In [22]:
```
# This cell runs automatically.  Don't edit it.
from CSE142L.notebook import *
from cfiddle import *
```

Double Click to edit and enter your
1. Name
2. Student ID
3. @ucsd.edu email address

Lab 1: The Performance Equation

Welcome to the first lab of CSE142L!

The main goals of this lab are:

1. To get you set up the lab environment
   A. Github
   B. Github Classroom
   C. Docker
   D. UCSD's Datahub cluster
   E. Jupyter Notebook
2. Gain experience with the performance equation
3. Collect a list of **Interesting Questions** that you probably cannot answer now, but will be able to by the end of the course.

This lab will be completed on your own.

Check Gradescope for due date(s).

1 FAQ and Updates

None yet.

2 Pre-Lab Reading Quiz

Part of this lab is a pre-lab quiz. The pre-lab quiz is on Canvas. It is due **on Wednesday before Section A meets**. It's not hard, but it does require you to read over the lab before class. If you are having trouble accessing it, make sure you are **logged into Canvas**.

2.1 How To Read the Lab For the Reading Quiz

The goal of reading the lab before starting on it is to make sure you have a preview of:

1. What's involved in the lab.
2. The key concepts of the lab.
3. What you can expect from lab.
4. Any questions you might have.

These are the things we will ask about on the quiz. You *do not* need to study the lab in depth. You *do not* need to run the cells.

You should read these parts carefully:

- Paragraphs at the top of section/subsections
- The description of the programming assignment
- Any other large blocks of text
- The "About Labs in This Class" section (Lab 1 only)
- The programming assignment description.

You should skim these parts:

- The questions.

You can skip these parts:

- The "About Labs in This Class" section (Labs other than Lab 1)
- Commentary on the output of code cells (which is most of the lab)
- Parts of the lab that refer to things you can't see (like cell output)
- Solution to completeness questions.

2.2 Taking the Quiz

You can find it here: https://canvas.ucsd.edu/courses/40763/quizzes (https://canvas.ucsd.edu/courses/40763/quizzes)

The quiz is "open lab" -- you can search, re-read, etc. the lab.

You can take the quiz 3 times. Highest score counts.

3 Browser Compatibility

We are still working out some bugs in some browsers. Here's the current status:

1. Chrome -- well tested. Preferred option. **Required for Moneta**

2. Firefox -- seems ok, but not thoroughly tested.
3. Edge -- seems ok, but not thoroughly tested.
4. Safari -- not supported at the moment.
5. Internet Explorer -- not supported at the moment.

At the moment, the authentication step must be done in Chrome. You usually *will not* have to re-authenticate between labs, so if things work OK for the first, things will probably work here.

4 About Labs In This Class

This section is the same in all the labs. It's repeated here for your reference.

Labs are a way to **learn by doing**. This means you *must* **do**. I have built these labs as Jupyter notebooks so that the "doing" is as easy and seamless as possible.

In this lab, what you'll do is answer questions about how a program will run and then compare what really happened to your predictions. Engaging with this process is how you'll learn. The questions that the lab asks are there for several purposes:

1. To draw your attention to specific aspects of an experiment or of some results.
2. To push you to engage with the material more deeply by thinking about it.
3. To make you commit to a prediction so you can wonder why your prediction was wrong or be proud that you got it right.
4. To provide some practice with skills/concepts you're learning in this course.
5. To test your knowledge about what you've learned.

The questions are graded in one of three ways:

1. "Correctness" questions require you to answer the question and get the correct answer to get full credit.
2. "Completeness" questions require you to answer the question.
3. "Optional" questions are...optional. They are there if you want to go further with the material.

Some of the "Completeness" problems include a solution that will be hidden until you click "Show Solution". To get the most from them, try them on your own first.

Many of the "Completeness" questions ask you to make predictions about the outcome of an experiment and write down those predictions. To maximize your learning, think carefully about your prediction and commit to it. **You will never be penalized for making an incorrect prediction.**

You are free to discuss "Completeness" and "Optional" questions with your classmates. You must complete "Correctness" questions on your own.

If you have questions about any kind of question, please ask during office hours or during class.

4.1 How To Succeed On the Labs

Here are some simple tips that will help you do well on this lab:

1. Read/skim through the entire lab *before* class. If something confuses you, you can ask about it.
2. Start early. Getting answers on edstem/piazza can take time. So think through the lab questions (and your questions about them) carefully.
   A. Go through the lab once (several days before the deadline), do the parts that are easy/make sense
   B. Ask questions/think about the rest
   C. Come back and do the rest.
3. Start early. The DSMLP cluster gets busy and slow near deadlines. "The cluster was slow the night of the deadline" is not an excuse for not getting the lab done and it is not justification for asking for an extension.
4. Follow the guidelines below for asking answerable questions on edstem/piazza.

You may think to yourself: "If I start early enough to account for all that, I'd have to start right after the lab was assigned!" Good thought!

The Cluster Will Get Slow DSMLP and our cloud machines will get crowded and slow *before every deadline*. This is completely predictable. DSMLP can also get crowded due to deadlines in other courses. You need to start early so you can avoid/work around these slowdowns. Unless there's some kind of complete outage, we will not grant extensions because the servers are crowded.

4.2 Getting Help

You might run into trouble while doing this lab. Here's how to get help:

1. Re-read the instructions and make sure you've followed them.
2. Try saving and reloading the notebook.
3. If it says you are not authenticated, go to the the login section of the lab and (re)authenticate.
4. If you get a `FileNotFoundError` make sure you've run all the code cells above your current point in the lab.
5. If you get an exception or stack dump, check that you didn't accidentally modify the contents of one of the python cells.
6. If all else fails, post a question to edstem/piazza.

4.3 Posting Answerable Questions on Edstem/Piazza

If you want useful answers on edstem/piazza, you need to provide information that is specific enough for us to provide a useful answer. Here's what we need:

1. Which part of which lab are you working on (use the section numbers)?
2. Which problem (copy and paste the *text* of the question along with the number).

If it's question about instructions:

1. Try to be as specific as you can about what is confusing or what you don't understand (e.g., "I'm not sure if I should do *X* or *Y*.")

If it's a question about an error while running code, then we need:

1. If you've committed anything, your github repo url.
2. If you've submitted a job with `cse142` you *must* provide the job id. It looks like this: `544e0cf2-4771-43c3-86f8-1c30d7af601f`. With the id, we can figure out just about anything about your job. Without it, we know nothing.
3. The *entire* output you received. There's no limit on how long an edstem/piazza post can be. Give us all the information, not just the last few lines. We like to scroll!

For all of the above **paste the text** into the edstem/piazza question. Please **do not provide screen captures.** The course staff refuses to type in job ids found in screen shots.

We Can't Answer Unanswerable Questions If you don't follow these guidelines (especially about the github repo and the job id), we will probably not be able to answer your question on edstem/piazza. We will archive it and ask you to re-post your question with the information we need.

4.4 Keeping Your Lab Up-to-Date

Occasionally, there will be changes made to the base repository after the assignment is released. This may include bug fixes and updates to this document. We'll post on piazza/edstem when an update is available.

In those cases, you can use the following commands to pull the changes from upstream and merge them into your code. You'll need to do this at a shell. It won't work properly in the notebook. Save your notebook in the browser first.

```
cd <your directory for this lab>git remote add upstream $(cat .star
ter_repo)  # You need to do this once each time you checkout a new
 lab. It will fail
                                              # harmlessly if you r
un it more than once.
cp Lab.ipynb Lab.backup.ipynb                 # Backup your work.
git commit -am "My progress so far."          # commit your work.
git pull upstream main --allow-unrelated-histories -X theirs # pull
the updates
```

Or you can use the script we provide:

```
./pull-updates
```

Then, reload this page in your browser.

4.5 How To Use This Document

You will use Jupyter Notebook to complete this lab. You should be able to do much of this lab without leaving Jupyter Notebook. The main exception will be some parts of the some of the programming assignments. The instructions will make it clear when you should use the terminal.

4.5.1 Logging In

If you haven't already, you can go to [the login section of the lab]() and follow the instructions to login into the course infrastructure.

4.5.2 Running Code

Jupyter Notebooks are made up of "cells". Some have Markdown-formatted text in them (like this one). Some have Python code (like the one below).

For code cells, you press `shift-return` to execute the code. Try it below:

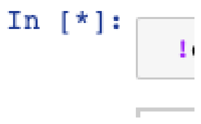
In [1]:
```
print("I'm in python")
```

Code cells can also execute shell commands using the `!` operator. Try it below:

In [2]:
```
!echo "I'm in a shell"
```

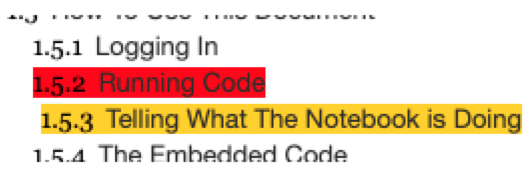
4.5.3 Telling What The Notebook is Doing

The notebook will only run one cell at a time, so if you press `shift-return` several times, the cells will wait for one another. You can tell that a cell is waiting if it there's a `*` in the `[]` to the left the cell:



```
In [*]:  !
```

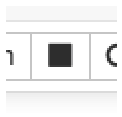
You'll can also tell *where* the notebook is executing by looking at the table of contents on the left. The section with the currently-executing cell will be red:


1.5.1 Logging In
1.5.2 Running Code
1.5.3 Telling What The Notebook is Doing
1.5.4 The Embedded Code

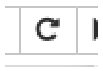
4.5.4 What to Do Jupyter Notebook It Gets Stuck

First, check if it's actually stuck: Some of the cells take a while, but they will usually provide some visual sign of progress. If *nothing* is happening for more than 10 seconds, it's probably stuck.

