MTH5001: Introduction to Computer Programming 2022/23

Final Report Project: "Longest Increasing Subsequences"

IMPORTANT: Start by filling in your 9 digit *student ID number* below. **DO IT NOW**.

Save this Jupyter Notebook with the name **MTH5001_ID.ipynb**, where instead of **ID** you write your *student ID number*.

Please check very carefully that your *student ID number* is accurately typed everywhere. Otherwise your marks might go to the wrong person.

Use the available cells to introduce the code. You can add additional cells if needed. explain your code as much as possible with `# comments`

ID: please replace the text after the colon by your 9 digit student ID number

Instructions:

You must write your answers in this Jupyter Notebook, using either Markdown or Python code as appropriate.

Your code must be **well documented**. As a rough guide, you should aim to include one line of comments for each line of code (but you may include more or fewer comments depending on the situation).

The total number of marks available is 100. Attempt all parts of all questions.

For this project, you are expected to write your code almost entirely 'from scratch', although you are allowed to use some specific packages like `numpy`, `matplotlib.pyplot`, etc.

Submission deadline:

You must submit your work via QMPlus, to the "Final Report Project" assignment in the "Final Report Project" section under the "Assessment" tab.

The submission deadline is **11:55pm on Thursday 4th of May, 2023**. *(May the 4th be with you!)* Late submissions will be penalised according to the School's guidelines (https://qmplus.qmul.ac.uk/pluginfile.php/3467941/mod_resource/content/1/School%20of%20Mathematical%20Sciences%20UG%20Student%20Handbook%20202

Your lecturers will respond to project-related emails until 5:00pm on Tuesday 2 May, 2022, only. You should aim to have your project finished by this time.

Marking of projects:

When writing up projects, good writing style is even more important than in written exams. According to the advice (https://qmplus.qmul.ac.uk/pluginfile.php/3467941/mod_resource/content/1/School%20of%20Mathematical%20Sciences%20UG%20Student%20Handbook%20202 in the student handbook,

> To get full marks in any assessed work (tests or exams) you must normally not only give the right answers but also explain your working clearly and give reasons for your answers by writing legible and grammatically correct English sentences. Mathematics is about logic and reasoned arguments and the only way to present a reasoned and logical argument is by writing about it clearly. Your writing may include numbers and other mathematical symbols, but they are not enough on their own. You should copy the writing style used in good mathematical textbooks, such as those recommended for your modules. **You can expect to lose marks for poor writing (incorrect grammar and spelling) as well as for poor mathematics (incorrect or unclear logic).**

Plagiarism warning:

Your work will be tested for plagiarism, which is an assessment offence, according to the School's policy on Plagiarism (https://qmplus.qmul.ac.uk/pluginfile.php/3467941/mod_resource/content/1/School%20of%20Mathematical%20Sciences%20UG%20Student%20Handbook%202 In particular, while only academic staff will make a judgement on whether plagiarism has occurred in a piece of work, we will use the plagiarism detection software "Turnitin" to help us assess how much of work matches other sources. You will have the opportunity to upload your work, see the Turnitin result, and edit your work once accordingly before finalising your submission.

However, you must use your own words as far as possible (within reason, e.g. you would not be expected to change the wording of a well-known theorem), and you **must** reference (https://qmplus.qmul.ac.uk/pluginfile.php/3467941/mod_resource/content/1/School%20of%20Mathematical%20Sciences%20UG%20Student%20Handbook%202 any sources that you use. You cannot communicate with other students on any part of the project. You should also note that some of the questions are personalised in the sense that you will need to import and manipulate data that will be unique to you (i.e. no other student will have the same data).

Introduction to the Project

Longest increasing subsequences appear in many areas across mathematics, computer science, and physics. This project will deal with algorithms for creating longest increasing subsequences of permutations of a set of n numbers, their graphical representation, and some analysis of their structure with computational and graphical means.

As Python indexing starts with zero, we will also start counting at zero and hence give the mathematical description in terms of the set of numbers $[n] = \{0, 1, \ldots, n-1\}$.

For a given $n \in \mathbb{N}$, let $\pi = (\pi_0, \pi_1, \ldots, \pi_{n-1})$ be a **permutation** of $[n]$, i.e., one of the $n!$ possible orderings of $[n]$.

A sequence $(\pi_{i_0}, \pi_{i_1}, \ldots, \pi_{i_{k-1}})$ is a **subsequence** of the permutation π if $0 \leq i_0 < i_1 < \ldots < i_k < n$.

A subsequence of the permutation π is **increasing** if $\pi_{i_0} < \pi_{i_1} < \ldots < \pi_{i_{k-1}}$.

An increasing subsequence of the permutation π is **longest** if there is no other increasing subsequence with $k' > k$.

