CSc 600-01 (Section 1)

Homework 4 - Functional Programming

prepared by Ilya Kopyl

CSc 600 Homework 4 - Scheme and Functional Programming

April 22, 2018

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

1. The concept of first class objects is fundamental for Scheme programming. In particular, in Scheme language any function is a first class object. The main properties of a function as a first class object are exemplified by answering the following questions:

a) The first class object may be expressed as an anonymous literal value (constant). Show an example of the anonymous function and its use.

b) The first class object may be stored in variables (i.e. it may have a symbolic name). Show examples of defining and using named functions.

c) The first class object may be stored in data structures. Show an example of a data structure (e.g. a list) that contains functions.

d) The first class object may be comparable to other objects for equality. Show an example of comparing functions and lists for equality.

e) The first class object may be passed as parameter to procedures/functions. Show an example of passing function as an argument to another function.

f) The first class object may be returned as result from procedures/functions. Show an example of returning a function as a result of another function.

g) The first class object may be readable and printable. Show examples of:

- reading function(s) from keyboard,
- reading function(s) from a file,
- displaying a function.

The answer is listed on the pages TBD through TBD.

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:

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

unsigned int maxlen(int *, unsigned int);
```

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

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

The result of the program execution:

```
Array a:    1  1  1  2  3  3  5  6  6  6  6  7  9
```

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.
 */
```

Main program, excluding the declaration of BIG_DIGITS array:

```
#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
```

The result of the program execution:

```
       Blah-blah-blah
asdasd 12 123 12 3123

123123
```

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);
```

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

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

The result of the program execution:

```
$ gcc -Wall -std=c99 hw2-problem3.c functions.c -O3
$ ./a.out
```

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:

```
#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);
```

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.
```

Initially, with minimal compiler optimization, the implementation of recursive Binary Search

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

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:

```
#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);
```

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...
```

As you can see, a slightly more aggressive optimization enables Frec function to find