To get it unstuck, you stop execution of the current cell with the "interrupt button":



You can also restart the underlying python instance (i.e., the confusingly-named "kernel" which is not the same thing as the operating system kernel) with the restart button:

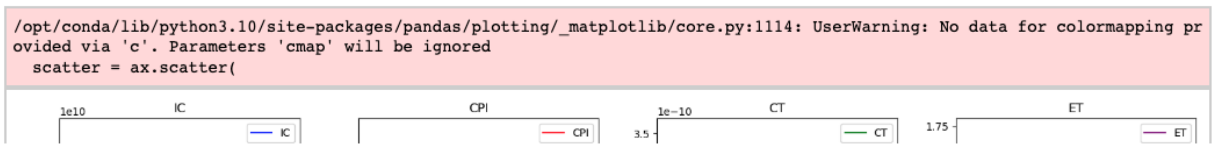


Once you do this, all the variables defined by earlier cells are gone, so you may get some errors. You may need to re-run the cells in the current section to get things to work again.

You can also try reloading the web page. That will leave Python kernel intact, but it can help with some problems.

4.5.5 Common Errors and Non-Errors

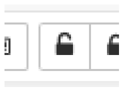
1. If you get `sh: 0: getcwd() failed: no such file or directory`, restart the kernel.
2. If you get `INFO:MainThread:numexpr.utils:Note: NumExpr detected 40 cores but "NUMEXPR_MAX_THREADS" not set, so enforcing safe limit of 8.`. It's not a real error. Ignore it.
3. If you get a prompt asking `Do you want to cancel them and run this job?` but you can't reply because you can't type into an output cell in Jupyter notebook, replace `cse142 job run` with `cse142 job run --force`. (see useful tip below.)
4. If you get an `Error: Your request failed on the server: 500 Server Error: Internal Server Error for url=http://cse142l-dev.wl.r.appspot.com/file`, trying running the job again.
5. Sometimes `cse142 job run` will just sit there and seemingly do nothing. Weirdly, interrupting the kernel (button above) seems to jolt it awake and cause it to continue.
6. Warnings like this in pink about deprecated or ignored arguments are harmless:


```
/opt/conda/lib/python3.10/site-packages/pandas/plotting/_matplotlib/core.py:1114: UserWarning: No data for colormapping pr
ovided via 'c'. Parameters 'cmap' will be ignored
  scatter = ax.scatter(
```

7. If you get `http.cookiejar.LoadError: '/home/youruserrname/.djr-cookies.txt` does not look like like a Netscape format cookies file.` remove the file and re-authenticate.

4.5.6 Useful Tips

1. If you need to edit a cell, but you can't you can unlock it by pressing this button in the tool bar (although you probably shouldn't do this because it might make the lab work incorrectly. A better choice is to copy and paste the cell, *and then* unlock the copy):



4.5.7 The Embedded Code

The code embedded in the lab falls into two categories:

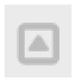
1. Code you need to edit and understand.
2. Code that you do not need to edit or understand -- it's just there to display something for you.

For code in the first category, the lab will make it clear that you need to study, modify, and/or run the code. If we don't explicitly ask you to do something, you don't need to.

Most of the code in the second category is for drawing graphs. You can just run it with shift-return to the see the results. If you are curious, it's mostly written with `Pandas` and `matplotlib`. These cells should be un-editable. However, if you want to experiment with them, you can copy *the contents* of the cell into a new cell and do whatever you want (If you copy the cell, the copy will also be uneditable).

Most Cells are Immutable Many of the cells of this notebook are uneditable. The only ones you should edit are some of the code cells and the text cells with questions in them.

Pro Tip The "carrot" icon in the lower right (shown below) will open a scratch pad area. It can be a useful place to do math (or whatever else you want.



4.5.8 Showing Your Work

Several questions ask you to show your work for calculations. We don't need anything fancy. Many of the questions ask you to compute something based on results of an experiment. Your experimental results will be different than others', so your answer will be different as well.

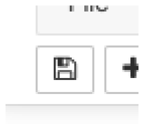
To make it possible to grade your work (and give you partial credit), we need to know where your answer came from. This why you need to show your work. For instance this would be fine as answer to "On average, how many weeks do you have per lab?":

```
Weeks in quarter/# of labs = 10/5 = 2 weeks/lab
```

2 significant figures is sufficient in all cases, but you can include more, if you want.

If you are feeling fancy, you can use LaTex, but it's not at all required.

When it's appropriate, you can also paste in images. However, Jupyter Notebook is flaky about it. Saving your notebook frequently by clicking the disk icon seems to help:



4.5.9 Answering Questions

Throughout this document, you'll see some questions (like the one below). You can double click on them to edit them and fill in your answer. Try not to mess up the formatting (so it's easy for us to grade), but at least make sure your answer shows up clearly. When you are done editing, you can `shift-return` to make it pretty again.

A few tips, pointers, and caveats for answering questions:

1. The answers are all in github-flavored markdown (https://guides.github.com/features/mastering-markdown/) with some html sprinkled in. Leave the html alone.
2. Many answers require you to fill in a table, and many of the `|` characters will be missing. You'll need to add them back.
3. The HTML needs to start at the beginning of a line. If there are spaces before a tag, it won't render properly. If you accidentally add white space at the beginning of a line with an html tag on it, you'll need to fix it.
4. Text answers also need to start at the beginning of a line, otherwise they will be rendered as code.
5. Press `shift-return` or `option-return` to render the cell and make sure it looks good.
6. There needs to be a blank line between html tags and markdown. Otherwise, the markdown formatting will not appear correctly.

You'll notice that there are three kinds of questions: "Correctness", "Completeness", and "Optional". You need to provide an answer to the "Completeness" questions, but you won't be graded on its correctness. You'll need to answer "Correctness" questions correctly to get credit. The "Optional" questions are optional.

Give it a try:

Question 1 (Completeness)

What will you do with your Jupyter Notebook to turn it in (Delete *all* incorrect answers)? Then fill in the table. Fix the formatting of the last line and the closing tag.

1. Print it out on paper and slide it under the professor's door.
2. Follow the directions at the end of the lab to produce a pdf
3. Read it aloud to a TA during office hours
4. Submit via gradescope.

Random Fact	Value	Second choice
Your favorite color		
Favorite food on campus		
Time of day		

```
This answer is formatted poorly.
</div>
```

Show Solution

5 Logging In To the Course Tools

In the course you will use some specialized tools to let you perform detailed measurements of program behavior. To use them you need to login with your `@ucsd.edu` email address using the instructions below. **You need to use the email address that appears on the course roster. That's the email address we created an account for. In almost all cases, this is your `@ucsd.edu` email address.**

You'll probably only have to do this once this quarter, but if you get an error about not being authenticated, just re-authenticate. You can return to this notebook (or any other of the lab notebooks) to login at any time.

Here's what to do:

1. Enter your `@ucsd.edu` email address in quotes after `login` below. It'll take a few seconds to load.
2. Click the google "G" login button below and login with your `@ucsd.edu` email address.
3. **Click the google button regardless of whether it says "sign in" or "signed in". Then be sure to select your `@ucsd.edu account` if it shows you multiple google acocunts**
4. You'll see a very long string numbers an letters appear above. Click "Copy it" to copy it.

Note: If it doesn't give you a choice about which account to log into and authentication fails, that means you are logged into a single Google account and that account is *not* your `@ucsd.edu` account. You'll have to log into your `@ucsd.edu` through Gmail or through Chrome's account manager and then try again.

Use Chrome The login process doesn't seem to work properly with Safari or Firefox. Use Chrome to login. You can use any of the other compatible browsers you want for the doing the rest of the lab, and it should be fine.

In [5]:
```
login("<Your @ucsd.edu email address>")
```

Next step: Paste it below between the quote marks. Press `shift-return` .

In [6]:
```
token("your_token")
```

It should have replied with

```
You are authenticated as <your email>
```

You are now logged in! Try submitting a job:

In [ ]:
```
!cse142 job run "echo Hello World"
```

If you see "Hello World", you're all set. Proceed with the lab!

Delete your token from the above cell. Because your token is esssentially your username and password combined, you should treat it like a password or ssh private key. **Sharing your token with another student or posessing another student's token is an AI violation**.

6 Grading

This is a 2 week lab.

Your grade for this lab will be based on your completion and submission of this notebook.

Part	value
Reading quiz	3%
Jupyter Notebook	65%

Part	value
Programming Assignment	30%
Post-lab survey	2%

We will grade 5 of the "completeness" problems. They are worth 3 points each. We will grade all of the "correctness" questions.

Check Gradescope for the due dates.

Instructions for submitting the lab are at the end of the lab.

No late work or extensions will be allowed.

