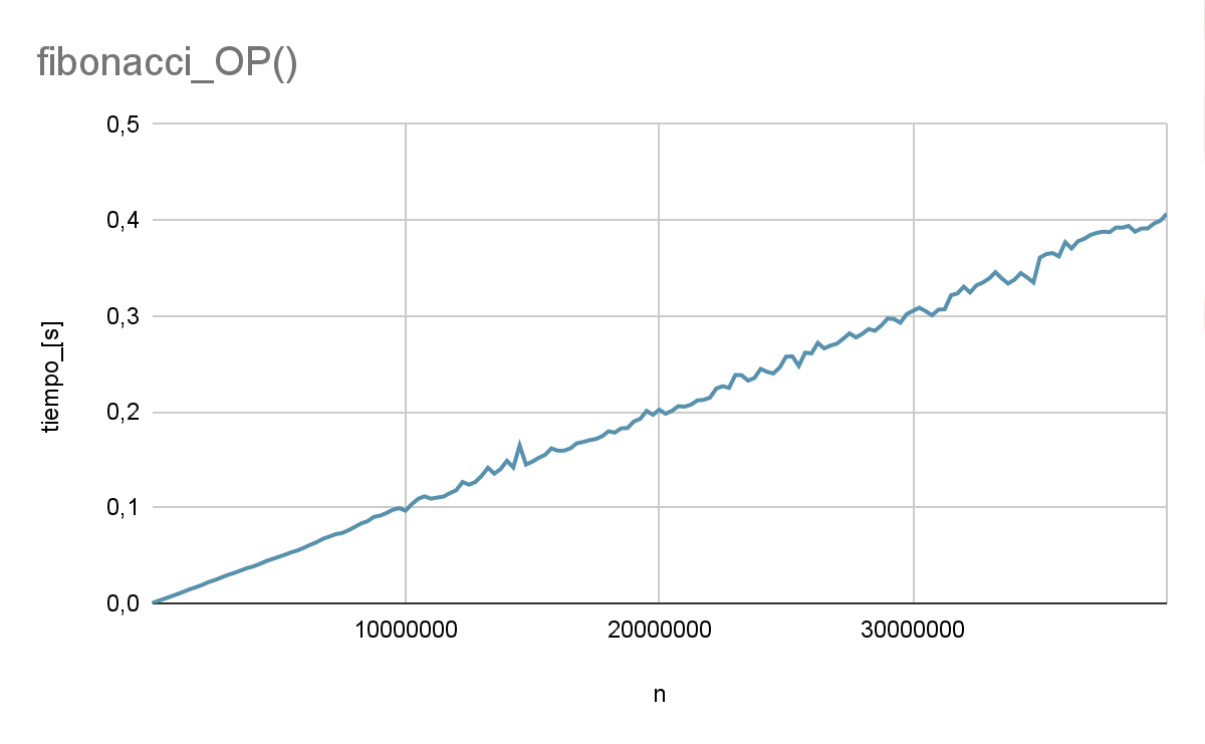
```
typedef long long int lli;

lli fibonacci_OP(lli n) {
   lli f0 = 0, f1 = 1, f2 = 1,f3 = 2, f2_cp;
   for(lli i = 4 ; i < n ; i += 4){
       f2_cp = f2;

       f0 = f2 + f3;
       f1 = f2 + f3 * 2;
       f2 = f2 * 2 + f3 * 3;
       f3 = f2_cp * 3 + f3 * 5;
   }

   switch(n%4) {
       case 1:
           return f0;
           break;
       case 2:
           return f1;
           break;
       case 3:
           return f2;
           break;
       case 0:
           return f3;
           break;
   }

   return -1;
}
```

Esta versión iterativa elimina los llamados innecesarios a otras secciones de código, los accesos a memoria y las operaciones innecesarias, también aplicamos loop unrolling con posibilidad de 4 operaciones por iteración del ciclo.

La complejidad de esta función es O(n).

2.2.2.1 Resultados de la experimentación:

Se realizó un experimentación idéntica a la realizada en el fibonacci anterior:



Con esta experimentación se alcanzaron valores mucho mayores sin llegar a los valores de tiempo anteriores.