For example, the increasing subsequences of the permutation $(0, 2, 1)$ are $()$, (0), (2), (1), $(0, 2)$, and $(0, 1)$. Clearly $(0, 2)$ and $(0, 1)$ are longest, so the length of the longest increasing subsequence of $(0, 2, 1)$ is 2. This example shows also that the longest increasing subsequence is not necessarily unique.

Some code to get you started

It is very easy to find code online to solve this problem. For example, the following function finds a longest increasing subsequence in a sequence of numbers:

```
def find_length_of_longest_increasing_subsequence(pi):
    n=len(pi)
    l=n*[0]
    for i in range(n):
        for j in range(i):
            if pi[j]<pi[i] and l[j]>l[i]:
                l[i]=l[j]
        l[i]=l[i]+1
    return max(l+[0])
```

The same code in a codebox:

In [1]:
```
def find_length_of_longest_increasing_subsequence(pi):
    n=len(pi)
    l=n*[0]
    for i in range(n):
        for j in range(i):
            if pi[j]<pi[i] and l[j]>l[i]:
                l[i]=l[j]
        l[i]=l[i]+1
    return max(l+[0])
```

Applied to the example $(0, 2, 1)$ above we indeed get the right result.

In [2]:
```
pi=[0,2,1]
print(pi,find_length_of_longest_increasing_subsequence(pi))
```

```
[0, 2, 1] 2
```

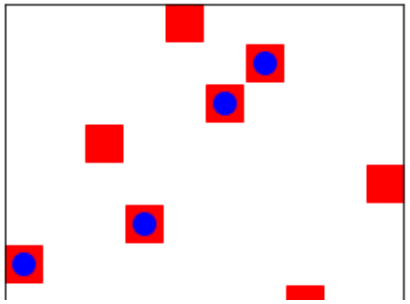
Three obvious cases to check are:

In [3]:
```
pi=[]
print(pi,find_length_of_longest_increasing_subsequence(pi))
pi=list(range(6))
print(pi,find_length_of_longest_increasing_subsequence(pi))
pi=pi[::-1]
print(pi,find_length_of_longest_increasing_subsequence(pi))
```

```
[] 0
[0, 1, 2, 3, 4, 5] 6
[5, 4, 3, 2, 1, 0] 1
```

Plotting Permutations

An easy way to visualise permutations is to consider them as n points (i, π_i) on the $[n]^2$-grid. For example, the permutation $(3, 1, 6, 4, 9, 7, 8, 2, 0, 5)$ is represented by the following picture, with the permutation shown in red and a longest increasing subsequence shown in blue (there are three others, can you find them?).



The structure of the Project

This project consists of **4 parts**. In each of the parts you will need to code up some specific functions, run some code, and respond to some questions. Recall that all code needs to be properly documented with `# comments`, and the explanations in these comments will indeed be assessed and you will receive lots of marks for adequate documentation.

- The **first part** (worth 35 marks) asks you to explain given Python code to compute longest increasing subsequences and analyse its complexity.
- The **second part** (worth 30 marks) asks you to write code to find all longest increasing subsequences.
- The **third part** (worth 20 marks) asks you to analyse longest increasing subsequences for one specifically given permutation.
- The **fourth part** (worth 15 marks) asks you to modify your code for a slight variation on the problem.

The following code box is used to load any necessary modules.

You may not import any other modules, except for the `project` module, which we introduce in Part 3.

In [4]:
```
#DO NOT CHANGE THE CONTENT OF THIS CODE BOX
import matplotlib.pyplot as plt
import numpy as np
from timeit import timeit
from itertools import permutations
```

Part 1: Analysis of given code [35 marks total]

Please remember to write plenty of `# comments` in the code cells. Mark scheme is depicted in each question. 50% of the marks will go to assess the actual code, and 50% of the marks will go to assess the `# comments`. Remember also to **reference** any sources you used. If you used any code or coding ideas you found elsewhere, you are encouraged to do that in the `# comments`

[1.1] [10 marks] Explain in detail how the function `find_length_of_longest_increasing_subsequence` works. More specifically:

- Which data types would be accepted by the function for the argument `pi`?
- What does the function return? What data type is it?
- Explain the purpose of the variable `l`.
- Explain what precisely is being computed in the nested loop involving the parameters `i` and `j`. What is the meaning of `pi[j]<pi[i]` and `l[j]<l[i]`?
- Why did I choose to write `max(l+[0])` and not `max(l)`? *Hint: consider what happens when the function is called with an empty list.*
- Does `find_length_of_longest_increasing_subsequence("computer")` work? Why? What does the output mean?

