159.271 Computational Thinking for Problem Solving

# Tutorial 6: Greedy algorithm for Magic

One aspect of the game called magic (or jugio) is as follows. You have `n` defence cards. Each of these has a **`worth`** and a **`defence ability`** (two separate integer values per card). Your opponent has the same number `n` of attack cards. Each of these has an **`attack ability`** (single integer value per card). You can see all of the cards. Your task is to define a one-to-one matching between attacking cards and defending cards. That is, you must allocate each of your defence cards to a unique attack card.

If one of your cards has a defence ability lower than the attack ability of your opponent's attack card to which it is allocated, then your defence card dies. Your goal is to **`maximise the sum of the worths of your cards that live`**.

Design a greedy algorithm to solve this problem and then implement it in Python. First of all, write your greedy algorithm down in English and explain why it is a good strategy.

To implement your algorithm in Python you will need the following steps: (1) obtain from the user an input value `n` to determine the number of cards that must be generated (2) generate random values between `1` and `n` for each of the worth and the defence ability of each of your `n` cards and store these pairs of values in a list, (3) generate random values between `1` and `n` for attack ability of each of your opponent's `n` cards and store these single values in a list, (4) use your **`greedy strategy`** to allocate each of your cards to one of your opponent's cards (5) display the two lists of defence and attack cards, the matching that your strategy produces (you can use a list index to identify each card, along with its associated values) and the sum of the worths of your surviving cards.

Remember that the choice made by a greedy algorithm may depend on choices made so far, but not on future choices. It iteratively makes one greedy choice after another, reducing the problem to a smaller one at each step. In other words, a greedy algorithm never reconsiders its choices.

Is your algorithm optimal? That is, does it always produce the maximum possible sum of worths for your cards that live? If you think so, can you give a **`proof`** that your algorithm is optimal? The proof could run along the following lines: If the matching is not optimal, there must be some live card that could be swapped with a dead card of higher value which would then survive, or some dead card that could be swapped with another dead card which would then survive. For each live card, show that your greedy strategy ensures it cannot be replaced in the matching by a dead card of higher value which would then survive. For each dead card, show that your greedy strategy ensures it cannot be replaced in the matching by another dead card which would then survive.

Submit your code as a single `.py` file. Submit the English description of your algorithm and any optimality proof that you produce in a separate text file.