Project 1: Bit Hacks

Last Updated: September 15, 2021

This project provides you with an opportunity to learn how to improve the performance of programs using the `perf` *tool and to experiment with* ***word parallelism****: the abstraction of a computer word as a vector of bits on which bitwise arithmetic and logical operations can be performed.*

Word parallelism — more colloquially called ***bit hacks*** *— can have a dramatic impact on performance. This project will also give you the opportunity to develop and practice your C programming skills.*

Generally, when you are concerned about the performance of a program, the best approach is often to implement something correctly and then evaluate it. In some cases, many parts of this initial implementation (or even the entire thing) may be adequate for your needs. If you find that you need to improve performance, however, you must decide where to focus your efforts, which is where profiling becomes useful. Profiling can help you identify the performance bottlenecks in your program. In this project will you will work toward significant speedup over the initial implementation!

Note: *It is a good idea to reread the Course Information handout before you get started!*

Contents

1 Due dates

- ☐ *Team contract:* 11:59 P.M. on Friday, September 17, 2021
- ☐ *Beta release:* 11:59 P.M. on Wednesday, September 22, 2021
- ☐ *Beta write-up:* 11:59 P.M. on Thursday, September 23, 2021
- ☐ *Final release:* 11:59 P.M. on Wednesday, September 29, 2021
- ☐ *Final write-up:* 11:59 P.M. on Thursday, September 30, 2021

SPEED
LIMIT
∞
PER ORDER OF 6.172
(a)
(b)

Figure 1: (a) The original 192-by-192 monochrome image. **(b)** The image after rotation.

2 Getting started

On your second week of work as an intern at Snailspeed Ltd., your boss asks you build a faster version of a program sold by Snailspeed's competitor, Diddled Bits, Inc. Snailspeed thinks that there is great demand for a faster binary image rotator and would like you to develop a faster product to compete with that of Diddled Bits. A binary image, more precisely a 1-bit monochrome image, is an image with a palette of only two colors — traditionally black and white. An example of a 192-by-192 monochrome bmp image is shown in Figure 1(a), and its rotated image is in Figure 1(b).

The author of Diddled Bits' program, Ben Bitdiddle, is an ex-Snailspeed employee, and Snailspeed has his original rotation program. You suspect that you can exploit word parallelism to greatly speed up this program. Your prototype, as a proof of concept, need only work on square $N \times N$ images where N is a multiple of 64. Currently, the prototype runs *very* slowly. It is your job to analyze its performance and make it faster.

Many programs operate under tight constraints, and getting respectable performance under these circumstances can be especially challenging. Part of Snailspeed Ltd.'s product portfolio targets mobile and embedded devices such as cell phones, where memory is limited. Your program must operate *in place*, meaning that it does not use much storage beyond that used by the image itself.

Since you can't change Snailspeed's culture overnight, write your program in standard C. Specifically, some things are off limits. You may not perform multithreaded parallelization. You may not use any assembly-language instructions or compiler intrinsics for the beta assignment, but you may use them for your final submission.

The x86 machines on which you will be running are ***little endian***: bytes of words are stored in memory with least-significant bytes first. For example, on a machine with 8-byte words, the 8-byte value `0x1A2B3C4D5E6F7A8B` would appear in memory as `8B 7A 6F 5E 4D 3C 2B 1A`. Understanding the difference between big endian and little endian is important when working with binary images since you will be changing the order of their bits to perform rotation. There are many excellent resources available for further reading on endianness such as `https://en.wikipedia.org/wiki/Endianness` and Danny Cohen's classic 1980 paper "On Holy Wars and a Plea for Peace," which introduced the terminology of endianness to computer science.

You need not make your code portable to big-endian machines. Please post on Piazza if you have any questions about the rules.

Team formation

Teams will be formed by random matching. A public list of teams will be posted on Piazza. The course staff can appreciate the desire of students to work with their friends, and for the capstone project, Project 4, you can do just that. But randomization is more equitable because some students may have a harder time than others making connections. Allowing students to form their own groups can reinforce racism, sexism, and other forms of discrimination. Even idealistically-minded MIT students can sometimes be unknowingly biased against their less privileged peers. Given the strife in our society, the impact of exclusionary practices (conscious or unconscious) must be recognized. Randomization is a known way to break down the barriers between the "in crowd" and the "out crowd." The course staff is committed to giving every student in 6.172 a fair chance.

