CSc 600-01 (Section 1)

Homework 2 - Procedural Programming

prepared by Ilya Kopyl

CSc 600 Homework 2 - Procedural Programming

March 2, 2018

Homework is prepared by: Ilya Kopyl.
It is formatted in LaTeX, using TeXShop editor (under GNU GPL license).

1. Plateau program (max sequence length) (a combinatorial algorithm)

The array a(1..n) contains sorted integers. Write a function maxlen(a,n) that returns the length of the longest sequence of identical numbers (for example, if a= 1, 1, 1, 2, 3, 3, 5, 6, 6, 6, 6, 7, 9 then maxlen returns 4 because the longest sequence 6, 6, 6, 6 contains 4 numbers. Write a demo main program for testing the work of maxlen. Explain your solution, and insert comments in your program. The time complexity of the solution should be in O(n).

The answer is listed on the pages 2 through 5.

The code listing of maxlen function:

```
unsigned int maxlen(int *a, unsigned int n)
{
    // handling the edge cases - arrays of size 0 and 1:
    if (n < 2)
        return n;

    unsigned int max_count, current_count, i;
    i = max_count = 0;
    current_count = 1;

    printf("    a[%d]=%d; \tcurrent_count=%d; \tmax_count=%d\n",
           i, a[i], current_count, max_count);

    for (i = 1; i < n; ++i)
    {
        if (a[i] == a[i-1])         // counting the current sequence
        {
            current_count++;

            // checking whether the longest sequence is at the end of array
            if(i == n-1 && current_count > max_count)
                max_count = current_count;
        }
        else                        // starting the count of the new sequence
        {
            // before resetting the counter, save it's value if it is above threshold
            if (current_count > max_count)
                max_count = current_count;

            // exit the loop if max_count is sufficiently large
            if (max_count >= n-i)
                break;

            current_count = 1;
        }

        printf("    a[%d]=%d; \tcurrent_count=%d; \tmax_count=%d\n",
               i, a[i], current_count, max_count);
    }
    return max_count;
}
```

The code listing of main program:

```c
#include <stdio.h>
#include <assert.h>
#include "functions.h"

unsigned int maxlen(int *, unsigned int);

int main()
{
    int result = 0;

    int a[13] = { 1, 1, 1, 2, 3, 3, 5, 6, 6, 6, 6, 7, 9 };
    printf("Array a:    ");
    printIntArray(a, sizeof(a));

    result = maxlen(a, 13);
    printf("Max sequence length of array a = %d\n\n", result);
    assert(result == 4);


    int b[0] = {};            // test case: an empty array
    printf("Array b:    ");
    printIntArray(b, sizeof(b));

    result = maxlen(b, 0);
    printf("Max sequence length of array b = %d\n\n", result);
    assert(result == 0);


    int c[1] = { 12 };        // test case: 1 element in the array
    printf("Array c:    ");
    printIntArray(c, sizeof(c));

    result = maxlen(c, 1);
    printf("Max sequence length of array c = %d\n\n", result);
    assert(result == 1);


    // testing the early loop exit:
    int d[6] = { 16, 16, 16, 18, 18, 20 };
    printf("Array d:    ");
    printIntArray(d, sizeof(d));

    result = maxlen(d, 6);
    printf("Max sequence length of array d = %d\n\n", result);
    assert(result == 3);
```

```
    int e[2] = { 0, 0 };        // test case: 2 elements with the same value
    printf("Array e:    ");
    printIntArray(e, sizeof(e));

    result = maxlen(e, 2);
    printf("Max sequence length of array e = %d\n\n", result);
    assert(result == 2);


    int f[2] = { 0, 1 };        // test case: 2 elements with different values
    printf("Array f: ");
    printIntArray(f, sizeof(f));

    result = maxlen(f, 2);
    printf("Max sequence length of array f = %d\n\n", result);
    assert(result == 1);


    int g[4] = { 1, 2, 3, 3 };  // test case: the longest sequence ends with array
    printf("Array g:    ");
    printIntArray(g, sizeof(g));

    result = maxlen(g, 4);
    printf("Max sequence length of array g = %d\n\n", result);

    return 0;
}
```

Auxiliary functions (in separate file "functions.c"):

```
#include <stdio.h>
#include "functions.h"
/** Passing the size of the array with sizeof() function */
void printIntArray(int *array, unsigned int size)
{
    genericPrintNumArray(array, size, sizeof(int));
}

void genericPrintNumArray(void *object, unsigned int size, unsigned int elem_size)
{
    char *p = (char *) object;
    while (p < (char *) object + size) {
        printf("%d  ", *p);
        p += elem_size;
    }
    printf("\n");
}
```

