

The University of Texas at San Antonio
Computer Science Program
San Antonio, Texas 78249

CS 1723, Data Structures
Spring Semester, 1997
Programming Assignment 1, Due Jan. 22, 1997
The 3/2 Problem

The Assignment: There is a famous game one can play with integers: start with any positive integer n , and carry out the following replacement step:

if n is odd **then** replace n by $(3n + 1)/2$
else if n is even **then** replace n by $n/2$.

This step is repeated over and over again. For example, if we start with $n = 17$, the game produces the following sequence of integers, which I call a *run*, of *length* 10:

17, 26, 13, 20, 10, 5, 8, 4, 2, 1

In this case and in every other case that individuals have tried, the sequence ends in 1. (Notice that 1 is replaced by 2 and then by itself, so nothing more happens after one of these sequences gets to 1.) Researchers conjecture that the sequence converges to 1 for any starting positive integer n , but no one has been able to prove this.

For this assignment, you are to write a program to investigate this game. More specifically, you should

- * Produce the runs for starting integers n from 1 to 100 000 (but don't print the results!). Your program should use `scanf` to read in the limiting value of 100 000 from `stdin`. You should assume that all runs will end in 1, and in fact you can write the program so that it would go into an infinite loop if a 1 was never produced in a run.
- * For these 100 000 runs, keep track of the number of times each *run length* occurs, and keep track of the *maximum value* that occurs in all runs of a given length. (It turns out that the longest run has length 221, for $n \leq 100\,000$.)
- * Use `printf` to print the results on `stdout`. You should print three numbers to a line: the run length, the number of times that run length occurred, and the maximum value that came up in all runs of that length. Do not print a line if there were no runs of that length.

Your C Program: Organize the program as a single file, with (of course)

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

at the top.

Your program should use the following four functions, in addition to `main()`, and the list of prototypes below should appear early in your C source file:

```
void run(int, int *, int *);
int next_value(int);
void stats(int, int);
void print_stats(void);
```

- ★ Here `run(n, &length, &maxi)` produces the run for the integer `n` and returns the run length in the `length` parameter and the maximum value of the run in the `maxi` parameter.
- ★ The function `next_value` should be used by `run` to generate the successor to a given integer, using the rule at the beginning of this writeup.
- ★ A call to `stats(length, maxi)` will enter the given run length and maximum value into the table used by this program.
- ★ Finally, a call to `print_stats()` will print out the results, as stored in the table.

The table itself should be global (defined before `main()` or the other function definitions) and should use the following C data structures:

```
#define MAXS 230 /* runs < 230, n < 100000 */
struct {
    int counts; /* num times run occurred */
    int max_val; /* max val, runs of this length */
} s[MAXS]; /* table to hold run data */
```

Alternatively, you could use the following equivalent declaration of the table. (But don't mix up the two of them. This is more in the style of the Kruse book, but C programmers tend to prefer the previous style.)

```
#define MAXS 230 /* runs < 230, n < 100000 */
typedef struct {
    int counts; /* num times run occurred */
    int max_val; /* max val, runs of this length */
} Entry_type;
Entry_type s[MAXS]; /* table to hold run data */
```

For a value of i (= run length) between 1 and 230, the two fields in this table can be accessed by: `s[i].counts`, and `s[i].max_val`.

The make Utility: Create a new directory `assign1` for your program, using `mkdir`. Name your source file `th.c`. Create a file named `makefile` with the following contents:

```
# makefile for the 3/2 program
th:      th.c
         cc -p -g -o th th.c
lint:    th.c
         lint -m -u th.c
```

Be careful, the third and fifth lines of this file must start with a *single tab character*. After creating your source file (using the editor `vi`), first type

```
runner$ make lint
```

to check out the program. Then type

```
runner$ make
```

to compile the program. The `make` utility will invoke the C compiler (`cc`) with options `-p` to produce profiling information, `-g` to include a run-time symbol table for debugging with the `dbx` utility, and `-o th` to name the executable file `th`.

Execute this program with

```
runner$ th
100000
```

Required Output: Your output should look roughly as follows, but with the extra lines included and your own name printed. My program that produced this output assumed that runs terminated when a 1 was reached, and that initial value $n = 1$ produced a run of length 1. I included the final 1 in the length of all runs. If you make different assumptions, you'll get slightly different output below, but that doesn't matter.

```
Three/halves program written by <your name here>.
Limit value for n: 100000
Run length:  1, Count:      1, Maximum value:      1
Run length:  2, Count:      1, Maximum value:      2
Run length:  3, Count:      1, Maximum value:      4
... (Many lines omitted)
Run length: 211, Count:      3, Maximum value: 296639576
Run length: 212, Count:      1, Maximum value: 296639576
Run length: 214, Count:      1, Maximum value:  53179010
Run length: 215, Count:      1, Maximum value:  53179010
Run length: 222, Count:      1, Maximum value: 10966508
```

Execution Profile: The execution of this program also produces information regarding the function calls—information stored in the file `mon.out`. Then the `prof` utility will use this information to produce an *execution profile*, showing the number of times each function was called and the CPU time consumed by each function. Use the command:

```
runner$ prof th
```

The `dbx` Utility—Location of Segmentation Faults: Finally, you should try to use the debugger in case of a *segmentation fault* (= address out of range). Add the following line to your `print_stats` function:

```
s[1000000].counts = 1;
```

This statement tries to access a memory location far beyond your own permitted portion of memory. It will produce a segmentation fault and a core dump. The example below shows a run, with the fault error message and the use of `dbx` to locate the fault. (Here **boldface** represents characters supplied by the system.)

```
runner$ th > /dev/null
100000
Memory fault(coredump)
runner$ dbx th
Reading symbolic information for th
corefile read successfully
Reading symbolic information for ...
program terminated by signal SEGV ...
(dbx) where
print_stats(), line 62 in "th.c"
main(), line 33 in "th.c"
(dbx) quit
runner$ rm core
```

At the “where”, the debugger will tell you the line number of the fault’s location (line 62 in this case), and the line number from which the function containing the fault was called (line 33), and so on if there is a deeper nesting of calls.

In the first command above, “`th > /dev/null`” will execute `th`, but will throw away (into the *bit bucket*) the standard output. The error output will still go to the terminal.

What To Turn In: You should turn in a *single* listing containing your *source file* (`th.c` in this case), your output, and the profile information, all concatenated together. (You do not need to turn in anything related to your experimentation with the segmentation fault.) One way to produce such a listing is the following:

```
runner$ make
runner$ th >th.output
100000
runner$ prof th >th.profile
runner$ cat th.c th.output th.profile >th.print
runner$ lp -d SCF1 th.print
```