Detailed Assessment Brief

BSc (Hons) Computer Science

Unit Title: Algorithms and Complexity

Unit Leader: Teo Dannemann teo.dannemann@arts.ac.uk
Programme Administration Contact: cci.ug@arts.ac.uk
Unit code: IU000266
Unit credit: 20 credits
Unit duration: Weeks 1-15
Year / Level: 2/3
Unit briefing date: 15 November 2024

About this brief

This brief gives details of your assignments what make up the assessment for this unit. It is in addition to the Assessments Unit brief on Moodle – please make sure to read that brief for the UAL official marking scheme and important support and assessment information.

This brief describes the one assignment you must complete and upload to Moodle, before the deadline listed. It also outlines the specific assessment criteria for each individual part of the assignment.

Learning outcomes and assessment criteria

On completion of this unit:

LO1 Use formal and informal notations to analyse problems algorithmically and sociotechnically. **(Communication)**
LO2 Identify key decision-making, searching and sorting algorithms and their standard applications in program design **(Enquiry)**
LO3 Use standard algorithms to automate decisions on data, while accounting for effects on marginal populations (Process)
LO4 Make critical decisions about where, when and how to introduce complexity into programs **(Realisation)**

Assessments for this unit

Practical Project (70%): You will individually complete a short project brief by writing software that applies algorithms and techniques introduced in class, perform an analysis of its complexity and communicate that using appropriate notation, and discuss the potential side-effects of your approach on marginalised populations.
The assessment for this unit is weighted. In element-based assessment, you must achieve at least an E grade in each element, and an aggregate grade of at least D- in the overall unit. Failure (F, or F-), or non-submission in any element defaults to Fail for the unit. Assessment will be against the specified marking criteria.

Practical Project Brief

Project Name: Restoring paintings after a museum catastrophe

The *Modern Museum of Fine Arts* had a terrible accident. Their collection was completely ruined by a fire covering their most important collection containing a thousand classical paintings.
The paintings are not completely lost, though, but there are pieces to put together, as they are unrecognisable.
This is what is left after the fire:

- A printed duplicate of each painting. However, the duplicates were mistakenly taken to the shredding machine, so now we only have stripes of papers that have to be sorted to be able to form the image back, as shown in the example below:

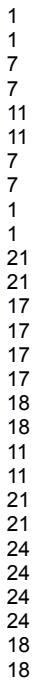

Figure 1: Example of the shredded pieces of duplicates and how they would have to be sorted

- The originals were seriously damaged, and it is difficult to recognise which one is which, so you will use the duplicates to compare and have a say about this.

You, as a hired digital restorer, will be asked to solve these two problems using state-of-the-art efficient algorithms to restore this digital collection of paintings.

Task 1: sorting the shredded duplicates

Luckily, the museum has found a way of helping you sorting the stripes. As you can see in figure 1, each stripe has a tag (a number) that gives its relative position when sorted.
Some duplicates are sorted in a much more random way than others, so in the museum they have grouped duplicates into three groups:

- A series completely shuffled. Ex: [5,1,2,30,11,7] [1,2,5,7,11,30]
- A series slightly shuffled. Ex: [1,2,5,11,7,30] [1,2,5,7,11,30]
- A series shuffled but with order going from [1,…,n] (i.e. without gaps). Ex: [6,2,3,1,4,5] [1,2,3,4,5,6]

You can find each series in the project datasets.

For this part, you are asked for the following:

- (For this part you will only be using the files available in the "Part_1.1" sub-folder).
  For each case, explore the following three sorting algorithms: Merge sort, insertion sort, and quicksort (choosing last element as a pivot).
  Test each algorithm for each of the cases and compare. Illustrate you comparisons with three plots (for a,b, and c). In each plot, x is the size of the lists, and y is the time it takes to sort, so you should have three curves plotted, one for each algorithm.
  Discuss on which algorithm would be best for each case and reflect on why they are outperforming the rest.

- ⭐ Propose an alternative sorting algorithm for case c that would outperform Merge sort, insertion sort, and quicksort.

- Now we will only work with datasets contained in "Part_1.3" sub-folder. In this folder you will find three shuffled arrays with their corresponding shuffled images (in different text files). The task now is to sort not only the lists, but the accompanying shuffled images. Only use insertion sort for this part. Bear in mind that each time you move an element from the order list you also have to move the corresponding stripe's position. Display the final sorted images of the three paintings.

- ⭐⭐ Create an animation of the image being sorted, i.e. a *gif* (or video) file that shows the evolution from the starting unsorted state (left image in Figure 1) to the completely sorted state (right image in Figure 1). Do this only for one painting of three from previous part.

Task 2: Relabelling the originals

For this part, you will only use files available in "Part_2" sub-folder.
Because severely damaged, the curators from the museum are not able to identify the original paintings. You will use the duplicates to compare and check which one is the most likely to match the original. For this you are given the following files:

- *Duplicates.txt* provides a dictionary where each key correspond to a painting

duplicate, with keys going from 1 to 20 (i.e. there are 20 paintings). The corresponding value to that key is a 2D array containing the pixels of the painting. The resolution of this is 20x20 pixels, so a very low resolution version of the painting, as shown in Figure 2. You will be using these 20x20 pixels black-and-white paintings as the duplicates to compare with the originals that have been destroyed.

- *Original.txt* contains one original painting that has been severely damaged. This painting is also in a 20x20 pixels resolution.

Figure 2: Example of the downscaling process for images. You will be working only with images like the one at the right side.

Because severely damaged, the original painting doesn't match perfectly with any of the 20 duplicates referred in point 1. Some pixels of it have been altered with the fire.

You are asked to algorithmically find which one of the 20 duplicates is the best possible match.

For this, you are asked to develop a genetic algorithm for finding the best solution.

- Implement the *createInitialPopulation()* function that creates 500 arrays of 20x20, each one being a copy of the array given in *Original.txt.* (HINT: be careful when making copies of 2D lists in Python. You might need to use the *copy.deepcopy()* function to make actual copies and not just binding objects)
- Implement the *fitness(2D_array)* function that takes a 2D_array as an input and return its fitness. You should compare the 2D_array with each one of the 20 arrays given in *Duplicates.txt* and calculate how similar are them (through some distance measure). Fitness should be optimal when the 2D_array matches perfectly with one of the 20 duplicates.
- Implement *selection(population)* function that should take a population, sort it by fitness, and take the individual with best fitness. Then it should create a new population of same size, but only containing copies of the individual with best fitness.
- Implement *mutation(individual,p),* that loops through the individual (i.e. 2D array) and for each element, it changes its value (from 1 to 0 or from 0 to 1) with probability *p.*

There is no crossover for this Genetic Algorithm.

- Now run the Genetic Algorithm, calling the corresponding functions for each step. Run the algorithm for 100 iterations, and check if the solution converges. For

mutation, use *p=0.05*. Does the original converge to a specific painting of the duplicates?

- ⭐ Discuss around the efficacy of Genetic Algorithms for these cases. Would you recommend using other kind of algorithm instead? Which ones and why?

Deliverables

Submission Format: Single .zip file.

The .zip file must contain:

- A Jupyter notebook containing all the code, separated for each task
- The *gif* or video file required in task 1.4.
- In you Jupyter notebook, you have to add text that explain your code as markdown or as comments to your code, but **it has to be clear to the reader of every step you are taking**. Also, every decision you take for your solution must be justified and explained.
- **IMPORTANT:** You can use your stars ONLY for the tasks that are marked with stars. In case you want to use your stars, you have to add a short text in the beginning of your notebook explaining for which task(s) are you using your stars.