Week 2: Lab 1

Lab 1

A warning about GenAI: Many of the problems given in the first few labs are classical problems with well known solutions. They are well known because they are useful learning problems. This means tools like ChatGPT and Copilot are almost certain to have been trained on similar problems. This means they are easily able to solve many of these problems. While this is useful sometimes, using these solutions will prevent you from learning what the labs are trying to teach you. Even worse, later labs will build on prior labs and increase in novelty and difficulty. Generative AI tools will not be able to solve them, and neither will you if you didn't understand the prior labs. We strongly suggest solving these problems yourself and writing your own code. You are free to use Gen AI, but you should consider it to be the similar to Googling the answer or asking your lab facilitator--only use it if you are stuck and make sure you understand the solution it produces.

We have been studying sorting algorithms during the first few weeks of lectures. This lab will cover some additional concepts related to sorting algorithms and aims to familiarize students with the automatic assessment system we will be using in labs for this unit.

Inversions

The concept of inversions is closely related to sorting. Given an array A we say that the elements at indexes i and j are inverted if A[i] > A[j]. Informally, an inversion is a pair of elements in an array that are out of order. For example, the array [3,2,5,4,1] has the following inversions (3, 2), (5, 4), (3, 1), (2, 1), (5, 1), (4, 1).

Sorting algorithms can be used to count the number of inversions in an array. Insertion sort was covered in lecture 2. The algorithm works by removing inversions one at a time until there are no inversions left in the array. This lab's first exercise is to modify the insertion sort function from the lecture to count the number of inversions in the array. Your function should return the number of inversions instead of the sorted list.

Here is the code from the lecture with some helpful comments added for this lab:

```
def insertion_sort(xs: list):
    # Iterate through prefix lengths
    for l in range(1, len(xs)):
        # Insert xs[l] into sorted prefix xs[0:l]
        for i in range(l, 0, -1):
            # xs[i] is element being inserted
            if xs[i] < xs[i-1]:
                # Wrong way around, swap them
                # Hint, this is an inversion!
                xs[i-1], xs[i] = xs[i], xs[i-1]
            else:
                # xs[i] is in the right spot
                break
```

```
            # xs[0:l+1] is now sorted
        return xs  # Modify this to return the inversion count instead of the sorted list
```

Testing with DOMjudge

Now that you have modified the insertion_sort function to return the number of inversions, let's look at the system we use to evaluate code in this unit. We use DOMjudge, which is a piece of software generally used in programming contests. It is used in this unit because it provides a convenient way to automatically run submitted code on a collection of test cases. Please keep in mind while using DOMjudge that it was designed to host programming contests, so some of the terminology on the website might not make sense. For example, your account will be referred to as a "team", since the system expects you to be a team competing in a contest. The server will also refer to labs as "contests" and we have set up a "contest" for each lab.

DOMjudge can determine if the code is correct—if the code gets the wrong answer on a test case, it is not correct. It can also determine (very roughly) if the code has the right algorithmic complexity. This is done by making sure that the code runs within a time limit—usually a few seconds. If the algorithm has the wrong time complexity it will exceed the time limit. DOMjudge will automatically run your code on a collection of test cases to check it for correctness and time complexity.

Example DOMjudge Problem

To submit our code to DOMjudge, it must be in the right format. DOMjudge expects a program that reads a test case using "standard input" and produces the answer on "standard output". These are generally the same as command line input and output. In Python, the input() function (see https://docs.python.org/3/library/functions.html#input) reads a line from standard input, and print(...) prints a line to standard output. Let's demonstrate this with a simple example.

First you will need to access DOMjudge. Go to the CITS3001 DOMjudge server (http://domjudge.gozz.au/). To create an account, go to "login" at the top right and select "Don't have an account? Register now." at the bottom of the page. When you register, you must pick a Username, please either use your student number or pick some other name if you wish to remain anonymous. **However, please make sure to enter your student number into the "Full name" field so that we can keep track of you!** This is important, because we won't be able to track down your account easily if you need technical help for the labs.

Once you have logged into your account, go to the top right and make sure "lab1" is selected from the drop-down menu. Once you are in the lab 1 "contest", click on Problem "APLUSB" to see the description of our example problem. This should display the PDF describing the problem.

The problem asks you to write a program that reads two numbers and output their sum. Let's look at how to do this.

We can read in a line of input and split it into all the space separated parts as follows.

```
x = input().split()
```

The variable x is now a list of strings. Note that the .split() method splits a string into space separated parts (see https://docs.python.org/3.3/library/stdtypes.html#str.split) So `"this 1 is a test".split()` gives `['this', '1', 'is', 'a', 'test']`.

We can get the two numbers as follows:

```
num1, num2 = int(x[0]), int(x[1])
```

Here we use the int(...) function to convert a string to an integer.

Finally, we can output the answer:
`print(num1+num2)`

This completes our solution. We can now submit our code to DOMjudge. To do this, click the Submit button at the top right of the screen. You can upload your python file. Make sure to select the right problem (aplusb).

Once you have submitted your code, the server will run it on all the test cases. You can download the test cases onto your machine. This can be useful if you are trying to debug errors in your code. Click on "Problemset" at the top left, then click on "samples" for any problem to download a zip of the test cases for that problem. Notice that the Problemset page also shows the time limit for each problem. This is important. Many problems are designed so that solutions with a time complexity that is too high will not run in the time limit. We will see this later in the lab.

On the home page, you should see all your previous submissions. You can click on a submission to see details. These details include the runtime and output of your program on each test case. This can also be useful when debugging your code.

Submitting Inversion Counting

Open "invcountsmall" on DOMjudge. This problem asks you to write a program to count inversions. You can use your modified insertion sort from earlier in the lab. Use input() and print(), write a program to read the input as specified in the problem statement and output the number of inversions. Then, submit your python file the DOMjudge. Make sure you submit to the "invcountsmall" problem.

A Faster Algorithm for Inversion Counting

Next, we tackle the final question in the lab. This is called "invcountlarge". Notice that this problem is the same as "invcountsmall", however the bounds have increased. Specially, N has increased to a maximum of 200,000. In the previous question, we modified insertion sort to get an algorithm that counted inversions one at a time. Recall that insertion sort has a worst-case time complexity of O(N^2). It will be too slow for this new problem. Please try to submit your insertion sort based solution from the previous problem.

You should see a "TIMELIMIT" verdict from DOMjudge. This means the program exceeded the maximum allowed time limit. If you click on the "Problemset" page, you should see that the time limit for invcountlarge is 5 seconds. Since our solution was O(N^2), we expect it to be too slow. As a rule of thumb, a typical CPU can do (very roughly) 10^8 operations a second in Python. For N=200,000, our algorithm is doing (very roughly) 200000^2 operations, which translates to about 400 seconds. This kind of very rough analysis is useful in deciding what time complexity a problem is asking for. In many problems, we will explicitly state what time complexity you are aiming to achieve to help guide you.

To solve invcountlarge, you should aim for a time complexity of O(N log N).

Hint: Can you modify merge sort somehow? Refer to the lecture on sorting that explains merge sort.
(To view the hint, select all the text to highlight it)