The result of the program execution:

```
Array a:    1  1  1  2  3  3  5  6  6  6  6  7  9
    a[0]=1;     current_count=1;        max_count=0
    a[1]=1;     current_count=2;        max_count=0
    a[2]=1;     current_count=3;        max_count=0
    a[3]=2;     current_count=1;        max_count=3
    a[4]=3;     current_count=1;        max_count=3
    a[5]=3;     current_count=2;        max_count=3
    a[6]=5;     current_count=1;        max_count=3
    a[7]=6;     current_count=1;        max_count=3
    a[8]=6;     current_count=2;        max_count=3
    a[9]=6;     current_count=3;        max_count=3
    a[10]=6;    current_count=4;        max_count=3
Max sequence length of array a = 4

Array b:
Max sequence length of array b = 0

Array c:    12
Max sequence length of array c = 1

Array d:    16  16  16  18  18  20
    a[0]=16;    current_count=1;        max_count=0
    a[1]=16;    current_count=2;        max_count=0
    a[2]=16;    current_count=3;        max_count=0
Max sequence length of array d = 3

Array e:    0  0
    a[0]=0;     current_count=1;        max_count=0
    a[1]=0;     current_count=2;        max_count=2
Max sequence length of array e = 2

Array f: 0  1
    a[0]=0;     current_count=1;        max_count=0
Max sequence length of array f = 1

Array g:    1  2  3  3
    a[0]=1;     current_count=1;        max_count=0
    a[1]=2;     current_count=1;        max_count=1
    a[2]=3;     current_count=1;        max_count=1
    a[3]=3;     current_count=2;        max_count=2
Max sequence length of array g = 2
```

2. Integer plot function (find a smart way to code big integers)

Write a program BigInt(n) that displays an arbitrary positive integer n using big characters of size 7x7, as in the following example for BigInt(170):

```
  @@      @@@@@@@@   @@@@@@
 @@@             @@  @@    @@
  @@           @@    @@    @@
  @@          @@     @@    @@
  @@         @@      @@    @@
  @@        @@       @@    @@
  @@       @@        @@    @@
@@@@@@   @@          @@@@@@
```

Write a demo main program that illustrates the work of BigInt(n) and prints the following sequence of big numbers 1, 12, 123, 1234,..., 1234567890, one below the other.

The answer is listed on the pages 7 through 9.

The code listing of the two-dimensional array that stores bit pattern of each BigInt digit. It is declared in the global space (outside of any function).

```
#define NUMBER_OF_ROWS 8

/**
 * Digits are stored as bit patterns of 8-bit unsigned integer (char) numbers.
 *
 * Each digit requires just 8 bytes of storage - which is polynomially smaller
 * than the storage in brute-force approach where each digit is represented by
 * a 2D array of 8x8 characters, with 64 bytes of storage per digit.
 */
const unsigned char BIG_DIGITS[NUMBER_OF_ROWS][10] =
{
    {   // row 0 of all 10 digits
        0b00000000u, 0b00000000u, 0b00000000u, 0b00000000u, 0b00000000u,
        0b00000000u, 0b00000000u, 0b00000000u, 0b00000000u, 0b00000000u
    },
    {   // row 1 of all 10 digits
        0b00111110u, 0b00001100u, 0b00011110u, 0b00011110u, 0b00000110u,
        0b00111111u, 0b00011110u, 0b01111111u, 0b00011110u, 0b00011110u
    },
    {   // row 2 of all 10 digits
        0b01100011u, 0b00011100u, 0b00110011u, 0b00110011u, 0b00001110u,
        0b00110000u, 0b00110011u, 0b00000011u, 0b00110011u, 0b00110011u
    },
    {   // row 3 of all 10 digits
        0b01100011u, 0b00001100u, 0b00110011u, 0b00000011u, 0b00010110u,
        0b00110000u, 0b00110000u, 0b00000110u, 0b00110011u, 0b00110011u
    },
    {   // row 4 of all 10 digits
        0b01100011u, 0b00001100u, 0b00000110u, 0b00001100u, 0b00110110u,
        0b00111110u, 0b00111110u, 0b00001100u, 0b00011110u, 0b00011111u
    },
    {   // row 5 of all 10 digits
        0b01100011u, 0b00001100u, 0b00001100u, 0b00000011u, 0b01100110u,
        0b00000011u, 0b00110011u, 0b00011000u, 0b00110011u, 0b00000011u
    },
    {   // row 6 of all 10 digits
        0b01100011u, 0b00001100u, 0b00011000u, 0b00110011u, 0b01111111u,
        0b00000011u, 0b00110011u, 0b00110000u, 0b00110011u, 0b00110011u
    },
    {   // row 7 of all 10 digits
        0b00111110u, 0b00111111u, 0b00111111u, 0b00011110u, 0b00000110u,
        0b00111110u, 0b00011110u, 0b01100000u, 0b00011110u, 0b00011110u
    }
};
```