Team contract

You and your teammate must agree to a team contract. A team contract is an agreement among teammates about how your team will operate — a set of conventions that you plan to abide by. Since your team was randomly formed, taking the time to discuss and agree upon a thorough team contract can save you unexpected disappointments.

The questions below provide some considerations for what might go into your team contract. You should also think back to good or bad aspects of team-project experiences you've already had. The questions are broken into three categories: meeting and communication norms, work norms, and decision making. You needn't address all these questions, which are simply suggestions. Focus on the issues that your team considers most important. A skimpy team contract is a bad idea. The TA's will review all contracts. If you have little in your team contract and it turns out that your team has problems working together, you make it harder for the course staff to help you with your plight. All team members should write their names at the end of the contract, to indicate that they agree with it.

Meeting and communication norms

- How often will the team plan to meet outside of class? How long do you anticipate meetings will be? What will you do if things change?
- Where and when will outside-class meetings be held? What will you do if someone fails to show for a meeting?

- How will you communicate outside of meetings? (Email list? Real-time messaging platform? Signed messages on the Bitcoin blockchain?)
- If someone in the group decides to drop the class, what obligations does that person have to inform the remaining team members and complete promised work?

Work norms

- How much time per week do you anticipate it will take to make the project successful?
- How will work be divided among team members? On which parts of the project will you do pair programming?
- What will happen if someone does not follow through on a commitment, e.g., not doing their work? What if someone gets sick?
- How will the work be reviewed? How will you manage your code branches?
- How will you deal with different work habits of individual team members? (E.g., some people like to get work done early, while others like to work under the pressure of a deadline.)

Decision making

- Do you need unanimous consent to make a decision? What process for decision-making will you use if you can't agree?
- How will you prioritize the work to be done? How will you deal with the common situation in which different team members have different optimization ideas?
- What happens if everyone does not agree on the level of commitment, e.g., some team members want an A, but others are willing to settle for a B.
- Is it acceptable for some team members to do more or less work than others?

Submitting your team contract

Each member should individually submit the team contract as a PDF document via Gradescope. In addition, once your team repository has been formed, you should commit a copy of your team contract to the top level of the repo under the name `team-contract.pdf`. Feel free to update the team contract in the repo as needed during the project, as long as all parties agree.

Getting the code

The initial code is available to view at https://github.mit.edu/6172-fall21/project1.

You can view your teammates and `team-name` on Piazza. You can use this command to get your team's project repository:

```
$ git clone git@github.mit.edu:6172-fall21/project1_<team-name>.git project1
```

We strongly recommend that groups practice pair programming, where two partners work together with one person at the keyboard and the other serving as a pair of watchful eyes. After an agreed-upon time (typically 20 or 30 minutes), the partners switch. This style of programming leads to bugs being caught earlier and both programmers retain familiarity with the code. You can experiment with what works best for you and your team.

Make sure to add and commit a copy of your team contract to your project repo for future reference by you, the course staff, and your MITPOSSE Deputies.

3 Code structure

The rotation code resides in the `snailspeed/` directory. Take a look at `rotate.c`. This code implements the most basic binary image rotation algorithm. Helper code resides in the `utils/` directory. Take a look at `utils.c` and `utils.h`. A bunch of useful functions are already predefined for you which you are free to use in your image-rotation code. In the implementation, every pixel in a binary image is represented by single bit. All of the bits of the image are stored in row-major order as a linear array in memory, but they can be easily accessed individually through the public `get_bit()` and `set_bit()` functions.

Your job is to improve the `rotate_bit_matrix()` function programmed by Ben Bitdiddle. Your implementation of `rotate_bit_matrix()` is considered correct if the contents of the rotated binary image, as accessed through `get_bit()`, is the same as after running Ben Bitdiddle's slow implementation.

Code constraints

Do not call `generate_bit_matrix()` from within your implementations. You can allocate small buffers on the stack or in the BSS section (e.g., global arrays), but you should not allocate large amounts of memory with `malloc()`. You may not use any intrinsics in your beta submission: this includes Clang's `__builtin` functions, `immintrin.h`, `byteswap.h`, and inline assembly code. You can, however, use any compiler flags and compiler directives (`#pragma ...`) for both the beta and final.