7 Academic Integrity Agreement

To continue in the class, you need to agree to the following:

At UCSD, academic integrity[1] means that you have the courage, even when it is difficult, to only submit academic work that is honest, responsible, respectful, fair, and trustworthy. When you excel with integrity in computer science, it means that you:

Honest submit work that is a truthful demonstration of your knowledge and abilities (rather than the knowledge and abilities of another)

Responsible manage your time so that you are not pressured to complete an assignment at the last minute

Respectful acknowledge the contributions of others to your work by citing them when ve used their words or ideas (e.g., after I've spoken to classmates or after I've used portions of a code written by another if permitted)

Fair complete your academic work according to stated standards and expectations even when it takes longer or re struggling

Trustworthy can be trusted to be honest, responsible, respectful, and fair even when no one is watching you.

When you act contrary to these values, you are cheating. Cheating undermines trust between students and professors, the value of the UCSD degree, and your learning/development of skills.

While we can't list every behavior that would be cheating, we can give you some illustrative examples like the following:

- Submitting any source code written by another person or copied from another person, submitting homework answers which were produced by another student.
- Submitting code/homework you have previously submitted to another course for credit without first obtaining permission from the instructor. The same restriction holds for publicly available code/homework solutions that you haven't written. Taking notes taken during any discussions with classmates about an assignment is prohibited.

- Using words or text written by someone else without citing text appropriately. Every figure or sentence fragment must be appropriately decorated with quotation marks or indention to indicate very clearly that someone else wrote the text. In addition, the passage must be labeled with a citation or citation number which refers to a footnote or bibliographic entry. Citing a paper once is not enough. Remember: citations should be used to illuminate a viewpoint which you hold. They are not a substitute for expressing your own ideas in your own words.
- Submitting any portion(s) of an assignment you have previously submitted for credit in another course.
- Copying from a neighbor during an exam or using an unauthorized aid to help you on your exam.
- Altering a graded exam or assignment and resubmitting it for regrade
- Allowing someone else to complete an assignment or exam for you, or allowing them to pretend to be you in class (e.g., by signing an attendance form or clicking for you).
- Making available to others source code, documentation, or notes useful for completing an assignment. You should neither produce, procure, nor accept such material. This includes students in current, past, and future offerings of the course, and applies to electronic transmissions including email, web pages, ftp, and so on, as well as hard copy such as source code listings.
- Possessing (at any time) source code, data, or answers to homeworks or lab questions created by another student. Having had any of these items in your possession (e.g., in your directory on the campus servers or on your personal computer) at any time, constitutes cheating. You should never accept these materials from anyone for any reason.
- Running other students' code from your account or allowing another student submit code using your credentials.

If the behavior you are considering isn't listed here, don't assume that it is allowed. Rather, you should always do independent work unless told otherwise. And before completing your academic work in a certain way, you should ask is it honest, respectful, responsible, fair, and trustworthy. You can also ask yourself "Would I be okay if my methods were exposed to the TA, professors, and fellow students?" If the answer is no, you shouldn't do it.

If you have any questions about what is and isn't cheating, be sure to discuss them with the instructor.

Any student who cheats, thereby undermining integrity, will be reported to the Academic Integrity Office. Students who cheat face various disciplinary sanctions as well as academic penalty imposed by the instructor in the course. *Academic penalties include, but are not limited to, receiving a grade of 0 for the assignment or test in question, and receiving an 'F' for the course.*

[1] For more information on academic integrity, including how you can excel with integrity, as well as information on sanctioning guidelines for cheating, visit the Academic Integrity Office website at: http://academicintegrity.ucsd.edu (http://academicintegrity.ucsd.edu)

Question 2 (Correctness - 1pts)

Please affirm your adherence to this agreement

Type 'I excel with integrity' here: [`type it (leave the brackets)`]

By submitting this file, I, [`Your Name`], a student enrolled in CSE142L affirm the principle of academic integrity and commit to excel with integrity by completing all academic assignments in the manner expected as described above, informing the instructor of suspected instances of academic misconduct by my peers, and fully engaging in the class and its related assignments for the purpose of learning.

To electronically sign this document, Enter your full name, date, and student ID below:

[`full name`] / [`date`] / [`Student ID #`]

This document was written in part by Rick Ord, CSE Lecturer and Dr. Bertram Gallant, Director of the UCSD Academic Integrity Office.

8 Skills to Learn

1. Get access to docker
2. `git` /GitHub basics
3. Navigating a Jupyter Notebook
4. Think about code and predict its behavior.
5. Optimize a simple algorithm.

9 Building and Running Code

In these labs you will be doing most of your coding within Jupyter Notebook using an interactive code analysis system called `cfiddle` (https://github.com/NVSL/cfiddle (https://github.com/NVSL/cfiddle)). Cfiddle is a Python library that lets you write code, compile it, analyze it, execute it, and analyze its performance all in a Jupyter Notebook.

9.1 Compiling and Inspecting Code with CFiddle

Here's how you compile Hello World with Cfiddle:

In [ ]:
```
hello_world = build(code(r"""
#include"cfiddle.hpp" // Connect to Cfiddle's performance measurement mec
#include<iostream>

extern "C" void hello_world(int k) {
    start_measurement(); // start measuring
    uint64_t sum = 0;
    for(int i = 0; i < k; i++) {
        sum += i;
    }

    std::cout << "Hello World! The sum of the day is " << sum << ".\n";
    end_measurement(); // stop measuring
}

""", file_name="hello_world.cpp"))
```

The `code()` function takes a Python string, writes it to a file and returns the filename. The `r` before the string makes it a "raw" string which disables Python's escape sequence processing. `extern "C"` makes it easier to find `hello_world`. We'll learn more about it in the next lab.

The `hello_world` is now *an array* of Python objects that represent compiled code. Usually, we'll just be interested in the first element of the array. For instance, we can look at the code itself:

In [ ]:
```
hello_world[0].source()
```

Or it's assembly:

In [ ]:
```
hello_world[0].asm("hello_world")
```

We can also configure how the code is compiled. For instance, we can compile it with optimizations by setting the `build_parameters` argument:

In [ ]:
```
hello_world_twice = build("hello_world.cpp", build_parameters=arg_map(OPT
```

the `arg_map` is a utility function that generates all combinations of its arguments (we'll see more of this later), but for now the important part is that it will compile with and without optimizations. The result is that the `hello_world` has two entries in: The first is the compiled code without optimizations and second is the code with optimizations. We can now compare the assembly for the two:

In [ ]:
```
compare([hello_world_twice[0].asm("hello_world"), hello_world_twice[1].as
```

Question 3 (Optional)

What did the compiler do? (We'll investigate this in details in Lab 2)

9.2 Running Code With CFiddle

Cfiddle can also run code. Let's run `hello_world()` :

In [ ]:
```
results = run(hello_world, function="hello_world", arguments=arg_map(k=4))
```

The `function` argument tells `cfiddle` which function to invoke, and `arguments` get passed to the function.

Running the code took a while, because, for this class, `cfiddle` is configure to run your code on our servers in the cloud, but you can see the output buried in there:

```
Hello World! The number of the day is 6.
```

`results` now has some measurements about the execution of `hello_world()` . We can render them as "data frame" with is a fancy 2D array with labeled columns (Technically it's a Pandas data frame (https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.html)):

In [ ]:
```
results.as_df()
```

These numbers don't mean much to you at the moment, but, for instance, `ET` is the execution time. So we can see that running `hello_world` didn't take very long when `k` is 4. It's so short, in fact, that doubt measurement is accurate. We need at least 0.5 seconds of execution for an accurate measurement.

We can also run `hello_world()` with different parameters and different compilation options. We can pass a list of values to `arg_map()` that vastly increase `k` , and we'll use `hello_world_twice` which holds two different compiled versions of our function.

In [ ]:
```
results = run(hello_world_twice, function="hello_world", arguments=arg_ma
```

Here, `cfiddle` ran `hello_world()` a total of 8 times for the 8 combinations of compiler flags and values for `k` : Four times without optimizations with `k == 400000000, 800000000, 1600000000` and four times with optimizations. And we the results to prove it:

In [ ]:
```
results.as_df()
```

Question 4 (Correctness - 2pts)

How much speedup does `-O4` provide compared to `-O1` for the smallest and largest value of `k` ?

Question 5 (Completeness)

Add another value of `k` to the cell above that is 2x the highest value. Re-run the experiment and analyze the speedup. What happened? Why (give your best idea)?

9.3 Local Execution

If you're debugging on running simple experiments you can run code locally. It'll be faster but performance measurements will be pretty meaningless because the datahub machines are shared across many users.:

In [ ]:
```
with local_execution():
    results = run(hello_world_twice, function="hello_world", arguments=ar
```

10 Analyzing Program Behavior with the Performance Equation

Keep track of questions you have. This lab is more about collecting questions than finding answers. I've called out some interesting questions throughout the lab. The last question of the lab asks for *other* questions you had while examining the data you'll collect below, so keep track of them as you work through the lab.

10.1 Meet the Code!

In this lab we are going to analyze a few simple funtions in `microbench.cpp` . The first is `baseline_int()` :

In [23]:
```
microbench = build("microbench.cpp")
microbench[0].source("baseline_int")
```

```
100%                                        1/1 [00:00<00:00, 52.50it/s]
```

Out[23]:
```
extern "C" uint64_t * baseline_int(unsigned long int size, uint reps) {
        uint64_t * array = new uint64_t[size];
        for(uint i = 0; i < size; i++) {
                array[i] = 0;
        }
        start_measurement();
        for(uint r = 0 ; r < reps; r++) {
                for (uint j = 0; j < 3; j++) {
                        for(uint i= 1 ; i < size; i++) {
                                array[i] += i/(1+j)+array[i - 1];
                        }
                }
        }
        end_measurement();
        return array;
}
```

`baseline_int()` initilizes `array` and then does some multiplies and additions to update it's contents. It's not a useful computation, so don't spend time trying to figure out what it does. The `reps` argument controls how many times we'll run it.

