This document describes the available bonus projects for this term. **Please read through everything carefully before starting to work on the projects:** failure to do so might likely result in some large disappointments in a month’s time!

**Philosophy**

These bonus projects are opportunities to explore some of the concepts presented in this course in greater detail, as well as gain some experience building larger software than you have done in CSC148. This can be a valuable experience in your development as a computer science student, but be warned: both projects will require a large time commitment to complete successfully, likely even more than a “final project” you might have in another programming course. Moreover, since we’re nearing the end of the term, you’ll likely be very busy with wrapping up course work and studying for final exams.

Do not take these projects lightly! I am not looking to reward effort or competence (either theoretical or technical); I want to be impressed. From your own experience, you know as well as I do how easy it is to distinguish good tools from great ones. So if you choose to attempt one or both of these projects, **do it because you want the learning experience, not because you hope to increase your mark.**

**General Specifications**

- You may work in groups of up to **FOUR** students from either section. You may work in different groups for different projects. In particular, **each student may submit work for both projects.**

- Project submissions must be received by **11:59 p.m., Sunday April 20.**

- Each project will be evaluated by the instructor; possible grades are 0, 1, and 2. I’ll do my best to provide detailed feedback to each submission, but due to time constraints, this might not happen until after the term is over.

- I will work with your submissions on CDF machines.

- Project grades will be applied as a bonus directly to each group member’s final grade.

- Remark requests will not be accepted. If you find yourself disappointed that I haven’t rewarded your effort, remember that you should be doing this for yourself, not for some number.

**Submission Instructions**

When your group is ready to submit, do the following:

1. Put all of your source code and documentation in a single folder. **DO NOT** include compiled files (e.g., `.class`, `.a`, `.o`). Only include precisely what you need in order to build your program.

2. Make sure **all** of your files have the full names and CDF accounts, and University of Toronto email addresses of all of your group members.

3. Make sure you can compile and run your code on the CDF machines.
4. Zip/tar/otherwise compress the folder containing your files.
5. Create a group on MarkUs for the correct project, and upload your single compressed file to MarkUs.
6. Wait for my feedback via email. I’m only using MarkUs as a submission system; you will not receive a grade or feedback on MarkUs.

Please note: failure to accomplish any of the above tasks by the deadline WILL result in me not marking your work. Reading and following requirements carefully is essential for success here and in your future.

What I’m Looking For

For both projects (and basically any software project of this size), there are three components you need to be constantly aware of: product, source code, and documentation.

Your Product

The project descriptions on the following pages mainly deal with the functionality I would like your tool to provide, so you have some clear-cut instructions. However, to truly impress me (and hence get a 2% bonus), you need to provide some extra feature(s) that go beyond what I’ve thought of and given you already.

Fundamental question: How much of your OWN thought have you put into what your product should do?

Your Source Code

Getting your product to work isn’t the end of the story; you must make sure your source code is well-designed and follows good style and naming conventions. This doesn’t mean you need to have an object-oriented approach; but your functions and data structures should be appropriately modular, so that you don’t have any Frankensteinian chunks of code. Feel free to split your code across multiple files. Running something like PEP8 (Python) or code style/lint checkers in other languages would be a good idea.

You may complete these projects in whatever programming language you want, as long as you are able to build and run it successfully on CDF. If you want to use any external libraries for anything other than the two tasks I’ve mentioned in the project descriptions, check with me first; my answer will probably be “no.” Also, please remember that users should be able to use your program on CDF without downloading any additional files.

Fundamental question: How easy is it for an inexperienced team member (e.g., David) to understand and modify your code?

Documentation

Of course, making your code readable is aided by excellent documentation. I’m looking for proper documentation within your code (including, but not at all limited to, pre- and postconditions for all functions). I’m also looking for user-facing documentation: a README (.txt, .doc, .pdf) that guides the user from decompressing your submission, through compiling your code, all the way to running your tool and exploring its various features. Keep in mind that the intended audience shouldn’t be me, but your fellow CSC236 classmates.