The timing and testing system resides in `utils/main.c` and `utils/tester.c`, which call your routines. Do not make modifications to any files within the `utils/` directory, as they will be replaced with fresh copies when the staff runs your code. By the same token, do not remove or change the type signatures of any of your functions that `utils/main.c` or `utils/tester.c` calls. In short, the provided files `utils/main.c` and `utils/tests.c` should always compile against your code and execute correctly!

Building the code

You can build the code by typing `make`. To build with debugging symbols, useful when debugging with gdb, type `make DEBUG=1`. The `rotate` binary accepts arguments that allow you to benchmark and test your code.

```
$ ./rotate
usage:
        -t {file|generated|     Select a test type: Required to select test type
            correctness|tiers}
        -f file-name            Input file name: Required for "file" test type
        -o output-file-name     Output file name: Optional for "file" test type
        -N dimension            Generated image dimension: Required for "generated" test type
        -m min-tier             Minimum tier: Optional for "tiers" test type. Default is 0.
        -l linear-tiers         Number of tiers to search linearly:
                                Optional for "tiers" test type.
                                Default is 8.
                                Set to -1 for all tiers to be linearly searched.
        -M max-tier             Maximum tier: Optional for "tiers" test type. Default is 25.
        -h                      This help message
```

For benchmarking your code, you should use `awsrun` as described in the next section. For playing around with your code or checking correctness, you may want to run the binary locally. If the binary that results from the `make` command does not run locally, your computer architecture might be old. Only in that case should you try compiling with `make LOCAL=1` as documented in the Makefile.

Benchmarks

The first matrix that we test, and hence all subsequent matrices, will be large. Thus, the naive version does not complete any tiers.

When evaluating your implementation's performance, you should use the `awsrun` command, as this is what the staff will use to grade you. You should not rely on any performance measures from running your implementation locally, as this varies wildly depending on your hardware and other factors. You can run a benchmark using the command

```
$ make  # equivalent to `make LOCAL=0 DEBUG=0`
$ awsrun ./rotate -t tiers
```

That said, this problem is memory intensive. The `awsrun` machines we make available to students are single-core VM's on the cloud. There are other tenants using other CPU cores and sharing the same physical RAM with the testing machine. We also run several of these `awsrun` machines to reduce your queue waiting time, and thus you might get different results depending on which machine your job gets routed to. *Since perfect timing is not a practical reality, we ask that you initially focus on making major improvements to your code's performance and not worry about variability. We will follow up with additional instructions on reducing variability.* For grading, we rent out a whole machine and take many steps to minimize variability, including tuning CPU settings and running your code several times, keeping the minimum execution time for each benchmark.

The `./rotate -t tiers` option benchmarks your code by testing it on a sequence of matrices whose sizes grow exponentially. You can see the definition of the growth in `utils/tester.c:326` with constants defined in `utils/main.c:31`. As your code becomes sufficiently fast, you will want to specify a higher max tier than the default by using the `-M max-tier` flag. If you know the approximate tier your code is able to reach, you can save some time by starting the search at a later tier by using the `-m min-tier` flag.

In the interest of not overwhelming the performance testing server, this benchmark command performs a linear search for the first few tiers, then binary search over the remaining tiers to find the highest tier you can reach. You can configure how many tiers to search linearly for with the `-l linear-tiers` flag.

You might observe that the relationship between the input size or the tier versus the runtime is not necessarily perfectly monotonic. If you want to check whether your code is affected by this phenomenon, run a linear search using the `-m min-tier -M max-tier -l linear-tiers` flags. For up to 2 failures, the linear search will "blow through" the failure and keep searching as if the failures didn't happen.

For grading both the beta and the final, we will run your code on all tiers. Your grade will depend not only on the highest tier reached, but also on the time taken on all other tiers. Incremental improvements, even if they do not move your team to the next tier, **do** improve your grade.

Testing

In addition to timing your program, we will check the correctness of your code by rotating random N-by-N images for various values of N, where N is a multiple of 64. The number N will not be known to you, and it may be as small as 64. Therefore, you should make sure your code is general and works on many sizes of images. Use the option `./rotate -t generated -N <dimension>` to test this aspect of your code.