10.2 Measuring The Performance Equation

Now that we know how to take measurements, we can try to understand `baseline_int()` 's performance. We will do this using the performance equation:

```
ET = IC * CPI * CT
```

So, we'll need to measure `IC` (instruction count), `CPI` (cycles per instruction), `CT` (cycle time), and `ET`, and we'll do that using "performance counters". Performance counters are the CPU provides to count events that occur in the processor.

`cfiddle` has support for for reading performance counters built in, we just need to tell it to collect data. We can do that passing some extra options to `run()`. Here's what they mean:

1. Passing `run_options` to `run()` lets us control the execution environment for the program. `MHz` set's the clock speed.
2. `perf_counters` controls which performance counters we collect.

In this case, `PERF_COUNT_HW_INSTRUCTIONS`, and `PERF_COUNT_HW_CPU_CYCLES` count the number of instructions executed and the number of CPU cycles, respectively. There are hundreds of other events we can count. We'll use some of them later in the class.

In [ ]:

```
r = run(microbench,
    function="baseline_int",
    arguments=arg_map(size=[1024*1024], reps = 25),
    run_options=arg_map(MHz=[3500]),
    perf_counters=["PERF_COUNT_HW_INSTRUCTIONS", "PERF_COUNT_HW_CPU_CYCLE
```

Here's the raw output

In [ ]:

```
display(r.as_df())
```

The key fields are `ET` (execution time) and the results of the performance counter measurements: `PERF_COUNT_HW_INSTRUCTIONS` and `PERF_COUNT_HW_CPU_CYCLES`.

Question 6 (Correctness - 4pts)

Using the value names in the table above, give expressions for `IC`, `CT`, and `CPI`. (You cannot use `ET` in your expressions.)

- `IC =`
- `CPI =`
- `CT =`

Then we'll do some calculations on the output and print the nicely. `PE_calc()` applies the same math you figured out for the question (assuming you got it right) above to the results. We're going to use this many times throughout the course.

In [ ]:
```
PE_calc(r.as_df())
```

We are going to see a lot of data like this so, let's be clear about what they mean:

function	size	requestedMHz	realMHz	cycles	IC	CPI	CT	ET
function	size of `array`	MHz requested	Measured MHZ	Processor Cycles	dynamic instructions executed	Cycles/instruction	Cycle Time	Execution Time

Note that `MHz` and `realMHz` don't quite match. This can be due to noise in measurements or the processor running faster than we requested.

10.3 Meet Your Processor

Let's gather a little bit of information about the CPU we'll be using. We can just ask the OS, but we need to ask it on our servers in the cloud.

To do that we'll need to run a command line program remotely. For that we'll use `cse142 job run`. It runs a command line remotely (this is the same mechanism that cfiddle uses internally). The `--force` option cancels any jobs you currently have running, and `--take NOTHING` keeps it from copying any local files to the server. `lscpu` is the command to run.

In [ ]:
```
!cse142 job run --force --take NOTHING lscpu
```

As you can see it's a E-1246G CPU running at 3.5GHz. The model number probably doesn't mean anything to you, but that's what google is for (https://ark.intel.com/content/www/us/en/ark/products/134866/intel-xeon-e-2146g-processor-12m-cache-up-to-4-50-ghz.html).

Question 7 (Correctness - 2pts)

Based on the program output and link above, fill out the table below. Some of the information is in the output above, some you'll need to google for. "Technology node" is roughly the size of the smallest transistors used in designing the chip.:

Parameter	Value
How many physical cores?	[YOUR ANSWER HERE]
How many threads?	
Base processor frequency	
Max turbo boost frequency	
Process technology node (nm)	

Parameter	Value
L1 data cache size (d)	
L1 instruction cache size (i)	
L2 Cache size	
L3 Cache size	

Question 8 (Optional)

Here's some other interesting questions to look into about our processor:

1. What's the Intel code name for our processor's microarchitecture? When was it introduced?
2. What's the maximum clock rate at which processors with this microarchitecture can run?
3. What's the most cores available on a single die with this microarchitecture?
4. What major revision of Intel's microarchitecture is it a part of? How old is this basic design?
5. What do all the things under `Flags` mean?

Here are some resources:

1. https://en.wikipedia.org/wiki/List_of_Intel_CPU_microarchitectures (https://en.wikipedia.org/wiki/List_of_Intel_CPU_microarchitectures)
2. https://en.wikichip.org/wiki/WikiChip (https://en.wikichip.org/wiki/WikiChip)

10.4 Instruction Count

Let's see how changing the instruction count (`IC`) affects performance. There are two ways we can increase instruction count for `baseline_int()` :

1. We can increase the number of times the outer loop runs with `reps` .
2. We can increase the size of `array` with the `size` parameter.

10.4.1 Running The Experiment Mutiple Times

Let's run the `baseline_int()` with three values for `reps` : 25, 50, and 100.

Answer the question *before* you look at the results. The goal of this question (and many more to follow) is for you to predict the answer and then see if the results match your intuition. Don't be discouraged if you frequently get the prediction wrong: They are intentionally challenging and some of them have intentionally non-intuitive results. Also, a major goal of the lab is to highlight behavior that seems non-intuitive so you can improve your intuition.

Kick off the experiment below and the answer this question:

Question 9 (Completeness)

In a moment, you'll see four graphs that show how each term of the performance equation changes as we increase `reps`.

What *shape* do you think each curve will have (linear? curved?) and what *direction* will it go (increasing? Decreasing? flat?)? For each term, predict the ratio between it's value at 100 and 25 (i.e., `value_at_100/value_at_25`).

	IC	CPI	CT	ET
Shape	[PUT YOUR ANSWERS IN THE TABLE]			
Direction				
100 vs 25 ratio				

In [ ]:
```
#This takes a while...
d = run(microbench, function="baseline_int",
        arguments=arg_map(size=1024*1024, reps=[25,50,100]),
        run_options=arg_map(MHz=3500),
        perf_counters=["PERF_COUNT_HW_INSTRUCTIONS", "PERF_COUNT_HW_CPU_C
results = PE_calc(d.as_df())
display(results)
```

In [ ]:
```
plotPE(df=results, lines=True, what=[ ('reps', "IC"), ("reps", "CPI"), ("
```

Question 10 (Correctness - 4pts)

Use the "scratch pad" (little triangle thing in the lower-right of this window) or a calculator to compute the actual ratio of the values at `reps = 100` and `reps = 25` and enter them in the table below. How does these results differ from what you expected? (We won't check that you actually used the scratch pad, but you should. It's a very useful tool.)

	IC	CPI	CT	ET
100 vs 25 reps ratio				
Differences compared to your expectations (if any):				

10.4.2 Increasing the Size of `array`

Instead of increasing `reps` we can increase the size of `array` so we'll run the same computation over more data. We will vary the size over a very large range: 10,000 to 20,480,000.

Kick off the cell below to collect the data. While it's running answer this question:

Question 11 (Completeness)

This time, the four graphs will show each term of the performance equation changes as we increase `--size`. What *shape* do you think each curve will have (linear? curved?) and what *direction* will it go (increasing? Decreasing? flat?)? For each term, predict the ratio between it's value at 5,120,000 and its value at 320,000.

	IC	CPI	CT	ET
Shape				
Direction				
5,120,000 vs 320,000 ratio				

In [ ]:
```
#This takes a while...
d = run(microbench, function="baseline_int",
        # this call to exp_range generates a sequence, a, where a_0 = 100
        arguments=arg_map(size=exp_range(10000, 20480000, 2), reps=20),
        run_options=arg_map(MHz=3500),
        perf_counters=["PERF_COUNT_HW_INSTRUCTIONS", "PERF_COUNT_HW_CPU_C
```

In [ ]:
```
df = PE_calc(d.as_df())
display(df)
plotPE(df=df, lines=True, what=[ ('size', "IC"), ("size", "CPI"), ("size"
```

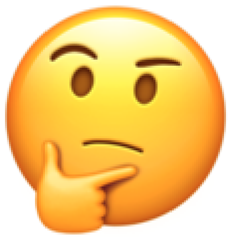
Question 12 (Correctness - 4pts)

Compute the actual ratio of the values at 5,120,000 and 320,000. How does these results differ from what you expected? What is the speedup for a an array of 5,120,000 vs an array of 320,000?

	IC	CPI	CT	ET
5,120,000 vs 320,000 ratio				
Differences compared to your expectations				

Speedup for 5,120,000 vs 320,000-element arrays:

Interesting Question: Why does increasing the size of `array` change CPI? And why does this change occur so quickly and at only over a certain range of sizes?



10.5 Cycle Time

Next, we'll take a look at how clock rate affects performance. Before we do, though, let's see what our options are for clock rate on our machine. We'll `cpupower` for this, a common Linux utility. Again, we'll run it with `cse142 job run` to run it in the cloud:

In [ ]:
```
!cse142 job run --force 'cpupower frequency-info -n'
```

As you can see, the processors in our target systems can run between 800MHz and 3501MHz at mostly 200MHz increments.

Let's see how that affects things by plotting execution time as a function of clock speed (we are skipping 3501MHz for the moment. We'll come back to it.). The readings for the current clock speed may vary from run to run. It just ends up at whatever the last experiment left it at.

