OPERATING SYSTEMS EXERCISE 2

Assignment – Fork Fourier Transform

Implement the Cooley-Tukey Fast Fourier Transform[1] algorithm.

```
SYNOPSIS
    forkFFT
```

Instructions

The input of the program are floating point values, which should be read from *stdin*. Subsequent values are separated by a newline character. The sequence ends when an EOF (End Of File) is encountered.

Your program must accept any number of values. Terminate the program with exit status `EXIT_FAILURE` if an invalid input is encountered.

The program computes the Fourier Transform of its input values recursively, i.e. by calling itself:

1. If the array consists of only 1 number, write that number to *stdout* and exit with exit status `EXIT_SUCCESS`.
2. Otherwise the array consists of $n > 1$ numbers. Split it into two parts with $n/2$ numbers each, such that one part contains all the numbers at even indices and the second part contains all the numbers at odd indices:
$$P_E = \{A[0],\ A[2],\ A[4],\ \dots\}$$
$$P_O = \{A[1],\ A[3],\ A[5],\ \dots\}$$
Terminate the program with exit status `EXIT_FAILURE` if the length of the array is not even.
3. Using *fork(2)* and *execlp(3)*, recursively execute this program in two child processes, one for each of the two parts. Use two unnamed pipes per child to redirect *stdin* and *stdout* (see *pipe(2)* and *dup2(3)*). Write P_E to *stdin* of one child and P_O to *stdin* of the other child. Read the respective results from each child's *stdout*. The two child processes must run simultaneously!
4. Use *wait(2)* or *waitpid(2)* to read the exit status of the children. Terminate the program with exit status `EXIT_FAILURE` if the exit status of any of the two child processes is not `EXIT_SUCCESS`.
5. Let R_E be the result of the even part P_E and R_O be the result of the odd part P_O. The result R of the Fourier Transform of the entire array can now be computed by applying the "butterfly" operation to the two results:
$$\forall k \mid 0 \leq k < n/2 \ : \quad R[k] = R_E[k] + e^{-\frac{2\pi i}{n} \cdot k} \cdot R_O[k] \quad \text{and} \quad R[k+n/2] = R_E[k] - e^{-\frac{2\pi i}{n} \cdot k} \cdot R_O[k]$$
which is identical to:
$$\forall k \mid 0 \leq k < n/2 \ : \quad \begin{aligned} R[k] &= R_E[k] + \left(\cos(-\tfrac{2\pi}{n} \cdot k) + i \cdot \sin(-\tfrac{2\pi}{n} \cdot k)\right) \cdot R_O[k] \\ R[k+n/2] &= R_E[k] - \left(\cos(-\tfrac{2\pi}{n} \cdot k) + i \cdot \sin(-\tfrac{2\pi}{n} \cdot k)\right) \cdot R_O[k] \end{aligned}$$
You may approximate the value of π with 3.141592654.

[1] `https://en.wikipedia.org/wiki/Cooley-Tukey_FFT_algorithm`

6. The resulting array is printed to *stdout*, one value per line. Since the resulting numbers are complex, the program prints two values per line: the real and the imaginary part of each value, separated by a whitespace (likewise R_E and R_O are imaginary, thus when reading them from the child processes, two values per line must be parsed). You may append `*i` or similar to the second value to indicate that it is the imaginary part (see examples). Terminate the program with exit status `EXIT_SUCCESS`.

Throughout the program, it is sufficient to use single precision floating point numbers.

Hints

1. Use *strtof(3)* to parse floating point values. The `endptr` argument can be helpful for reading two numbers per line when reading from the child processes.
2. Use the header *math.h* for the calculation of the sine and cosine. Add `-lm` to the linker options to use these functions.
3. When dealing with complex numbers, you may find following property useful to implement multiplication: $(a + i \cdot b)(c + i \cdot d) = a \cdot c - b \cdot d + i \cdot (a \cdot d + b \cdot c)$ Addition should be straight-forward to implement and no other operation with complex numbers is required.
4. In order to avoid endless recursion[2], fork only if the input number is greater than 1.
5. To output error messages and debug messages, always use *stderr* because *stdout* is redirected in most cases.