Testing is a fundamental component of good software engineering. A ***regression suite*** is a series of tests that includes all the tests that the software previously failed on. You will find that having a good regression suite for your projects speeds your ability to make performance optimizations. When you find a bug in your code, but your code nevertheless passes your regression tests, you add a new test to the suite *before fixing the bug*. That is, you first write and add to the regression suite a new test to check for the presence of the bug. You then verify that your buggy

$$\begin{array}{cccc} a_{00} & a_{01} & a_{02} & a_{03} \\ a_{10} & a_{11} & a_{12} & a_{13} \\ a_{20} & a_{21} & a_{22} & a_{23} \\ a_{30} & a_{31} & a_{32} & a_{33} \end{array}$$

(a) before applying cyclic permutation

$$\begin{array}{cccc} a_{30} & a_{01} & a_{02} & a_{00} \\ a_{10} & a_{11} & a_{12} & a_{13} \\ a_{20} & a_{21} & a_{22} & a_{23} \\ a_{33} & a_{31} & a_{32} & a_{03} \end{array}$$

(b) after applying cyclic permutation

Figure 2: Applying the cyclic permutation $a_{00} \to a_{03} \to a_{33} \to a_{30} \to a_{00}$ to a 4×4 matrix.

code fails the test. Finally, you fix the bug and check that the corrected code passes the newly revised regression suite. It is well documented that most bugs are in fact old bugs that occurred previously in software development.

4 Rotating a binary image

Your task is to come up with a more efficient implementation for `rotate_bit_matrix()`, given the rules stated earlier. Your write-up and documentation should explain clearly and succinctly how your method works.

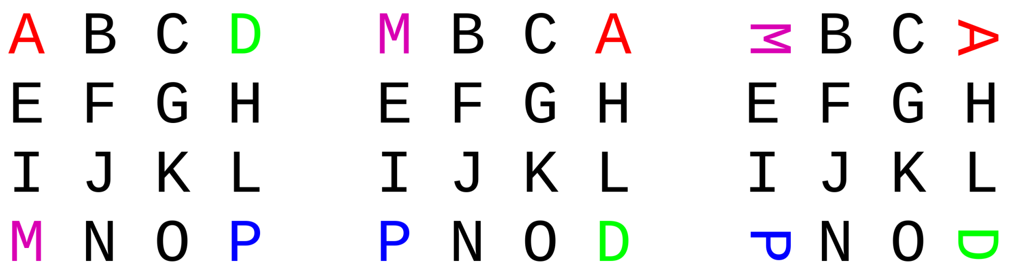
The program we give you implements an algorithm called "follow the cycles." One advantage of this algorithm is that it can rotate a matrix in place. At a high-level, the idea is that a rotation of an image (i.e., a matrix of bits) is simply a permutation of the bits in that matrix, and it is well known that any permutation can be decomposed into a set of cyclic permutations. When rotating a square matrix, the decomposition comprises a set of cyclic permutations of size four. The algorithm simply enumerates and performs each of these cyclic permutations.

In a rotation of an $N \times N$ matrix, the bit in position (j, i), i.e., row j, column i, is mapped to position $(i, N - j - 1)$. The cycle of length four starting at bit (j, i) is then

$$\begin{array}{rcl} (j, i) & \to & (i, N - j - 1) \\ (i, N - j - 1) & \to & (N - j - 1, N - i - 1) \\ (N - j - 1, N - i - 1) & \to & (N - i - 1, j) \\ (N - i - 1, j) & \to & (j, i). \end{array}$$

Upon rotation of a single bit to its final rotated destination, that bit displaces a different bit located there. As an optimization to keep this algorithm in place, the algorithm now simply rotates that displaced bit which in turn displaces a different bit, and then rotates that one, etc. After four of these operations, the final bit should land in the location of the first rotated bit of this series of four. Since this bit has already been rotated, it is safe to simply overwrite that destination. Figure 2 illustrates this bit rotation technique applied to a 4 by 4 matrix. In the example, only one of the cycles have been implemented, the one starting at bit $(0, 0)$.

The trick for the faster algorithm is to take the same approach, but scale it so that instead of



(a) Original image (b) Image after translating blocks (c) Image after rotating blocks

Figure 3: Illustration of rotation transformation performed on corner blocks of an image.

moving one bit at a time, we move entire blocks of bits at a time to their final relative destinations. And then individually, you rotate each block of bits. Figure 3 is a sample illustration of the block-rotation technique. If you were to continue this pattern accordingly with all of the other blocks, you would eventually achieve a fully rotated image. One thing to be careful about with this algorithm is that there must be symmetry in the shape of the image: it won't work for general rectangular images. It is safe, however, to ignore this detail as the problem specification only requires rotation of images that are square.