Kick off the cell below to collect the data. While it's running answer this question:

Question 13 (Completeness)

We are going to plot four graphs that show how each term of the performance equation changes as we increase clock rate. What *shape* do you think each curve will have (linear? curved?) and what *direction* will it go (increasing? Decreasing? flat?)? For each term, predict the ratio between its value at 3500MHz and its value at 1800Mhz.

	IC	CPI	CT	ET
Shape				
Direction				
3500MHz/1800Mhz ratio				

In [ ]:
```
#This takes a while...
ct_data = run(microbench, function="baseline_int",
              arguments=arg_map(size=1024*1024*32, reps=1),
              run_options=arg_map(MHz=[800, 1000, 1200, 1400, 1600, 1800, 2(
              perf_counters=["PERF_COUNT_HW_INSTRUCTIONS", "PERF_COUNT_HW_CI
```

In [ ]:
```

display(ct_data.as_df())
ct_df = PE_calc(ct_data.as_df())
display(ct_df)
plotPE(df=ct_df, lines=True, what=[ ('requestedMHz', "IC"), ("requestedMH:
```

Question 14 (Correctness - 4pts)

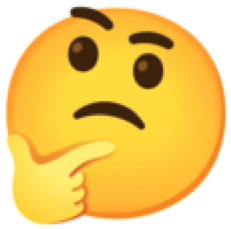
Compute the actual ratio of the values at 3500MHz and 1800MHZ. How do these results differ from what you expected? How much speedup does double doubling the clock rate provide?

	IC	CPI	CT	ET
3500MHz/1800Mhz ratio				

	IC	CPI	CT	ET
Differences compared to your expectations:				

How much speedup does doubling the clock rate from 1800MHz to 3500MHz provide? (show your work):

Interesting question: How can clock rate affect `CPI` ?



The data above had `size = 1024*1024*32` and `reps = 1` . Let's call that "1-big". Note that 1-big, `IC` did not change significantly with clock speed (although, why did it change at all?).

Consider another configuration called "32-small" with `size = 1024*1024` and `reps = 32` . Kick off the cell below and answer this question:

Question 15 (Completeness)

How will 1-big's behavior compare to 32-small's? Fill out the table with your predictions:

Expression	Value
1-big IC / 32-small IC (Your estimate)	
1-big ET @ 3500MHz/ 1-big ET @ 800Mhz (you calculated this in the previous problem)	
32-big ET @ 3500MHz/32-big ET @ 800Mhz (your estimate)	

In [ ]:

```
#This takes a while...
ct_data2 = run(microbench, function="baseline_int",
               arguments=arg_map(size=1024*1024, reps=32),
               run_options=arg_map(MHz=[800, 3500]),
               perf_counters=["PERF_COUNT_HW_INSTRUCTIONS", "PERF_COUNT_HW_C
```

In [ ]:
```
display(ct_data2.as_df())
ct2_df = PE_calc(ct_data2.as_df())
display(ct2_df)
```

Question 16 (Correctness - 1pts)

Fill in the table to see how your predictions did.

Expression	Actual values
1-big IC / 32-small IC	
32-big ET @ 3500MHz/32-big ET @ 800Mhz	

Interesting question: Why doesn't clock rate always affect `CPI` ?"

10.6 Cycles Per Instruction

Unlike `IC` and `CT` we can't set `CPI` directly, but we can adjust the code and see how `CPI` changes. We'll do this in two ways. First, we'll change the data type we are operating on. Then, we'll change the compiler options. Finally, we'll restructure the code.

10.6.1 Floating Point vs Integer Operations

`baseline_double()` (on the left) that is identical to `baseline_int()` (on the right) but uses 64-bit floating point values (of type `double`) point instead of 64-bit integers (`uint64_t`):

In [24]:
```
compare([microbench[0].source(show="baseline_double"), microbench[0].sour
```

```
extern "C" double *baseline_double
(unsigned long int size,  uint rep
s) {
        double * array = new doubl
e[size];
        for(uint i = 0; i < size;
i++) {
                array[i] = 0;
        }

        start_measurement();
        for(uint r = 0 ; r < reps;
r++)
                for (double j = 0;
j < 3; j++) {
                        for(uint i
= 1 ; i < size; i++) {
                                ar
ray[i] += i/(1+j)+array[i - 1];
                        }
                }
        end_measurement();
        return array;
}
```

```
extern "C" uint64_t * baseline_int
(unsigned long int size, uint reps
) {
        uint64_t * array = new uin
t64_t[size];
        for(uint i = 0; i < size;
i++) {
                array[i] = 0;
        }
        start_measurement();
        for(uint r = 0 ; r < reps;
r++) {
                for (uint j = 0; j
< 3; j++) {
                        for(uint i
= 1 ; i < size; i++) {
                                ar
ray[i] += i/(1+j)+array[i - 1];
                        }
                }
        }
        end_measurement();
        return array;
}
```

Kick off the the cell below to run both functions, and answer this question:

Question 17 (Completeness)

How do you think each term in the performance equation will change for `baseline_double()` compared to `baseline_int()` ?

IC:

CPI:

CT:

ET:

In [ ]:

```
cpi_data = run(microbench, function=["baseline_int","baseline_double", "ba
               arguments=arg_map(size=1024*1024, reps=25),
               run_options=arg_map(MHz=3500),
               perf_counters=["PERF_COUNT_HW_INSTRUCTIONS", "PERF_COUNT_HW_
```

In [ ]:

```
cpi_df = PE_calc(cpi_data.as_df())
display(cpi_df)
plotPEBar(df=cpi_df, what=[ ('function', "IC"), ("function", "CPI"), ("fu
```

Question 18 (Completeness)

How did the results for each term in the PE differ from your predictions (if they did)?

IC:

CPI:

CT:

ET:

Question 19 (Optional)

In `microbench.cpp` there are also `baseline_char()` and `baseline_float()` . Add those functions to the experiment above. What do you find?

Interesting question: How and why do the datatypes we use change `IC` and `CPI` ?

Does anyone else think this version of the "thinking" emoji looks like an alien?

10.6.2 The Compiler's Effect

`microbench.cpp` contains the following function:

In [25]:

```
display(microbench[0].source("baseline_int_O4"))
```

```
extern "C" uint64_t *__attribute__ ((optimize(4))) baseline_int_O4 (unsig
ned long int size,  uint reps) {
        uint64_t * array = new uint64_t[size];
        for(uint i = 0; i < size; i++)
                array[i] = 0;

        start_measurement();
        for(uint r = 0 ; r < reps; r++) {
                for (uint j = 0; j < 3; j++) {
                        for(uint i= 1 ; i < size; i++) {
                                array[i] += i/(1+j)+array[i - 1];
                        }

                }
        }
        end_measurement();
        return array;
}
```

It's identical to `baseline_int()` except that for the `__attribute__ ((optimize(4)))` which is a little bit of `gcc` magic to optimize this functions as much as it can (it's the equivalent of passing `-O4` on the command line but just for this function).

Let's see how optimizations affect performance. Kick off the experiment in the cell below and answer this question while it runs:

Question 20 (Completeness)

How do you think each term in the performance equation will change for `baseline_int()` compared to `baseline_int_O4()` ?

IC:

CPI:

CT:

ET:

In [ ]:

```
cpi_opt_data = run(microbench, function=["baseline_int","baseline_int_O4"
                   arguments=arg_map(size=8388608, reps=2),
                   run_options=arg_map(MHz=3500),
                   perf_counters=["PERF_COUNT_HW_INSTRUCTIONS", "PERF_COUNT_HW_
```

In [ ]:

```

cpi_opt_df = PE_calc(cpi_opt_data.as_df())
display(cpi_opt_df)
plotPEBar(df=cpi_opt_df, what=[ ('function', "IC"), ("function", "CPI"),
```

Question 21 (Completeness)

Based on the data above, describe in words what affect the optimizations had on the code and the value of each term of the PE.

IC:

CPI:

CT:

ET:

10.6.3 Code Structure

These two functions increment all the elements in an array by `1.0` , but they do it slightly different ways.

In [26]:
```
compare([microbench[0].source(show="matrix_row_major"),
         microbench[0].source(show="matrix_column_major")])
```

```
extern "C" uint64_t *__attribute__
((optimize(4))) matrix_row_major(u
nsigned long int size, uint reps)
{
        double * array = new doubl
e[size];

        start_measurement();
        for(uint r = 0 ; r < reps;
r++)
                for(uint i= 0; i <
size/ROW_SIZE; i++) {
                        for (int k
= 0; k < ROW_SIZE; k++) {
                                ar
ray[i*ROW_SIZE + k] += 1.0; // Thi
s Line
                        }
                }
        end_measurement();
        return (uint64_t*)array;
}
```

```
extern "C" uint64_t *__attribute_
_ ((optimize(4))) matrix_column_ma
jor(unsigned long int size, uint r
eps) {
#define ROW_SIZE 1024
        double * array = new doubl
e[size];

        start_measurement();

        for(uint r = 0 ; r < reps;
r++)
                for (int k = 0; k
< ROW_SIZE; k++) {
                        for(uint i
= 0 ; i < size/ROW_SIZE; i++) {
                                ar
ray[i*ROW_SIZE + k] += 1.0; // Thi
s Line
                        }
                }
        end_measurement();
        return (uint64_t*)array;
}
```

We'll run both versions and compare their performance. Kick off the experiment below and answer this question:

Question 22 (Completeness)

If `size` is equal to 8,388,608 how many times will "This Line" execute in each function? Do you think one will be faster than the other? Why?