Examples

Constant signal:

```
$ cat constant.txt
1.0
1.0
$ ./forkFFT < constant.txt
2.0 0.0*i
0.0 0.0*i
```

Sinus wave:

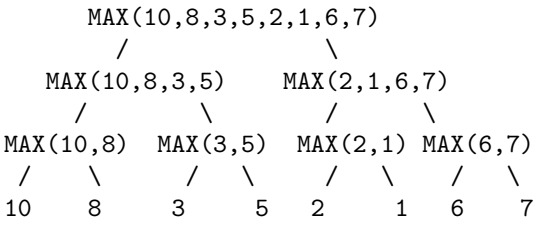
```
$ cat sine.txt
0.000000
0.707107
1.000000
0.707107
0.000000
-0.707107
-1.000000
-0.707107
$ ./forkFFT < sine.txt
0.000000 0.000000*i
0.000000 -4.000001*i
0.000000 0.000000*i
0.000000 -0.000001*i
0.000000 0.000000*i
-0.000000 0.000001*i
0.000000 0.000000*i
-0.000000 4.000001*i
```

[2] `http://en.wikipedia.org/wiki/Fork_bomb`

Bonus exercise, 5 points

Print the parent and child relations in form of a tree to stdout. Print all children which are forked from the parent. The tree should be readable at least to a depth of three. For every node, the intermediate result should be printed.
Depending on how often you call fork the wider the tree becomes. A simple tree example which searches for the maximum number in a set is shown below.

```
10
```



Instructions on how to print the tree

- leaf node:
  A leaf node should print the substep executed by it to stdout with a terminating newline.
- inner node:
  - To get the necessary identation use several blank characters. Think about a good way to find the right number of blank characters. For example you could use precalculated values or calculate the number from the first line you read from the children.
  - Calculate the intermediate result and print this and the executed operation to stdout.
  - Slash and backslash, which represent the branches of the tree, are printed to stdout.
  - Read the output from the children line by line via a pipe. This means read the first line from the first child, then the first line from the second child and so on. Remove the newline characters. Line up the results and then print it with a terminating newline to stdout. Do this for each line returned by the child.

Coding Rules and Guidelines

Your score depends upon the compliance of your submission to the presented guidelines and rules. Violations result in deductions of points. Hence, before submitting your solution, go through the following list and check if your program complies.

Rules

Compliance with these rules is essential to get any points for your submission. A violation of any of the following rules results in 0 points for your submission.

1. All source files of your program(s) must compile via

   ```
   $ gcc -std=c99 -pedantic -Wall -D_DEFAULT_SOURCE -D_BSD_SOURCE -D_SVID_SOURCE
                                  -D_POSIX_C_SOURCE=200809L -g -c filename.c
   ```

   without *errors* and your program(s) must link without *errors*. The compilation flags must be used in the Makefile. The feature test macros must not be bypassed (i.e., by undefining these macros or adding some in the C source code).

2. The functionality of the program(s) must conform to the assignment. The program(s) shall operate according to the specification/assignment given the test cases in the respective assignment.

General Guidelines

Violation of following guidelines leads to a deduction of points.

1. All source files of your program(s) must compile with

   ```
   $ gcc -std=c99 -pedantic -Wall -D_DEFAULT_SOURCE -D_BSD_SOURCE -D_SVID_SOURCE
                                  -D_POSIX_C_SOURCE=200809L -g -c filename.c
   ```

   without *warnings and info messages* and your program(s) must link without warnings.

2. There must be a Makefile implementing the targets: `all` to build the program(s) (i.e. generate executables) from the sources (this must be the first target in the Makefile); `clean` to delete all files that can be built from your sources with the Makefile.

3. The program shall operate according to the specification/assignment without major issues (e.g., segmentation fault, memory corruption).

4. Arguments have to be parsed according to UNIX conventions (we strongly encourage the use of `getopt(3)`). The program has to conform to the given synopsis/usage in the assignment. If the synopsis is violated (e.g., unspecified options or too many arguments), the program has to terminate with the usage message containing the program name and the correct calling syntax. Argument handling should also be implemented for programs without arguments.

5. Correct (=normal) termination, including a cleanup of resources.

6. Upon success the program has to terminate with exit code 0, in case of errors with an exit code greater than 0. We recommend to use the macros `EXIT_SUCCESS` and `EXIT_FAILURE` (defined in `stdlib.h`) to enable portability of the program.

