Grading Policy

This handout explains the grading policy for projects.

1 Point distribution

Points for the project are distributed between the beta and the final software releases with an emphasis on testing in the beta and on performance and code quality in the final.

Beta	45%	*Final*	55%
Performance	20%	Performance	25%
Correctness	15%	-	-
Test coverage	10%	-	-
-	-	Code quality	25%
-	-	Writeup	5%

2 Performance

The fastest solution to each problem receives full credit for performance on the problem. For every other solution, the staff uses the following formulas to compute the performance portion of the grade, where M is the maximum speedup achieved, and S is the speedup of a given solution:

$$G_{beta} = \frac{\lg(S_{beta})}{\lg(M_{beta})}$$

Roughly speaking, if you achieve no speedup, i.e. $S = 1$, then you get zero credit. If $S = M$, you receive full credit. If the fastest solution has a $1000\times$ speedup, then for every factor of 2 you are slower than the fastest solution, you lose about 10%.

For the final stage, you have the opportunity to match or even exceed the best speedup. However, you will not earn more points for exceeding the best beta speedup, though you will earn bragging rights. Formally, the final grading formula will be:

$$G_{final} = \frac{\lg(MIN(S_{final}, M_{beta}))}{\lg(M_{beta})}$$

The exact formula we use might be revised if the grade distribution turns out to be unreasonable, but you should still expect that your performance grade will be relative to the fastest *beta* speedup, and somehow compensates for when some students achieve a gigantic speedup relative to everyone else.

3 Test format

As part of your project, you will be adding test cases to `tests.c` following the pattern of the test cases that we have provided you. To grade your code for test coverage and implementation correctness, we will build and run all pairwise combinations of `problem.c` and `tests.c`, and count the test failures.

The tests you write *must* adhere to the existing output format of the tests, or we will not be able to parse the output. In particular, we are looking for "`--> test_name: FAIL`" on `stderr`, which indicates a test failure. Of course, if your test suite causes an implementation to crash, we'll interpret that as a test failure. We have provided `TEST_PASS` and `TEST_FAIL` macros that print out these messages in the correct format.

You are allowed to print whatever else you like to aid in your debugging efforts – the grading scripts will ignore everything except `stderr`.

Note that although we will grade your test suite as a single pass-or-fail unit, we expect that your tests are modular and follow good software engineering principles. You are encouraged to use the framework provided for having one test per function.

4 Coverage

For each solution that fails any test in the aggregate test suite, we will distribute 10 points evenly amongst all of the groups that catch the bug (ie, they have tests that fail with the buggy code). Your coverage grade will be the number of points accumulated over the maximum number of points accumulated by any group. In case all of the solutions pass all of the tests, we will provide some buggy solutions so that your test suite has some bugs to catch. Finally, your coverage grade will be **severely penalized** if your test suite fails on the reference implementation we supply – there is no excuse for failing to run your test suite against the program we supply!

5 Correctness

Correctness grading is more subjective than coverage to allow us to evaluate the severity of bugs. Generally, grades should fall into the following categories. If all tests pass, you will get full marks. If your implementation is essentially correct, but misses some corner cases, you can expect between 80% to 90% of the points. If your implementation fails more tests but the performance tests still run, you can expect 60% to 80%. If you cannot run the performance tests, then you will get a very low correctness grade and potentially a zero for performance.

6 Code quality

Although code quality is subjective, good programmers produce programs that are neatly formatted, contain descriptive variables and function names, are partitioned well into modular units, are well commented, and contain liberal use of assertions via the `assert.h` package. For example, every significant loop and recursive function should have an invariant (whether self-explanatory or documented in a comment) that can be verified with an assertion. You will find that it is easy to write a program that is "bigger than your head," where you return to the code even just a few days — sometimes hours — later and find it hard to figure out what you yourself were doing without investing a significant amount in time. Comments and assertions can greatly improve your ability to work on your program over a long period of time.

The course staff expects high-quality code from you and will grade you down for lack of quality. You will find that although the code we give you is often slow code, it is fairly straightforward to understand. We expect you to write code that is at least as readable, while running much faster.

7 Write-up

Along with your code, you must also submit a writeup, which should guide the staff through the changes you made to the code. The write-up should provide answers to the questions asked in the project, as well as a good description of what you implemented.

We also want you to describe all the ideas you tried out — even ones that did not work out in the end. Try to think about why the changes you made produced the results they did, especially when those results are surprising to you. This kind of experimentation will build your intuition about performance programming.