The row-column-row algorithm

Another algorithm that you may find helpful to investigate, especially for small matrices, is based on the following observation: Any permutation of the elements in a rectangular matrix can be decomposed into a set of permutations on the rows of the matrix, followed by a set of permutations on the columns, followed by a set of permutations on the rows. In particular, to rotate a square matrix, it suffices to circularly rotate the jth row to the left $j+1$ positions, for $j = 0, 1, \ldots, N-1$; then circularly rotate the ith column down $i+1$ positions, for $i = 0, 1, \ldots, N-1$; and then circularly rotate the jth row to the left j positions, for $j = 0, 1, \ldots, N-1$.

A proof of the row-column-row theorem can be found in *Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes*, by F. T. Leighton, Morgan Kaufmann, 1992, pp. 186–197. We will also present some algorithms in recitation on September 17. There may also be other competitive approaches. Think, code, and profile! The TA's are happy to discuss your ideas with you at office hours.

5 Evaluation

Grade breakdown

We will grade your project submission based on the following point distribution:

	Beta	*Final*
Team contract	3%	—
Correctness	7%	10%
Performance (assuming correct code)	20%	25%
Performance portability	5%	10%
Addressing MITPOSSE comments	—	10%
Write-up	5%	5%
Total	40%	60%

This point distribution serves as a *guideline* and not as an exact formula.

We recommend that you use pair programming for this assignment. It can be hard to divide the work in this project into independent components, and it's easier to balance commits (a requirement). More importantly, you will likely discover that a "backseat driver" helps you to avoid silly errors that may later slow you down when testing and debugging. Studies have shown, perhaps paradoxically, that the development of high-quality correct code goes significantly faster with pair programming than with independent programmers. Yes, the independent programmers can produce more lines of code in a given amount of time, but they lose much more in testing and debugging. Also, pair programming means (or should mean) that both programmers understand the code and they are better equipped to make changes independently, should it prove necessary. Finally, working together makes it easier to balance your commits.

The staff will review your Git commit logs to assess the dynamics of your team. *Please ensure that all team members author an equivalent amount of project commits.*

Now, let's run through each of these grading components in more detail.

Correctness

Correctness grading is more subjective than performance 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 credit. If your implementation is essentially correct, but fails occasionally on some dimensions, you can expect roughly 70% to 80% of the points. If your implementation fails more rotation sizes but the performance tests still run, you can expect 50% to 70%. If the performance tests don't run, you will get a low correctness grade.

If you are aware of a correctness bug in your own implementation, you will benefit if you inform us of the deficiency in your submission write-up. Please include what steps you have taken to debug it, even though your debugging might not have been 100% successful. If you appear completely unaware of a bug, you will receive no correctness points for that test case.

Performance (assuming correct code)

Your performance grade for your beta release is based primarily on how fast your *correct* program runs. That means you should focus on building a test infrastructure for the project before spending too much time optimizing for performance. Two or three days after the project becomes available, the course staff will announce a baseline performance goal. If you can beat the baseline, you are guaranteed at least a grade of B on the performance portion of the beta. But watch out not to spend too much time optimizing without carefully ensuring that your code stays correct. The higher the number of test cases on which your submitted solution fails, the lower your grade.

You are not competing with other students for your grade. There is no curve dictating how many A's, B's, and C's there will be. Instead, the course staff builds reference implementations against which the performance of your code is measured. Although we will sometimes share the binaries (never the source code) of our reference implementations with you, generally, we do not.

Performance portability

A portion of your project grade will depend on the performance robustness of your optimizations. Does your code continue to perform well when the underlying hardware changes? Or is it "overfitted" (overly specialized) for a specific hardware platform, running fast on that platform but much more slowly on other platforms?

Most of your performance grade will be measured on the standard hardware that you have access to. But for the performance-portability part of the grade, we will run your code on other machines, some of which you may not have access to. And you won't know in advance what they are. Since we're generally interested in scaling upward — those machines will typically be larger than your standard hardware. They will often have more memory, more cache, more processing cores (although that's irrelevant for this first project, since your code will be not be multithreaded), more vector lanes, etc. If you have "overfitted" your optimizations to the parameters of the standard machine, you may find that your code is tuned badly for larger machines and won't run as well.