How many times does `This Line` execute in `matrix_row_major()`:

How many times does `This Line` execute in `matrix_column_major()`:

Is there any difference in the "Big-O" running time of these two functions?

Do you think one will be faster than the other? Why?

In [ ]:
```
cpi_order_data = run(microbench, function=["matrix_row_major", "matrix_co
                          arguments=arg_map(size=32*1024*1024, reps=1),
                          run_options=arg_map(MHz=3500),
                          perf_counters=["PERF_COUNT_HW_INSTRUCTIONS", "PERF_C
```

In [ ]:
```
#data_cell
cpi_order_df = PE_calc(cpi_order_data.as_df())
display(cpi_order_df)
plotPEBar(df=cpi_order_df, what=[ ('function', "IC"), ("function", "CPI")
```

Question 23 (Completeness)

Calculate the speedup of `matrix_row_major` over `matrix_column_major`. Why is this result surprising?

Speedup:

Why is this surprising?:

Interesting Question: Why does the order in which the program performs calculations affect `CPI`?

11 Amdahl's Law

Recall from CSE142 that Amdahl's Law limits the speed up an optimization can provide. It's given as

$$S_{tot} = \frac{1}{\left(\frac{x}{S}\right) + (1-x)}$$

Where S is the speedup provided by the optimization, x is the fraction of execution time affected by the optimization, and S_{tot} is total speedup. There is also a more general form of Amdahl's law that covers multiple optimizations operating on two *disjoint* portions of a program:

$$S_{tot} = \frac{1}{\left(\frac{x_1}{S_1}\right) + \left(\frac{x_2}{S_2}\right) + (1-x_1-x_2)}$$

It can be generalized to handle any number of optimizations operating on disjoint portions of the program.

We are going to explore both of these using functions similar to those we studied above. The code is in `amdahl.cpp` . Take a moment to compare them to the code we saw earlier in `microbench.cpp` and understand what's different.

In [27]:

```
amdahl = build("amdahl.cpp")
amdahl[0].source()
```

```
100%                                        1/1 [00:00<00:00, 67.39it/s]
```

Out[27]:

```
#include <cstdlib>
#include <unistd.h>
#include <unordered_set>
#include<algorithm>
#include<cstdint>
#include"cfiddle.hpp"

#define CLINK extern "C"
#define OPT(a) __attribute__(a)

extern "C" void init(uint64_t * array, uint64_t size) {
        for(uint i= 0 ; i < size; i++) {
                array[i] = 0;
        }
}

extern "C" void baseline_int(uint64_t * array, uint64_t size) {
        for (uint j = 0; j < 3; j++) {
                for(uint i= 1 ; i < size; i++) {
                        array[i] += i/(1+j)+array[i - 1];
                }
        }
}

extern "C" void __attribute__ ((optimize(4))) baseline_int_O4 (uint64_t *
array, uint64_t size) {
        for (uint j = 0; j < 3; j++) {
                for(uint i= 1 ; i < size; i++) {
                        array[i] += i/(1+j)+array[i - 1];
                }

        }
}

volatile int ROW_SIZE = 1024;
extern "C" void __attribute__ ((optimize(4))) matrix_column_major(uint64_
t * array, uint64_t size) {


        for (int k = 0; k < ROW_SIZE; k++) {
                for(uint i= 0 ; i < size/ROW_SIZE; i++) {
                        array[i*ROW_SIZE + k] += 1.0; // This Line
                }
        }

}
```

```
extern "C" void __attribute__ ((optimize(4))) matrix_row_major(uint64_t *
array, uint64_t size) {

        for(uint i= 0; i < size/ROW_SIZE; i++) {
                for (int k = 0; k < ROW_SIZE; k++) {
                        array[i*ROW_SIZE + k] += 1.0; // This Line
                }
        }
}
```

The cell below contains the source for a function that calls several of these functions in sequence and has them operate on the same array of data.

`start_measurement()` , `restart_measurement()` , and `end_measurement()` break up the function into sections that we'll measure independently. The argument to `start_measurement()` and `restart_measurement()` are a "tag" to label those phases in the output.

The cell will compile the code and then run it.

In [ ]:
```
amdahl = build(code(r"""
#include"cfiddle.hpp"
extern "C" void init(uint64_t * array, uint64_t size);
extern "C" void baseline_int(uint64_t * array, uint64_t size);
extern "C" void baseline_int_O4 (uint64_t * array, uint64_t size);
extern "C" void matrix_column_major(uint64_t * array, uint64_t size);
extern "C" void matrix_row_major(uint64_t * array, uint64_t size);

extern "C" uint64_t *__attribute__ ((optimize(0))) everything(unsigned lo
    uint64_t * array = new uint64_t[size];

    start_measurement("init phase");

    init(array, size);

    restart_measurement("matrix phase");

    matrix_column_major(array, size);

    restart_measurement("baseline phase");

    baseline_int(array,size);

    end_measurement();

    return array;
}
"""), build_parameters=arg_map(MORE_SRC="amdahl.cpp"))

amdahl_data = run(amdahl, function="everything",
              arguments=arg_map(size=32*1024*1024),
              run_options=arg_map(MHz=3500),
              perf_counters=["PERF_COUNT_HW_INSTRUCTIONS", "PERF_COUNT_HW_
```

In [ ]:
```
amdahl_df = PE_calc(amdahl_data.as_df())
display(amdahl_df)
```

Imagine that you are a manager and your team is tasked with the improving the performance of `everything()` . Members of your team propose two different approaches:

1. Option 1: Replacing `baseline_int()` with `baseline_int_O4()`
2. Option 2: Replacing `matrix_column_major()` with `matrix_row_major()` .

To answer the question below, it'll be helpful to have some of the data from our earlier experiments:

In [ ]:
```
display(cpi_order_df)
display(cpi_opt_df)
```

Question 24 (Completeness)

Based on the data you've collected, calculate the two speedups below and then decide which of the two options will give the best overall speedup for `everything()` (Show your work).

Speedup of `baseline_int_O4()` vs `baseline_int()` :

Speedup of `matrix_row_major()` vs `matrix_column_major()` :

Which of the two options will give the best speedup?

Question 25 (Correctness - 4pts)

Based on the data you collected earlier in this lab for the performance of `baseline_int()` , `baseline_int_O4()` , `matrix_column_major()` , and `matrix_row_major()` , use Amdahl's law to predict the speedup of each approach described below. (Show your work, including the values of `x` and `S` and how you computed them.)

Option 1 (Replacing `baseline_int()` with `baseline_int_O4()`) Speedup:

x =

S = (Hint: you computed this in the previous question)

S_tot =

Option 2 (Replacing `matrix_column_major()` with `matrix_row_major()`) Speedup:

x =

S = (Hint: you computed this in the previous question)

S_tot =

Option 3 (Replace both) Speedup:

This one is more challenging. Show your work.

Modify the code for `everything()` above to implement all three versions (Note: it might be wise to save a copy of the original code). Store the results for each experiments in different variables (e.g., `opt_baseline` instead of `amdahl_data`) and use the cell below to print them out `display()` out. Make it clear either through comments in the code below or the data itself which version is which:

In [ ]:

```
# display(your_data)
```

Question 26 (Correctness - 2pts)

What was the actual speedup for each option? Did Amdahl's law get it right?

Option 1 speedup:

Option 2 speedup:

Option 3 speedup:

Did Amdahl's Law get it right?:

t = build(code(r""" #include "cfiddle.hpp" #include #include #include #include "fastrand.h" #include "winner.hpp"

extern "C" uint64_t winner_winner(const std::vector & A, const std::vector & B, const std::vector & queries) {

```
    uint64_t sum = 0;

    start_measurement();
    for(auto i = queries.begin(); i != queries.end(); i++)  {
        uint64_t a_loc = std::numeric_limits<uint64_t>::max();
        uint64_t b_loc = std::numeric_limits<uint64_t>::max();
        for(uint64_t a = 0; a < A.size(); a++) {
            if (*i == A[a]) {
                a_loc = a;
                break;
            }
        }

        for(uint64_t b = 0; b < B.size(); b++) {
            if (*i == B[b]) {
                b_loc = b;
                break;
            }
        }

        uint64_t min = std::min(b_loc, a_loc);
        if (min != std::numeric_limits<uint64_t>::max()) {
```

// std::cout << min << " *\n"; sum += min; } else { // std::cout << 0 << " *\n"; } } end_measurement(); return sum; }

""", file_name="winner_solution.cpp"), arg_map(OPTIMIZE=["-O4"], MORE_SRC="winner.cpp"))

with cfiddle_config(ExternalCommandRunner_type=CSE142L_ExternalRunnerLocalDelegate): d = run(t, function="run_winners", arguments=arg_map(seed=range(3,20), A_max_size=1000, B_max_size=1000, max_value=1000, queries_max_size=10000000))

display(d.as_df())

Run the cell below to plot the data

12 Programming Assignment: Nibble Search

The labs will all have a programming assignment as part of them. The main purpose of this one is to get you familiar with the tools and the autograding submission process, but hopefully it'll be a little bit entertaining as well.

For the programming assignment, you'll use the cell below. You can edit your code there, run the cell, and it'll compile it, run it, and it tell you how fast it went.