Type *Markdown* and LaTeX: α^2

In [ ]:
```
1
```

[1.2] [5 marks] Now write a well-documented version of the function `find_length_of_longest_increasing_subsequence`. Add a document string and plenty of comments, but do not change the code itself.

In [ ]:
```
1
```

[1.3] [5 marks] What can you say about the computational complexity of `find_length_of_longest_increasing_subsequence`? Discuss how run time and memory requirement grow as the length n of the permutation increases.

Type *Markdown* and LaTeX: α^2

[1.4] [5 marks] `numpy` has functionality that allows you to generate random permutations, and `timeit` (imported above) allows you to compute the time a function call takes. Verify your answer about the time complexity from the previous question by computing the run time for randomly chosen permutations and a suitable range of lengths and producing an appropriate plot of the run times against the lengths of the permutations.

In [ ]:
```
1
```

[1.5] [5 marks] Now that you understand the code well, here's an alternative function written by some crazy python programmer who has been using recursion, lambda functions, enumerate, and list comprehension.

```
def find_length_of_longest_increasing_subsequence_alternative(pi):
    aux=lambda psi:[] if len(psi)==0 else \
                    (lambda _:_+[1+max([0]+[k for j,k in enumerate(_) if psi[j]<psi[-1]])])(aux(psi[:-1]))
    return 0 if len(pi)==0 else max(aux(pi))
```

Explain how recursion is used here.

Type *Markdown* and LaTeX: α^2

[1.6] [5 marks] Compare the time complexity of `find_length_of_longest_increasing_subsequence` and `find_length_of_longest_increasing_subsequence_alternative`, using appropriate permutation lengths.

In [ ]:
```
1
```

Type *Markdown* and LaTeX: α^2

Part 2: Find all longest increasing subsequences [30 marks total]

Please remember to write plenty of `# comments` in the code cells. Mark scheme is depicted in each question. 50% of the marks will go to assess the actual code, and 50% of the marks will go to assess the `# comments`. Remember also to **reference** any sources you used. If you used any code or coding ideas you found elsewhere, you are encouraged to do that in the `# comments`.

[2.1] [15 marks] The code in Part 1 only gives the length of the longest increasing subsequence for a given permutation. Moreover, we have seen above that the largest increasing subsequence need not be unique.

By using the existing code for `find_length_of_longest_increasing_subsequence(pi)`, **and not otherwise**, write a function `find_all_longest_increasing_subsequences(pi)` that gives out all of them for a given `pi`.

The output should be in the form of a list of lists, i.e. `[sequence1, sequence2, ...]`, where each entry is a longest increasing subsequence.

Test this function on `[]`, `[0,2,1]`, `[0,1,2,3,4,5]`, `[5,4,3,2,1]`, and `"computer"`.

In [ ]:
```
1
```

Note: if you could not complete Q2.1 and write a working function `find_all_longest_increasing_subsequences`, you are allowed to use other code you found to work on subsequent questions, provided you provide a complete reference.

[2.2] [10 marks] Among permutations of fixed size, there are many with several longest increasing subsequences. Find all permutations π of length 10 with the largest number of different longest increasing subsequences. How many permutations π have you found, and how many different longest increasing subsequences do they each have? Print each of these permutations along with their longest increasing subsequences.

You are allowed to use generate permutations using `itertools.permutation` (imported above) as follows.

```
perms=permutations(range(10))
for p in perms:
    ...
    ...
```

In [ ]:
```
1
```

[2.3] [5 marks] As you should now have seen, there are many different questions one could ask about the properties of longest increasing subsequences. We shall finish this part by looking at "typical" lengths for large random permutations.

Generate 10000 random permutations of size 100, compute the lengths of their longest increasing subsequences and display them in a suitable histogram, making sure that each bin corresponds precisely to one single length.

In [ ]:
```
1
```

Part 3: Visualising longest increasing subsequences [20 marks total]

Please remember to write plenty of `# comments` in the code cells. Mark scheme is depicted in each question. 50% of the marks will go to assess the actual code, and 50% of the marks will go to assess the `# comments`. Remember also to **reference** any sources you used. If you used any code or coding ideas you found elsewhere, you are encouraged to do that in the `# comments`.

You have been provided with a Python file called **"project.py"** on QMPlus, which you should **save in the same directory as this Jupyter notebook**. This file contains a function `create_permutation()` which creates a permutation for you, based on your student id. (This will look random, but will be unique to you, i.e. no two students will have the same permutation to work with.) In this section, you will analyse this permutation.

[3.1] [5 marks] Execute the following code cell to create your permutation. You **must** replace the number "123456789" with your 9-digit student number.

Important note. This question essentially gives you 5 free marks for typing your student ID correctly. If you do not do this correctly, you will score zero for this part, and if you instead use another student's ID, then your submission will be reviewed for plagiarism.