Main program, excluding the declaration of BIG_DIGITS array:

```c
#include <stdio.h>
#include <math.h>

void BigInt(unsigned int);
unsigned int getNumberOfDigits(unsigned int);

#define NUMBER_OF_BITS 8
#define NUMBER_OF_ROWS 8

/** BIG_DIGITS[][] is declared here; its declaration is listed on the previous page */

int main()
{
    BigInt(1);
    BigInt(12);
    BigInt(123);
    BigInt(1234);
    BigInt(1234567890);
    return 0;
}
void BigInt(unsigned int n)
{
    unsigned int numOfDigits, c;
    c = numOfDigits = getNumberOfDigits(n);
    int decimals[numOfDigits];

    // decomposing the number into an array of decimal digits
    do {
        decimals[c-1] = n % 10;
        c--;
    } while ((n /= 10));

    // printing all digits at once, row by row
    for (int row = 0; row < NUMBER_OF_ROWS; row++)
    {
        for (int digit = 0; digit < numOfDigits; digit++)
            /* iteratively extracting bit pattern of each char of BIG_DIGITS array
               and printing it, starting from the most significant bit first */
            for (int bit = NUMBER_OF_BITS-1; bit >= 0; bit--)
                printf("%c",
                ((BIG_DIGITS[row][decimals[digit]] >> bit) & 1) == 1 ? '@' : ' ');
        puts("");    // adds a newline character at the end of each printed row
    }
}
unsigned int getNumberOfDigits(unsigned int n)
{
    return (unsigned int) log10(n) + 1;        // floor of log10(n) + 1
}
```

The result of the program execution:

3. Array processing (elimination of three largest values) (one of many array reduction problems)

The array a(1..n) contains arbitrary integers. Write a function reduce(a, n) that reduces the array a(1..n) by eliminating from it all values that are equal to three largest different integers. For example, if a=(9, 1, 1, 6, 7, 1, 2, 3, 3, 5, 6, 6, 6, 7, 9) then three largest different integers are 6, 7, 9, and after reduction the reduced array would be a=(1, 1, 1, 2, 3, 3, 5), n=7. The time complexity of the solution should be in O(n).

The answer is listed on the pages 11 through 13.

The code listing of the entire program for problem #3:

```c
#include <stdio.h>
#include "functions.h"

unsigned int reduce(int *, unsigned int);
void findTop3MaxValuesInArray(int *, unsigned int, int *, int *, int *);
void nullifyTop3MaxValuesInArray(int *, unsigned int, int, int, int);
unsigned int moveZeroesToEndOfArray(int *, unsigned int);

int main()
{
    int a[16] = { 9, 1, 1, 6, 7, 1, 2, 3, 3, 5, 6, 6, 6, 6, 7, 9 };

    puts("Original array: ");
    printIntArray(a, sizeof(a));

    int reducedN = reduce(a, 16);

    puts("Reduced array with original bounds: ");
    printIntArray(a, sizeof(a));

    printf("Reduced array with new bounds (n = %d): \n", reducedN);
    printIntArray(a, reducedN * sizeof(int));

    return 0;
}

unsigned int reduce(int * a, unsigned int n)
{
    int max1, max2, max3;
    max1 = max2 = max3 = 0;

    findTop3MaxValuesInArray(a, 16, &max1, &max2, &max3);
    nullifyTop3MaxValuesInArray(a, 16, max1, max2, max3);

    return moveZeroesToEndOfArray(a, n);
}

/** Setting all occurrences of numbers max1, max2, max3 in the array to zero. */
void nullifyTop3MaxValuesInArray(int * a, unsigned int n, int max1, int max2, int max3)
{
    unsigned int i;
    for (i = 0; i < n; i++)
        if (a[i] == max1 || a[i] == max2 || a[i] == max3)
            a[i] = 0;
}
```

```c
/** Finding the first 3 maximum values of an array. */
void findTop3MaxValuesInArray(int * a, unsigned int n, int *max1, int *max2, int *max3)
{
    *max1 = *max2 = *max3 = 0;
    unsigned int i;
    for (i = 0; i < n; i++)
    {
        if (a[i] > *max1)
        {
            *max3 = *max2;
            *max2 = *max1;
            *max1 = a[i];
        }
        else if (a[i] > *max2 && a[i] < *max1)
        {
            *max3 = *max2;
            *max2 = a[i];
        }
        else if (a[i] > *max3 && a[i] < *max2)
            *max3 = a[i];
    }
}

/** Moves all zeroes to the end of array by copying all non-zero
 * values to the beginning of the array and returning the total
 * amount of non-zero values (i.e. the size of reduced array). */
unsigned int moveZeroesToEndOfArray(int * a, unsigned int n)
{
    unsigned int indexOfNullValue, sizeOfReducedArray;
    indexOfNullValue = 0;

    // copying all non-zero values to the beginning of array
    unsigned int i;
    for (i = 0; i < n; i++)
        if (a[i] != 0)
            a[indexOfNullValue++] = a[i];

    sizeOfReducedArray = indexOfNullValue;

    // nullifying the remaining part of the array
    while (indexOfNullValue < n)
        a[indexOfNullValue++] = 0;

    // returning the size of reduced array
    return sizeOfReducedArray;
}
```