A good strategy is to optimize first for the standard platform, since that's what most of your grade will be based on. Then see what you can do to remove ***voodoo constants*** (specific machine-based parameters in your code) so that your code is as independent as possible of a specific hardware configuration.

Addressing MITPOSSE comments

The MITPOSSE Deputies will give you feedback in GitHub on your code quality. We expect that you will respond to and act upon their comments thoughtfully in your final submission. We will review the MITPOSSE comments, your final submission, and your write-up to ensure that you are addressing the MITPOSSE comments.

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. Follow the Google C++ style guide (that is, the parts that apply to C) at `https://google.github.io/styleguide/cppguide.html`. 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.

When multiple people are working on the same codebase, it can be difficult to maintain a consistent style, unless everyone agrees to a common style. For this class, we shall use Google's style. We have provided a Python script `clint.py`, which is designed to check a subset of Google's style guidelines for C code. To run this script on all source files in your current directory use the command

```
$ python clint.py *
```

The code the staff has provided contains no Google-style errors. You should use this tool to clean up your source code for this project.

A helpful tool to automatically format your code to Google style guide is

```
$ # SYNTAX: clang-format -i -style=Google <file(s) to format>
$ clang-format -i -style=Google *.c *.h
```

Write-ups

Please submit write-ups, including a project log, for the beta and final releases. A good write-up helps the course staff (who understand the assignment and its general strategies but have never seen your code) to fairly grade your assignment. In addition, your beta write-up can help your MITPOSSE Deputy to advise you effectively. Please make sure to commit a copy of your beta write-up to the top level of your repo.

Your team's beta write-up should include the following topics:

- A brief overview of your design, particularly what improvements you made over the starter code for your beta design. Diagrams can be especially helpful. Your overview is *not* a substitute for appropriate documentation within the code.

- The general state of completeness and expected performance of your implementation, as well as any bugs or gotchas that you are aware of. You may also discuss your plans for your final release, especially if you had a good idea that you didn't have time to get working for the beta.
- Any additional information that you feel would be helpful to the staff or MITPOSSE Deputy in understanding your submission, e.g., if you spent a lot of time on failed approaches that didn't result in a speedup, etc.
- An acknowledgment of any help received from course staff, classmates on Piazza, or publicly available materials. (Please review the Academic Honesty section of Handout 1 Course Information.)

Your final write-up can be an update of your beta write-up. In addition to the topics above, it should include the following:

- An overview of changes you made to your beta release and what motivated you to do them, e.g., surprised by performance ranking, new revelations after your MITPOSSE meeting, ideas conceived before the beta deadline but ran out of time implementing them, etc.
- Comments on your meeting with your assigned MITPOSSE Deputy.
- Any updates to the acknowledgment, including specifically which of classmates' beta codes may have inspired you.

Your write-up should include a project log indicating how you spent your time. Tracking your time will help you and your team understand for yourselves whether you are working effectively. Do you get going in a timely manner? How much total time did each of you spend? Where did you spend the most time? Your logs will also help the course staff make adjustments so that student time is well spent. Figure 4 shows an example log for two fictional partners, Nek and Ping. To manage your log, you may choose whatever method works best to allow you to submit an easy-to-read log: a simple spreadsheet, a website such as `https://clockify.me`, etc.

6 Submitting your work

When you complete your beta release, your team will need to submit your code with Git and, the next day, submit a write-up of your project to Gradescope. Please also commit the beta write-up to your Git repo for your MITPOSSE Deputies. Similarly, when you finish your final release, you need to submit your code and again a write-up the next day.

Submit your code with Git before the beta and final deadlines. For grading, we will use the code on the main branch of your team's repository on GitHub at the time of the submission deadline.

If you want to keep developing near the deadline, use a branch other than main. Remember to explicitly add new files to your repository before committing and pushing your final changes:

```
$ git add <new-files>
$ git commit -am "Your commit message"
$ git status
$ git push
```

If `git status` shows any modified files, then you probably have not committed your code into your repository properly.

In keeping with good programming practice, you should commit incremental changes to your repository as you write and test your code. Remember to balance these commits among your team members. We expect you to submit your code promptly by the due date.