2.3 Invertir

2.3.1 Invertir sin Optimizaciones:

A continuación, se presenta el código de invertir() que hicimos, con máximo grado de abstracción posible;

```
void invertir(char *str) {
    for (int i = 0 ; i < strlen(str) / 2 ; ++i) {
        char aux = str[i];
        str[i] = str[strlen(str) - i - 1];
        str[strlen(str) - i - 1] = aux;
    }
}
```

Está compuesto con la función *strlen(int size)* de la biblioteca de <string.h> de C que es O(n), con un carácter de aux que ocupa un byte, y tres llamadas a strlen en el loop.

Su funcionamiento es básico; se realiza un swap entre el último y el primer carácter de nuestra cadena de chars, equivalente a complejidad temporal O(n/2). Lo que le da la suficiente abstracción a este proceso, es como llama a strlen demasiadas veces, para sólo ocuparlo como una comparación. También se puede ver que al llamar n/2 veces a la función strlen, el algoritmo pasa a ser O(n^2) (siendo n el largo del string), por lo que hay bastante espacio para mejorarlo.

2.3.2 Optimizaciones:

A continuación, se presentan tres diferentes tipos de optimizaciones realizadas a la función invertir:

2.3.2.1 invertir1:

En esta optimización se hace uso de un nuevo integer, length, que guarda el resultado de la función strlen, reduciendo el total de sus llamadas en el algoritmo original y pasando a ser O(n). Reduciendo considerablemente el CPE.

```
void invertir1(char *str) {
    int length = strlen(str);
    char aux;
    for (int i = 0 ; i < length / 2 ; ++i) {
        aux = str[i];
        str[i] = str[length - i - 1];
        str[length - i - 1] = aux;
    }
}
```

2.3.2.2 invertir2:

En esta optimización se hace uso de la técnica de loop unrolling, específicamente se hacen dos intercambios por iteración.

```
void invertir2(char *str) {
    int length = strlen(str);
    char aux;
    char aux2;
    int i;
    for (i = 0 ; i < (length / 2) - 1 ; i += 2) {
        aux = str[i];
        aux2 = str[i+1];
        str[i] = str[length - i - 1];
        str[i+1] = str[length - i - 2];
        str[length - i - 1] = aux;
        str[length - i - 2] = aux2;
    }
    if (i < (length / 2)) {
        aux = str[i];
        str[i] = str[length - i - 1];
        str[length - i - 1] = aux;
    }
}
```

2.3.2.3 invertir3:

Por último, en esta optimización reducimos la cantidad de operaciones del loop unrolling anteriormente realizado.

```
void invertir3(char *str) {
    int length = strlen(str);
    int halflength = (length / 2) - 1;
    char aux, aux2;
    int i, j;
    for (i = 0, j = length-1 ; i < halflength ; i += 2, j -= 2) {
        aux = str[i];
        aux2 = str[i+1];
        str[i] = str[j];
        str[i+1] = str[j-1];
        str[j] = aux;
        str[j-1] = aux2;
    }
    if (i < halflength + 1) {
        aux = str[i];
        str[i] = str[j];
        str[j] = aux;
    }
}
```

2.3.3 Experimentación:

A continuación se encuentran los valores (En clocks per second) de los tiempos de ejecución de cada invertir(). Como se puede ver, invertir() con máxima abstracción se demora considerablemente más que todos los demás.

Tamaño del Array	invertir()	invertir1()	invertir2()	invertir3()
10000	0.011000	0	0	0
20000	0.44400	0.000100	0.000100	0
30000	0.101900	0.000100	0	0
40000	0.175000	0	0.000100	0
50000	0.277400	0.000200	0.000100	0.000100
60000	0.400400	0.000100	0	0
70000	0.546300	0	0	0.000200
80000	0.721300	0.000100	0	0.000200
90000	1.040100	0.000200	0	0
100000	1.320500	0.000400	0.000200	0

3. Conclusión

Esta actividad fue una buena instancia para poner en práctica los conocimientos adquiridos sobre optimización, en la que nos encontramos con múltiples problemas y pudimos encontrar solución a estos mismos.