Auxiliary functions (in separate file "functions.c"):

```c
#include <stdio.h>
#include "functions.h"
/** Passing the size of the array with sizeof() function */
void printIntArray(int *array, unsigned int size)
{
    genericPrintNumArray(array, size, sizeof(int));
}

void genericPrintNumArray(void *object, unsigned int size, unsigned int elem_size)
{
    char *p = (char *) object;
    while (p < (char *) object + size) {
        printf("%d  ", *p);
        p += elem_size;
    }
    printf("\n");
}
```

The result of the program execution:

```
$ gcc -Wall -std=c99 hw2-problem3.c functions.c -O3
$ ./a.out
Original array:
9  1  1  6  7  1  2  3  3  5  6  6  6  6  7  9
Reduced array with original bounds:
1  1  1  2  3  3  5  0  0  0  0  0  0  0  0  0
Reduced array with new bounds (n = 7):
1  1  1  2  3  3  5
```

4. Iteration versus recursion (an opportunity for performance measurement)

Make a sorted integer array a[i]=i, i=0,...,n-1. Let bs(a, n, x) be a binary search program that returns the index i of the array a[0..n-1] where a[i]=x. Obviously, the result is bs(a, n, x)=x, and the binary search function can be tested using the loop

```
for (j=0; j < K; j++)
    for (i=0; i < n; i++)
        if (bs(a, n, i) != i)
            cout << "\nERROR";
```

Select the largest n your software can support and then K so that this loop with an iterative version of bs runs 3 seconds or more. Then measure and compare this run time and the run time of the loop that uses a recursive version of bs. Compare these run times using maximum compiler optimization (release version) and the slowest version (minimum optimization or the debug version). If you use a laptop, make measurements using AC power, and then the same measurements using only the battery. What conclusions can you derive from these experiments? Who is faster? Why?

The answer is listed on the pages 15 through 19.

The code listing of the entire program for problem #4:

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

#define K 1000                          // system-dependent constant

void initializeArray(int *, int);
int ibs(int *, int, int);
int rbs(int *, int, int, int);
double ibsTest(int *, int);
double rbsTest(int *, int);

int main()
{
    int sizeOfArray = 65535;         // 2^16-1, staying conservative
    int a[sizeOfArray];

    initializeArray(a, sizeOfArray);

    printf("Running time of iterative Binary Search: %f seconds.\n",
           ibsTest(a, sizeOfArray));
    printf("Running time of recursive Binary Search: %f seconds.\n",
           rbsTest(a, sizeOfArray));

    puts("Benchmarking is complete!");

    return 0;
}

void initializeArray(int *a, int n)
{
    for (int i = 0; i < n; i++)
        a[i] = i;
}
```

```c
/**
 * Iterative implementation of Binary Search
 */
int ibs(int *a, int n, int value)
{
    int itemLocation = -1;
    int low, mid, high;
    low = 0;
    high = n;

    while (high >= low && itemLocation == -1)
    {
        mid = (low + high) / 2;
        if (value == a[mid])
            itemLocation = mid;
        else if (value < a[mid])
            high = mid - 1;
        else
            low = mid + 1;
    }
    return itemLocation;
}