A quick note about `git commit -a`: this command will commit all modifications to tracked files to your local repository. If you only wish to commit changes to certain files, you can `git add` them explicitly and `git commit` will commit only files that have been manually staged (use `git status` to check what's been staged). If this is confusing, please just stick to `git commit -a`.

7 Good luck, and have fun!

We hope you enjoy the first project of 6.172. Remember that although 6.172 is an 18-unit class, it's easy to spend more than 18 hours — but please don't. You can *always* make your code faster, but there tend to be diminishing returns. Start early and budget your time wisely, both within 6.172 and to keep a balance with your other classes and activities. Good luck, and happy coding!

#	Date	Start time	Duration		Description
			Nek	Ping	
1	Tue, Sep 10	4:30 PM		1.25	Read handout carefully.
2	Tue, Sep 10	6:00 PM	1		Read handout carefully.
3	Wed, Sep 11	10:00 AM	0.75		Checked out provided code (Code 0), studied it, compiled it, and tested it.
4	Thu, Sep 12	10:00 AM			Nek and Ping formed a team, filled out partner form.
5	Thu, Sep 12	4:00 PM	0.5	0.5	Profiled Code 0. Determined that it would be important to solve base cases efficiently.
6	Fri, Sep 13	10:00 AM		0.5	Designed an initial algorithm for base case.
7	Fri, Sep 13	3:00 PM	0.75		Designed an initial algorithm for base case.
8	Fri, Sep 13	6:00 PM	0.5	0.5	Discussed our two base-case algorithms and agreed on an implementation strategy for Code 1.
9	Fri, Sep 13	6:30 PM	1.5	1.5	Pair programmed Code 1 for base case. Debugged and tested. Added some tests to the regression suite.
10	Sat, Sep 14	12:30 PM	0.5		Designed a general algorithm (Code 2).
11	Sat, Sep 14	2:30 PM		1	Profiled Code 1 performance. Tweaked implementation. Tested. Gained almost 1 tier.
12	Sat, Sep 14	7:00 PM	0.25		A BOTEC showed the Code 2 algorithm couldn't be as fast as we thought. Texted Ping.
13	Sat, Sep 14	8:00 PM		0.5	Figured out a better algorithm. Emailed revised algorithm to Nek.
14	Sun, Sep 15	2:00 PM	0.75	0.75	Pair-programmed Code 2, the revised algorithm for the general case.
15	Sun, Sep 15	2:45 PM	1.25	1.25	Debugged and tested Code 2. Profiling gives us 6 tiers!
16	Sun, Sep 15	8:00 PM	0.25		Discovered a compiler switch that gave us over half a tier with no work.
17	Sun, Sep 15	10:00 PM		0.5	Implemented more tests. Discovered a bug in Code 2. Crashes on some big inputs. Texted Nek.
18	Mon, Sep 16	10:00 AM	1.5		Tried to find the bug Ping discovered in Code 2. No luck, but learned how to use GDB. :^)
19	Mon, Sep 16	4:30 PM		1	Tried to find the bug in Code 2. This is FRUSTRATING!
20	Mon, Sep 16	6:00 PM		0.25	TA at office hours saves the day! There IS a difference between signed and unsigned ints!
21	Mon, Sep 16	7:15 PM	1		Added a test for the bug to our regression suite. Implemented the fix — Code 3. Lost almost a tier of performance, but the code is correct.
22	Mon, Sep 16	8:00 PM		1	Double-checked Nek's fix. Tweaked Code 2 in five different ways, but lost performance on every single tweak, except one, which only gave 10% of a tier.
23	Tue, Sep 17	7:30 PM	1	1	Videoconferenced to profile our unsuccessful tweaks together. Discovered lots of cache misses. Figured out a way for the code to do less work without incurring lots of cache misses. New design is Code 3.
24	Tue, Sep 17	8:30 PM	0.75	0.75	Implemented, debugged, and tested Code 3. Added more tests to regression suite. Gained 2 tiers!
25	Wed, Sep 18	7:15 PM	1.5	1.5	Decided not to make any last-minute tweaks for fear we'd break things. Focused instead on making sure Code 3 is well tested. Found and fixed several minor bugs. Added tests to the regression suite. (Good thing we did! One of our fixes had a bug, which was caught by a test we had just added.) Submitted our Beta with plenty of time to spare.
Total			**13.75**	**13.75**	

Figure 4: A sample project log.