In [5]:
```
from project import create_permutation

# Replace "123456789" below with your 9-digit student number

my_permutation=create_permutation(123456789)

# Replace "123456789" above with your 9-digit student number
```

[3.2] [5 marks] As described in the introduction above, a permutation π can be visualised using the points (i, π_i). Plot your permutation (choosing an appropriate marker size!), and describe what you see. How large is your permutation?

In [ ]:
```

```

Type *Markdown* and LaTeX: α^2

[3.3] [5 marks] How long is the longest increasing subsequence? How many are there?

You will notice that it might take a minute or two to compute all those subsequences. Be patient...

Note: if you could not complete Q2.1 and write a working function `find_all_longest_increasing_subsequences` , you are allowed to use other code you found, provided you provide a complete reference.

In [ ]:
```

```

[3.4] [5 marks] Now that you have found all longest increasing subsequences, display these together with the permutation in a figure. What do you notice?

In [ ]:
```
1
```

Type *Markdown* and LaTeX: α^2

Part 4: Longest Increasing Shifted Subsequences [15 marks total]

Please remember to write plenty of `# comments` in the code cells. Mark scheme is depicted in each question. 50% of the marks will go to assess the actual code, and 50% of the marks will go to assess the `# comments`. Remember also to **reference** any sources you used. If you used any code or coding ideas you found elsewhere, you are encouraged to do that in the `# comments`.

We conclude with a modification of our original problem. For any $n \in \mathbb{N}$, let τ_n be the permutation of $[n]$ given by

$$\tau_n = (1,\ 2,\ \ldots,\ n-1,\ 0)\ .$$

Right-multiplying a permutation by τ_n has the effect of shifting the elements left by one place (with the first element being cycled back around to the end). For example, if we have $n = 5$, then

$$(2,\ 4,\ 3,\ 1,\ 0) \cdot \tau_5 = (4,\ 3,\ 1,\ 0,\ 2)\ .$$

Let π be a permutation of length n. We call a sequence a **longest increasing shifted subsequence (LISS)** of π if it is a longest increasing subsequence of all permutations of the form $\pi \cdot \tau_n^m$, where $m \in \mathbb{N}$. Clearly after shifting n times we have come full circle, so $\pi \cdot \tau_n^n = \pi$.

Returning to the example above, we see that $(0,\ 2,\ 4)$ is a LISS of the permutation $(2,\ 4,\ 3,\ 1,\ 0)$, since it is a longest increasing subsequence of $(2,\ 4,\ 3,\ 1,\ 0) \cdot \tau_5^2$.

[4.1] [5 marks] Let `pi` be a permutation of $[n]$, for some $n \in \mathbb{N}$, written in the form of a Python list. Write a function `find_all_LISSes` that accepts such an input `pi`, and returns a list of every LISS of this permutation.

Once this is done, use this function to compute the LISSes of the permutation `[2,4,3,1,0]`, and compare these with its longest increasing subsequences.

Hint: Your function may use the function `find_all_longest_increasing_subsequences` that has already been defined above. There is no need to re-write this code.

Note: if you could not complete Q2.1 and write a working function `find_all_longest_increasing_subsequences`, you are allowed to use other code you found, provided you provide a complete reference.

In [ ]:
```
1
```

You should find that the length of the LISSes of `[2,4,3,1,0]` is three, while the length of the longest increasing subsequences of `[2,4,3,1,0]` is only two. It should be clear from the definition that the length of the LISSes of a permutation will always be longer than the length of its longest increasing subsequences.

Note however, that we if we were to evaluate the LISSes of `[4,3,1,0,2]`, we would get identical results. This is because the LISSes of a permutation are invariant under shifts. The same is not true for longest increasing subsequences, which vary each time the permutation is shifted. Indeed, if we left-shift twice, the set of LISSes and longest increasing subsequences would now coincide.

This means that if we were to consider the ratio of the length of the LISSes with the length of the longest increasing subsequences, we would have reduced it from 1.5 to 1. This brings us to our next question.

[4.2] [5 marks] Take the permutation `my_permutation` that you constructed in Q3.1, and determine how many places you must shift by in order to minimise the length of the longest increasing subsequence.

Once this is done, perform this shift, and label the result `shifted_permutation`.

Note that `shifted_permutation` will have a maximal ratio between the length of its LISSes to the length of its longest increasing subsequences.

In [ ]:
```
1
```

[4.3] [5 marks] Replot the figure from Q3.4 with this new shifted permutation. Your plot should now highlight the LISSes of your permutation, rather than the longest increasing subsequences. Explain how the behaviour of the LISSes differs from the behaviour of the longest increasing subsequences.

In [ ]:
```
1
```

Type *Markdown* and LaTeX: α^2