7. If a function indicates an error with its return value, it *should* be checked in general. If the subsequent code depends on the successful execution of a function (e.g. resource allocation), then the return value *must* be checked.

8. Functions that do not take any parameters have to be declared with `void` in the signature, e.g., int get_random_int(void);.

9. Procedures (i.e., functions that do not return a value) have to be declared as `void`.

10. Error messages shall be written to `stderr` and should contain the program name `argv[0]`.

11. It is forbidden to use the functions: `gets`, `scanf`, `fscanf`, `atoi` and `atol` to avoid crashes due to invalid inputs.

FORBIDDEN	USE INSTEAD
gets	fgets
scanf	fgets, sscanf
fscanf	fgets, sscanf
atoi	strtol
atol	strtol

12. Documenation is mandatory. Format the documentation in Doxygen style (see Wiki and Doxygen's intro).

13. Write meaningful comments. For example, meaningful comments describe the algorithm, or why a particular solution has been chosen, if there seems to be an easier solution at a first glance. Avoid comments that just repeat the code itself
(e.g., i = i + 1; /* i is incremented by one */).

14. The documentation of a module must include: name of the module, name and student id of the author (`@author` tag), purpose of the module (`@brief`, `@details` tags) and creation date of the module (`@date` tag).

    Also the Makefile has to include a header, with author and program name at least.

15. Each function shall be documented either before the declaration or the implementation. It should include purpose (`@brief`, `@details` tags), description of parameters and return value (`@param`, `@return` tags) and description of global variables the function uses (`@details` tag).

    You should also document `static` functions (see `EXTRACT_STATIC` in the file `Doxyfile`). Document visible/exported functions in the header file and local (`static`) functions in the C file. Document variables, constants and types (especially `structs`) too.

16. Documentation, names of variables and constants shall be in English.

17. Internal functions shall be marked with the `static` qualifier and are not allowed to be exported (e.g., in a header file). Only functions that are used by other modules shall be declared in the header file.

18. All exercises shall be solved with functions of the C standard library. If a required function is not available in the standard library, you can use other (external) functions too. Avoid reinventing the wheel (e.g., re-implementation of `strcmp`).

19. Name of constants shall be written in upper case, names of variables in lower case (maybe with fist letter capital).

20. Use meaningful variable and constant names (e.g., also semaphores and shared memories).

21. Avoid using global variables as far as possible.

22. All boundaries shall be defined as constants (macros). Avoid arbitrary boundaries. If boundaries are necessary, treat its crossing.

23. Avoid side effects with `&&` and `||`, e.g., write if (b != 0) c = a/b; instead of if (b != 0 && c = a/b).

24. Each `switch` block must contain a `default` case. If the case is not reachable, write `assert(0)` to this case (defensive programming).

25. Logical values shall be treated with logical operators, numerical values with arithmetic operators (e.g., test 2 strings for equality by strcmp (...) == 0 instead of !strcmp (...)).

26. Indent your source code consistently (there are tools for that purpose, e.g., `indent`).

27. Avoid tricky arithmetic statements. Programs are written once, but read more times. Your program is not better if it is shorter!

28. For all I/O operations (read/write from/to `stdin`, `stdout`, files, sockets, pipes, etc.) use *either* standard I/O functions (`fdopen(3)`, `fopen(3)`, `fgets(3)`, etc.) *or* POSIX functions (`open(2)`, `read(2)`, `write(2)`, etc.). Remember, standard I/O functions are buffered. Mixing standard I/O functions and POSIX functions to access a common file descriptor can lead to undefined behaviour and is therefore forbidden.

29. If asked in the assignment, you must implement signal handling (`SIGINT`, `SIGTERM`). You must only use *async-signal-safe* functions in your signal handlers.

30. Close files, free dynamically allocated memory, and remove resources after usage.

31. Don't waste resources due to inconvenient programming. Header files shall not include implementation parts (exception: macros).

Exercise 2 Guidelines

Violation of following guidelines leads to a deduction of points in exercise 2.

1. Correct use of fork/exec/pipes as tought in the lectures. For example, do not exploit inherited memory areas.

2. Ensure termination of child processes without `kill(2)` or `killpg(2)`. Collect the exit codes of child processes (`wait(2)`, `waitpid(2)`, `wait3(2)`).