The function you are going to optimize counts the number of times a 4-bit pattern (or nibble (https://en.wikipedia.org/wiki/Nibble) appears in an array of 16-bit numbers. For instance, consider this list of numbers:

```
0x1010
0xaba4
```

Which in binary look like

```
0001 0000 0001 0000
1010 1011 1010 0100
```

And we were looking for this pattern:

```
0100 (0x4)
```

The answer would be 4. It appears twice in the first number and twice in the second. Note that the 4-bit pattern does not have to be aligned to a 4-bit boundary and matches do not span two adjacent words.

The code below contains a simple implementation and the cfiddle code to compile it.

In [ ]:

```
nibble_search = build(code(r"""
#include <vector>
#include <iostream>
#include "cfiddle.hpp"
#include "nibble.hpp"

extern "C" uint64_t nibble_search(uint8_t query, const std::vector<uint16_
    uint64_t count = 0;

    for(auto & i: targets) {
        for(int b = 0; b < 16 - 4; b++) {
            uint8_t extracted = ((i >> b) & (0xf));
            if (extracted == query) {
                count+=1;
            }
        }
    }
    return count;
}
""", file_name="nibble_solution.cpp"), build_parameters=arg_map(OPTIMIZE=
```

There is a new build parameter passed `build()` : `MORE_SRC` lists additional source files to compile. We'll discuss the contents of those files below.

12.1 Running the Code

Rather than invoking this code directly, we will use a testing harness to call it. Here's the code for that:

In [28]:
```
build("nibble.cpp")[0].source()
```

```
100%                                              1/1 [00:00<00:00, 66.73it/s]
```

Out[28]:
```
#include "cfiddle.hpp"
#include<iostream>
#include <vector>


extern "C" uint64_t nibble_search(uint8_t query, const std::vector<uint16
_t> & targets);

void rand_init(std::vector<uint16_t> & v, uint64_t & seed) {
    for(auto &i : v) {
        i = fast_rand(&seed);
    }
}

extern "C" uint64_t run_nibble(uint64_t target_count, uint64_t seed) {

    for(int i = 0; i < 100; i++) {
        fast_rand(&seed);
    }

    std::vector<uint16_t> targets(target_count);

    rand_init(targets, seed);
    uint8_t query = fast_rand(&seed) & 0xf;

    start_measurement();
    auto answer = nibble_search(query, targets);
    end_measurement();

    return answer;
}
```

As you can see, it just initialized a random vector and calls `nibble_search()` . In this assignment, you can modify the implementation of `nibble_search()` but you can't change the harness.

You can run it and display results like so:

In [ ]:
```
d = run(nibble_search, function="run_nibble",
        arguments=arg_map(target_count=200000000, seed=range(1,3)))
display(d.as_df())
```

Note that `run_nibble()` takes two parameters: `target_count` (the number of 16-bit words to search) and `seed` (the random number seed used to generate those 16-bit words). The cell above uses `range()` to generate the array `[1,2]`, so the code runs twice with two different seeds.

Question 27 (Completeness)

Experiment with the value of `seed`. Does it affect the performance of `nibble_search()`? Why or why not?

12.2 The Test Suite

Tests are great: Tests are about the best thing ever (although writing them is a hassle). If you run the tests consistently, you can worry *much* less about correctness. Make small incremental changes to your code, run the tests after each change and enjoy the warm glow of happiness when they pass!

NOTE: You normally will not need to run `run_tests.exe` in the cloud. It'll work fine, but it takes longer which will slow your work down. The test suite is about *correctness* not performance.

The lab provides a comprehensive test suite for your implementation. The code in is `run_test.cpp`. It use GTest (https://github.com/google/googletest) which is a very powerful C++ testing tool from Google. Because of this, the tests run outside of cfiddle.

You can build the tests with:

In [ ]:
```
!make run_tests.exe
```

NOTE: When you build run_tests.exe, it'll use the version of `nibble_solution.cpp` it finds in your lab directotry. The contents of this file will reflect the last time your executed the cell above with the `build()` command in it, so if you make changes there, you'll need to re-run the cell.

Then you can run the tests like this:

In [ ]:
```
!./run_tests.exe
```

The tests cover several different values for `target_count` and bunch of different values for `seed`. When you submit your code for autograding, it will run additional hidden tests (this is to keep students from over-customizing their implementations). The hidden tests are extremely similar to those that `run_tests.exe` includes.

12.2.1 Running the Test Suite

The test suite is meant to help keep you on the right track as you go through the assignment. When you make a change to you code, I would:

1. Run all the tests. If they all pass, great!
2. If some fail, run `simple_tests` first.
3. Once I find a particular test case that fails, you can run and debug just that test (see below).

WHen a test fails, `run_tests.exe` will print out a description of the test that failed:

```
[ RUN      ] simple_tests/NibbleTestFixtureSmall.simple_tests/0
1
run_tests.cpp:23: Failure
Expected equality of these values:
  computed
    Which is: 2
  answer
    Which is: 1
nibble_search(15, [15]) == 1 but your code produced 2
```

The last line shows you what parameters caused the failure and what the right answer is.

At the end of the `run_tests.exe` output, there's a summary of what failed:

```
[----------] Global test environment tear-down
[==========] 13 tests from 4 test suites ran. (400 ms total)
[  PASSED  ] 0 tests.
[  FAILED  ] 13 tests, listed below:
[  FAILED  ] simple_tests/NibbleTestFixtureSmall.simple_tests/0, wh
ere GetParam() = (15, '\xF' (15), 1)
[  FAILED  ] simple_tests/NibbleTestFixtureSmall.simple_tests/1, wh
ere GetParam() = (15, '\a' (7), 1)
[  FAILED  ] simple_tests/NibbleTestFixtureSmall.simple_tests/2, wh
ere GetParam() = (4369, '\x1' (1), 3)
[  FAILED  ] simple_tests/NibbleTestFixtureSmall.simple_tests/3, wh
ere GetParam() = (4369, '\x4' (4), 3)
[  FAILED  ] simple_tests/NibbleTestFixtureSmallDouble.simple_test
s/0, where GetParam() = (15, 15, 15, 2)
[  FAILED  ] simple_tests/NibbleTestFixtureSmallDouble.simple_test
s/1, where GetParam() = (1, 4369, 1, 4)
[  FAILED  ] VisibleTests/NibbleTestFixtureBig.NibbleTest/0, where
 GetParam() = (10000, 1, 7471)
[  FAILED  ] VisibleTests/NibbleTestFixtureBig.NibbleTest/1, where
 GetParam() = (100000, 2, 75432)
[  FAILED  ] HiddenTests/NibbleTestFixtureBig.NibbleTest/0, where G
etParam() = (100, 1, 77)
[  FAILED  ] HiddenTests/NibbleTestFixtureBig.NibbleTest/1, where G
etParam() = (100, 2, 66)
[  FAILED  ] HiddenTests/NibbleTestFixtureBig.NibbleTest/2, where G
etParam() = (100, 3, 75)
[  FAILED  ] HiddenTests/NibbleTestFixtureBig.NibbleTest/3, where G
etParam() = (100, 4, 73)
[  FAILED  ] HiddenTests/NibbleTestFixtureBig.NibbleTest/4, where G
etParam() = (10000000, 3, 7497827)

13 FAILED TESTS
```

Each test has a name (e.g., `simple_tests/NibbleTestFixtureSmall.simple_tests/0`). To run just `simple_tests` you can do:

In [ ]:
```
!./run_tests.exe --gtest_filter=simple_tests*
```

To run just one test you can do something like:

In [ ]:
```
!./run_tests.exe --gtest_filter=simple_tests/NibbleTestFixtureSmall.simpl
```

You can also get a list of all the tests with

In [ ]:
```
!./run_tests.exe --gtest_list_tests
```

12.2.2 Debugging With the Test Suite

If you want to debug your test, open a shell and do

```
$ gdb run_tests.exe
GNU gdb (Ubuntu 12.0.90-0ubuntu1) 12.0.90
Copyright (C) 2022 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/license
s/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<https://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
    <http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from run_tests.exe...
```

Then set a breakpoint on `nibble_search` :

```
(gdb) b nibble_search
Breakpoint 1 at 0x2f922: file nibble_solution.cpp, line 9.
```

and then run the executable with `--gtest_filter` to run the test you want to debug:

```
(gdb) run --gtest_filter=simple_tests/NibbleTestFixtureSmall.simple
_tests/0
Starting program: /cse142L/labs/CSE141pp-Lab-The-PE/run_tests.exe -
-gtest_filter=simple_tests/NibbleTestFixtureSmall.simple_tests/0
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_d
b.so.1".
Note: Google Test filter = simple_tests/NibbleTestFixtureSmall.simp
le_tests/0
[==========] Running 1 test from 1 test suite.
[----------] Global test environment set-up.
[----------] 1 test from simple_tests/NibbleTestFixtureSmall
[ RUN      ] simple_tests/NibbleTestFixtureSmall.simple_tests/0
1

Breakpoint 1, nibble_search (query=15 '\017', targets=std::vector o
f length 1, capacity 1 = {...}) at nibble_solution.cpp:9
9    extern "C" uint64_t nibble_search(uint8_t query, const std::ve
ctor<uint16_t> & targets) {
(gdb)
```

And debug away!