/**
 * Recursive implementation of Binary Search
 */
int rbs(int *a, int low, int high, int value)
{
    if (high < low)
        return -1;                // the value not found
    else
    {
        int mid = low + (high - low) / 2;
        if (a[mid] == value)
            return mid;
        else if (a[mid] > value)
            return rbs(a, low, mid-1, value);
        else
            return rbs(a, mid+1, high, value);
    }
}
```

```
/**
 * Function for testing the performance
 * of iterative binary search.
 */
double ibsTest(int *a, int n)
{
    clock_t start_t, end_t, running_time = 0;
    int i, j;

    start_t = clock();
    for (j = 0; j < K; j++)
        for (i = 0; i < n; i++)
        {
            if (ibs(a, n, i) != i)
                puts("ERROR");
        }
    end_t = clock();
    running_time += (end_t - start_t);

    return (double) running_time/CLOCKS_PER_SEC;
}

/**
 * Function for testing the performance
 * of recursive binary search.
 */
double rbsTest(int *a, int n)
{
    clock_t start_t, end_t, running_time = 0;
    int i, j;

    start_t = clock();
    for (j = 0; j < K; j++)
        for (i = 0; i < n; i++)
        {
            if (rbs(a, 0, n, i) != i)
                puts("ERROR");
        }
    end_t = clock();
    running_time += (end_t - start_t);

    return (double) running_time/CLOCKS_PER_SEC;
}
```

The result of the program execution with different setup & compiler optimizations:

```
# connected to charger, no Wi-Fi, no monitors connected:

$ gcc -Wall -std=c99 hw2-problem4.c -O0
$ ./a.out
Running time of iterative Binary Search: 12.815268 seconds.
Running time of recursive Binary Search: 13.416952 seconds.
Benchmarking is complete!

$ gcc -Wall -std=c99 hw2-problem4.c -O1
$ ./a.out
Running time of iterative Binary Search: 3.122441 seconds.
Running time of recursive Binary Search: 3.351381 seconds.
Benchmarking is complete!

$ gcc -Wall -std=c99 hw2-problem4.c -O2
$ ./a.out
Running time of iterative Binary Search: 2.831430 seconds.
Running time of recursive Binary Search: 2.775777 seconds.
Benchmarking is complete!


$ gcc -Wall -std=c99 hw2-problem4.c -O3
$ ./a.out
Running time of iterative Binary Search: 3.120335 seconds.
Running time of recursive Binary Search: 2.861631 seconds.
Benchmarking is complete!
```

Initially, with minimal compiler optimization, the implementation of recursive Binary Search runs comparatively slower than iterative Binary Search. However, as we increase the intensity of the compiler optimization, we shall see that the recursive Binary Search outperforms its iterative counterpart. Another observation is related to the program performance while a laptop is powered by only a battery: as you can see, it runs a bit faster than while plugged in.

```
$ gcc -Wall -std=c99 hw2-problem4.c -O0
$ ./a.out
Running time of iterative Binary Search: 12.409296 seconds.
Running time of recursive Binary Search: 13.290153 seconds.
Benchmarking is complete!

$ gcc -Wall -std=c99 hw2-problem4.c -O3
$ ./a.out
Running time of iterative Binary Search: 3.029139 seconds.
Running time of recursive Binary Search: 2.747411 seconds.
Benchmarking is complete!
```

One possible explanation to that counterintuitive phenomena could be related to behavior defined in operating system. When it activates a power savings mode, it is very likely that certain background processes halt, thus making my program run faster by tens of milliseconds.

5. Iteration versus recursion (another opportunity for performance measurement)

Write a recursive function Frec(n) that computes Fibonacci numbers. Then write an iterative version of Fibonacci number function Fit(n). Functions Frec(n) and Fit(n) return the same value but with different performance.

Write the main program that discovers the value N10 so that Frec(N10) runs on your machine exactly 10 seconds. Then measure the run time of Fit(N10) and compute how many times is Fit(N10) faster than Frec(N10). Show what is N10 on your machine.

Notes:

1. When you measure the speed, your machine should be disconnected from the Internet, it should use the AC power supply, and it should run only one program (your performance measurement program).

2. In C++ you can measure current time in seconds using the following function:

