159.271 Computational Thinking for Problem Solving

Tutorial 3: Greedy algorithm for the "Magic" Card Game

Part of the game called Magic (or Jugio) is about your ability to defend against attacks, as follows:

- you have N **defense cards**. Each has a **value** & a ***defense* strength** (both ints)
- your opponent also has N of attack cards. Each has an ***attack* strength** (an int).

You can see all cards. Your task is to pair match attacking cards and defending cards. That is, you must pair each of your defense cards with a single attack card.

If one of your cards has a defense strength lower than the attack strength of your opponent's attack card to which it is allocated, then your defense card dies.

>>> Your goal is to **maximise the sum of the values of your cards that live**.

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

One implement of your algorithm in Python would be to to:

1. define N to determine the number of cards that must be generated
2. generate random values between **1** and N for the value and defense strength of each of your N cards and store these in a list,
3. generate random values between **1** and N for attack strength of each of your opponent's N cards and store these in a list,
4. use a **greedy strategy** to allocate each of your cards to one of your opponent's cards
5. display the two lists of defense 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 values of your surviving cards.

>>> All this would take too much of the two-hour tutorial, so I've created a framework for you.

YOUR TASKS: ***DESIGN AND IMPLEMENT*** **TWO DIFFERENT STRATEGIES** (one GOOD one and one other) to try and improve on the 50% preservation score that results from choosing the attacking and defending cards randomly.

>>> **>>> Download the Magic_Game_Framework from Stream**

To understand how to use the framework, **start by reading the function *play_game()*)** as it include the random strategy code (both the attack and defending cards are chosen randomly) and runs a single simulation.

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

OPTIONAL: Are your strategies optimal?

Is your algorithm optimal? That is, does it always produce the maximum possible sum of values 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 replaced with a dead card of *higher value* which would then survive, or some dead card that could be replaced 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 by another dead card in the matching which would then survive.

Giovanni Moretti
March 20, 2017

A sample run from the framework is on the next page.

With large N (i.e. 10 - 20) & many of runs, preserved value ~ 50%

Another algorithm (not shown) preserves about 80-85%

Can you do better?

Here's the output from a single run of the game framework

Attacking and defending card are chosen randomly

```
>>>> ROUND 1
My cards at the start:
  (value:   1 - defense:   4)
  (value:   4 - defense:   3)
  (value:   3 - defense:   1)
  (value:   3 - defense:   4)
  (value:   4 - defense:   2)
Attacker's cards [1, 2, 4, 1, 3]
My cards: total initial value: 15

Attacker's Strength 2
My remaining cards:
  (value:   1 - defense:   4)
  (value:   4 - defense:   3)
  (value:   3 - defense:   1)
  (value:   3 - defense:   4)
  (value:   4 - defense:   2)
My card   (value:   3 - defense:   1) vs. Attacker's: 2 >>> My card DIES

Attacker's Strength 1
My remaining cards:
  (value:   1 - defense:   4)
  (value:   4 - defense:   3)
  (value:   3 - defense:   4)
  (value:   4 - defense:   2)
My card   (value:   4 - defense:   3) vs. Attacker's: 1 >>> My card LIVES

Attacker's Strength 4
My remaining cards:
  (value:   1 - defense:   4)
  (value:   3 - defense:   4)
  (value:   4 - defense:   2)
My card   (value:   4 - defense:   2) vs. Attacker's: 4 >>> My card DIES

Attacker's Strength 3
My remaining cards:
  (value:   1 - defense:   4)
  (value:   3 - defense:   4)
My card   (value:   3 - defense:   4) vs. Attacker's: 3 >>> My card LIVES

Attacker's Strength 1
My remaining cards:
  (value:   1 - defense:   4)
My card   (value:   1 - defense:   4) vs. Attacker's: 1 >>> My card LIVES

Remaining Live cards
  (value:   4 - defense:   3)
  (value:   3 - defense:   4)
  (value:   1 - defense:   4)
Value Preserved in alive cards = 53%

Average Value Preserved = 53.3%
```