12.3 Managing Multiple Versions

As you work on our code, you can keep multiple versions around in separate cells. For instance, here's another version of `nibble_search()` that searches backwards:

In [ ]:
```
nibble_search_backwards = build(code(r"""
#include <vector>
#include <iostream>
#include "cfiddle.hpp"
#include "nibble.hpp"

extern "C" uint64_t nibble_search(uint8_t query, const std::vector<uint16_
    uint64_t count = 0;

    for(auto & i: targets) {
        for(int b = 16-4-1; b >= 0; b--) {
            uint8_t extracted = ((i >> b) & (0xf));
            if (extracted == query) {
                count+=2;
            }
        }
    }
    return count;
}
""", file_name="nibble_solution.cpp"), build_parameters=arg_map(OPTIMIZE=
```

Note the `TAG="reverse"` in the last line. This is not necessary, but it is handy to add as we will see.

You can then run it like so:

In [ ]:
```
d = run(nibble_search_backwards, function="run_nibble",
        arguments=arg_map(target_count=200000000, seed=range(1,3)))
display(d.as_df())
```

And you can run both the original version and your new version like this:

In [ ]:
```
d = run(nibble_search + nibble_search_backwards, function="run_nibble",
        arguments=arg_map(target_count=200000000, seed=range(1,3)))
```

In [ ]:
```
display(d.as_df())
PE_calc(d.as_df())
```

This works because `nibble_search` and `nibble_search_backwards` are both lists of compiled versions of the code, so we can concatenate them with `+`.

In [ ]:
```
PE_calc(d.as_df())
```

As you can see the value for `TAG` shows up in the output, so you can tell the two version apart.

Question 28 (Optional)

How does the performance of the `reverse` version and the original version compare? What's going on there?

12.4 Final Measurement and Grading

When you are done, make sure your best solution is in `nibble_solution.cpp` . Then you can submit your code to the Gradescope autograder. It will run the commands given above and use the `ET` values from `autograde.csv` to assign your grade.

Your grade is based on your speed up relative to the original version of `nibble_solution.cpp` in the lab. The target speedup is X. Your score is `your_speedup/target_speedup` .

You don't get extra credit for beating the target.

To get points, your code must also be correct. The autograder will run the regressions in `run_tests.exe` to check it's correctness.

You can mimic exactly what the autograder will do with the command below. You can run the cell below to list them and the target speedups.

After you run it, the results will be in `autograde/autograde.csv` rather than `./autograde.csv` . This command builds and runs your code in a more controlled way by doing the following:

1. Ignores all the files in your repo except `nibble_solution.hpp` .
2. Copies those files into a clean clone of the starter repo.
3. Builds and runs `run_tests.exe` with the hidden tests enabled.
4. Runs your code using `run_bench.py` .
5. It then runs the `autograde.py` script to compute your grade.

Running the cell below does just what the Gradescope autograder does. And the cell below shows the name and target speedups for each benchmark. This takes 1-2 minutes to run.

Only Gradescope Counts The scores produced here **do not** count. Only gradescope counts. The results here should match what Gradescope does, but I would test your solution on Gradescope well-ahead of the deadline to ensure your code is working like you expect.

In [ ]:
```
!cse142 job run --take nibble_solution.cpp --lab intro-bench --force autog
```

In [ ]:
```
render_csv("autograde/autograde.csv")
```

And run the autograder

In [ ]:
```
!./autograde.py --submission autograde --results autograde.json
from autograde import compute_all_scores
df = compute_all_scores(dir="autograde")
display(df)
print(f"total points: {round(sum(df['score']), 2)}")
```

The "capped_score" column contains the number of points you'll receive.

And see the autograder's output like this:

In [ ]:
```
render_code("autograde.json")
```

Most of it is internal stuff that gradscope needs, but the key parts are the `score` , `max_score` , and `output` fields.

All that's left is commit your code:

In [ ]:
```
!git add nibble_solution.cpp
!git commit -m "Yay! I finished the first lab!"
!git push
```

If this asks you for a password, you'll need to interrupt your Jupyter notebook kernel and do it in a shell instead.

If `git commit` tell you something like:

```
*** Please tell me who you are.

Run

git config --global user.email "you@example.com"
git config --global user.name "Your Name"

to set your account's default identity.
Omit --global to set the identity only in this repository.

fatal: unable to auto-detect email address (got 'prcheng@dsmlp-jupy
ter-prcheng.(none)')
Warning: Permanently added the RSA host key for IP address '140.82.
112.3' to the list of known hosts.
Everything up-to-date
```

Then, run these after filling in your @ucsd.edu email and your name.

In [ ]:
```
!git config --global user.email "you@example.com"
!git config --global user.name "Your Name"
```

13 Recap

This lab has collected real data to understand the how the performance equation, the power equation, and Amdahl's law apply to some simple programs. This exploration presented the following questions:

- Why does increasing the size of array change `CPI` ? And why does this change occur so quickly?
- How can clock rate affect `CPI` ?
- How and why do the datatypes we use change `IC` and `CPI` ?
- Why does the order in which the program performs calculations affect `CPI` ?

Throughout the rest of the course and the labs, we'll find answers to most of these and see how can use those answers to make better use of modern processors.

Question 29 (Completeness)

Which of these questions do you find most interesting and why?

Question 30 (Completeness)

Give three other questions you have after completing this lab.

1. question 1
2. question 2
3. question 3

14 Turning In the Lab

For each lab, there are two different assignments on gradescope:

1. The lab notebook.
2. The programming assignment.

There's also a pre-lab reading quiz on Canvas and a post-lab survey which is embedded below.

14.1 Reading Quiz

The reading quiz is an online assignment on Canvas. It's due before the class when we will assign the lab.

14.2 The Note Book

You need to turn in your lab notebook and your programming assignment separately.

After you complete the lab, you will turn it in by creating a version of the notebook that only contains your answers and then printing that to a pdf.

Step 1: Save your workbook!!!

In [ ]:
```
!for i in 1 2 3 4 5; do echo Save your notebook!; sleep 1; done
```

Step 2: Run this command:

In [ ]:
```
!turnin-lab Lab.ipynb
!ls -lh Lab.turnin.ipynb
```

The date in the above file listing should show that you just created `Lab.turnin.ipynb`

Step 3: Click on this link to open it: ./Lab.turnin.ipynb (./Lab.turnin.ipynb)

Step 4: Hide the table of contents by clicking the

Step 5: Select "Print" from *your browser*'s "file" menu. Print directly to a PDF.

Step 6: Make sure all your answers are visible and not cut off the side of the page.

Step 7: Turn in that PDF via gradescope.

Print Carefully It's important that you print directly to a PDF. In particular, you should *not* do any of the following:

1. **Do not** select "Print Preview" and then print that. (Remarkably, this is not the same as printing directly, so it's not clear what it is a preview of)
2. **Do not** select `Download as-> PDF via LaTex. It generates nothing useful.

Once you have your PDF, you can submit it via gradescope. In gradescope, you'll need to show us where all your answers are. Please do this carefully, if we can't find your answer, we can't grade it.

14.3 The Programming Assignment

You'll turn in your programming assignment by providing gradescope with your github repo. It'll run the autograder and return the results.

14.4 Lab Survey

Please fill out this survey when you've finished the lab. You can only submit once. Be sure to press "submit", your answers won't be saved in the notebook.

In [29]:

```
from IPython.display import IFrame
IFrame('https://docs.google.com/forms/d/e/1FAIpQLSdEyaIDy52FLLUzQEXoJJmz7
```

Out[29]:

CSE142L Lab Survey

swanson@eng.ucsd.edu Switch account

* Required

Email *

Your email

Student ID (to get credit) *

Your answer

Student ID (to confirm) *

Your answer

Which lab is this? *

Choose

How hard did you think this lab was? *

	1	2	3	4	5	
Very easy	○	○	○	○	○	Ve

How much did you learn from this lab? *

	1	2	3	4	5	
Very little	○	○	○	○	○	A gre

How interesting was this lab? *

	1	2	3	4	5	
Very boring	○	○	○	○	○	Very inte

How many hours did you spend on this lab? *

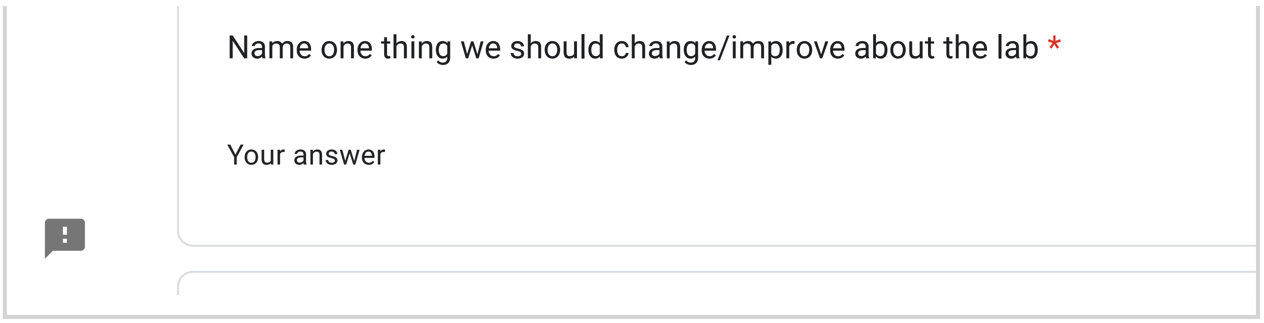
Your answer

Name one thing you liked about the lab *

Your answer

Name another thing you liked about the lab *

Your answer



Name one thing we should change/improve about the lab *

Your answer