```
double sec(void)
{
    return double(clock()) / double(CLOCKS_PER_SEC);
}
```

To measure the run time of fast programs you must repeat them many times inside a loop. Take care to eliminate the overhead generated by the loop.

The answer is listed on the pages 20 through 22.

The code listing of the entire program for problem #5:

```c
#include <stdio.h>
#include <zconf.h>
#include <time.h>

int Frec(int);
int Fit(int);

double findN(int *);

// using the function pointer to avoid unnecessary code repetition:
double benchmarkFibFunction(int (*f)(int), int);


int main()
{
    puts("Depending on the compiler optimizations, results may vary. Please wait...");

    int threshold = 10;
    // !!! the value of threshold will become an index of Nth Fibonachi term
    double runningTimeFrec = findN(&threshold);
    printf("N10 = %d\n", threshold);
    printf("Running time of Frec(%d) is %f seconds.\n",
           threshold, runningTimeFrec);

    double runningTimeFit = benchmarkFibFunction(Fit, threshold);
    printf("Running time of Fit(%d) is %f seconds.\n",
           threshold, runningTimeFit);

    double speedupFactor = runningTimeFrec / runningTimeFit;

    printf("Fit(%1$d) is %2$.2f times faster than Frec(%1$d).\n",
           threshold, speedupFactor);

    return 0;
}
```

```
/** Function iteratively runs benchmarks against recursive Fibonacci function
 * until the running time exceeds the given time threshold in seconds.
 * @return index of nth Fibonacci term is returned through parameter (dirty hack).
 * @return the time it took to compute nth Fibonacci term. */
double findN(int *timeThreshold) {
    int n = 0;
    double runningTime = 0;
    while ((runningTime = benchmarkFibFunction(Frec, ++n)) < *timeThreshold);
    *timeThreshold = n;
    return runningTime;
}

/** Function measures the performance of both Fibonacci functions.
 *
 * @param f accepts any function that takes and returns an int value.
 * @param n int value that will be passed as a parameter to f
 * @return returns running time in seconds of function f for input n. */
double benchmarkFibFunction(int (*f)(int), int n)
{
    clock_t start_t, end_t = 0;
    start_t = clock();
    (*f)(n);
    end_t = clock();
    return (double) (end_t - start_t) / CLOCKS_PER_SEC;
}

/** Recursive implementation of function
 * computing Fibonacci numbers. */
int Frec(int n)
{
    return n <= 1 ? n : Frec(n - 1) + Frec(n - 2);
}
/** Iterative implementation of function computing Fibonacci numbers. */
int Fit(int n)
{
    int first, second, temp, i;
    first = 0;
    second = 1;

    if (n <= 1)
        return n;
    else
        for (i = 2; i <= n; i++)
        {
            temp = first + second;
            first = second;
            second = temp;
        }
    return temp;        // the n-th value in Fibonacci sequence
}
```

The result of the program execution with different compiler optimizations:

```
$ gcc -std=c99 -Wall hw2-problem5.c -O0
$ ./a.out && ./a.out
Depending on the compiler optimizations, the results may vary. Please wait...
N10 = 46
Running time of Frec(46) is 10.042850 seconds.
Running time of Fit(46) is 0.000000 seconds.
Fit(46) is inf times faster than Frec(46).
Depending on the compiler optimizations, the results may vary. Please wait...
N10 = 46
Running time of Frec(46) is 12.380931 seconds.
Running time of Fit(46) is 0.000002 seconds.
Fit(46) is 6190465.50 times faster than Frec(46).

$ gcc -std=c99 -Wall hw2-problem5.c -O1
$ ./a.out && ./a.out
Depending on the compiler optimizations, the results may vary. Please wait...
N10 = 47
Running time of Frec(47) is 15.694785 seconds.
Running time of Fit(47) is 0.000001 seconds.
Fit(47) is 15694785.00 times faster than Frec(47).
Depending on the compiler optimizations, the results may vary. Please wait...
N10 = 47
Running time of Frec(47) is 15.601271 seconds.
Running time of Fit(47) is 0.000001 seconds.
Fit(47) is 15601271.00 times faster than Frec(47).
```

As you can see, a slightly more aggressive optimization enables Frec function to find 46-th term of Fibonacci sequence under 10 seconds, thus making a number 47 - index of 47th element of Fibonacci sequence - a new return value. Also worth noting that sometimes Fit function might produce a result in such a small amount of time that we don't even have enough precision in double precision floating point numbers to store such data, which would result in underflow. In such cases we can observe artifacts like in the first result, line 3: "Fit(46) is inf times faster than Frec(46)".