Fundamental question: If a user gets stuck trying to run/use your software, is there enough written support to help him or her figure out what to do?

\(^1\)not “may” or “might”
Project 1: Translating Python to Dafny

We saw in class Microsoft Research’s tool Dafny, a small programming language with powerful program verification features. In particular, we saw how it supported pre- and postconditions, loop invariants, and even loop size measures.

Your task in this project is to build a **Python-to-Dafny Translator** which takes in as input a Python file (.py), and output a syntactically correct Dafny file. The main use of this would be allowing users to write Dafny-style pre/postconditions, invariants, and size measures as comments in a Python program, and have Dafny actually verify these statements in the Dafny translated output.

As a simple example, consider the following Python program:

```python
def add(x, y):
    # Post: returns x + y
    z = x + y
    return z
```

This might be translated to a Dafny program which looks something like:

```dafny
method add(x: int, y: int) returns (z: int)
    ensures z == x + y;
{
    z := x + y;
    return z;
}
```

This is only one possible translation; it will be up to you write translation functions that handle as large a subset of Python as possible, while still producing valid Dafny code that corresponds directly to the Python code.

Some general comments about this project:

- The ultimate goal should be to **encourage students to write correctness specifications as comments in their Python programs**. We’re trying to support this habit by giving them a tool which will automatically verify whether their specifications are actually good.

- A very important requirement for this project is that you should require as LITTLE EFFORT from the Python programmer as possible. While you can make your job easier by making the Python programmer give you more information as comments in the original program, the more people have to type, the less likely they are to use your tool.

- That said, it’s up to you to specify how Python programmers should format their Dafny-comments for your program to understand them.

- Also, users should still be able to write regular, non-Dafny comments as well, which your program should leave untouched.

- My advice is to start with very simple Python programs (e.g., just variable assignments and arithmetic) and once you have that working, slowly add more and more features. Here is an incomplete selection of Python features which CSC108/CSC148 students commonly use:
  - If-Else blocks
  - Loops
  - Recursion
  - Lists
  - Classes

- Your translator will have to parse (“read and understand”) Python programs. You can probably get away with simple local (i.e., line-by-line) transformations for a lot of the basic features (simple example: replace the assignment operator = with :=), but this may fail you at some point. **You may use external parsing libraries** – and in particular, the ast Python module might be helpful if you’re working in Python – but be warned that these will take some time to understand.

---

2 State in your documentation any that you use, of course.
Project 2: Visualizing CPDQ

Recall from lecture the divide-and-conquer algorithm for solving the Closest Pair problem. Your task for this project is to create an interactive tool to visualise this algorithm. The main goal here is an educational one: undergraduate computer science students should be able to use your program to understand how this algorithm really works!

Here are some general comments about this project:

• You’re expected to implement the divide-and-conquer algorithm yourself, no matter what language you choose. It’s not that hard finding implementations online, but this negates one of the main learning opportunities in this project, so don’t do it. Also, make sure you improve on the algorithm presented in class, so that your algorithm actually runs in $O(n \log n)$ time, not $O(n \log^2 n)$ time.

• You are more than welcome to use whatever graphics libraries you like though I strongly recommend for ease of use that you stick with the popular ones. While your graphics need not be too complex (points and lines, mostly), I expect the visuals to be clean and clear; show some aesthetic consideration, and make the interface look good.

• This task is simpler than the previous one, from a technical point of view. However, keep in mind that the overall difficulty is the same, because for this task you really need to spend time designing and creating a good user interface.

• Note that a good interface need not be flashy or complex, but should be easy to pick up by other students of a similar level of experience.

• In terms of interactivity, it would be cool if users could create their own points and then visualise the algorithm running on that input. Doubly cool if users could pause, step forward, and step back through the algorithm.

• Keeping with the theme of “no call stack”, it would be nice if there were two modes: one with all of the recursive calls visualised, and one where the results of the recursive call appear immediately.

3State in your documentation any that you use, of course.