Karnav Popat
karnav.popat_ug24@ashoka.edu.in

November 29, 2021
FCP Quiz 2

Karnav Popat - Exam 2 - Question 1

I agree with the wiki page's summary that it's basically just a restatement of determinism vs free will. I personally believe in the https://en.wikipedia.org/wiki/Predestination_(film) / Harry Potter theory of time travel, so I'd pick only box B, because I'm assuming that the perfect predictor (being perfect) already knows that I'd pick only B and therefore would reward.

Also from a costs/gains perspective, box A is a trivial reward to lose, whereas the potential box B is a huge reward to gain. It makes more sense to me to go for whatever chance I have of getting the bix box B because as long as the odds of determinism being true instead of free will are better than 1:100, the reward for B (1,000,000 * chance of determinism) is better than A+B (1,000 * chance of free will).

As for the consciousness part, it comes back to the same thing: regardless of what the simulation chooses, I have the (illusion of) free will to defy it. If the machine predicts that I will defy what it said, it'll factor that in and tell me the opposite of what it wants me to do. But I have the capability of anticipating that, and therefore defying it by actually agreeing to it. The machine can simulate that… this will go on and on until eventually the machine stack overflows, and therefore I preserve my free will.

Karnav Popat - Exam 2 - Question 2

Even though the machines seem creative, the fact remains that they are not. Even the most well-trained and advanced AI models are just regurgitating the data that's trained into them by simple reinforcement learning. This is analogous to the famous Chinese room experiment, where the conclusion is clear: even though the person who uses the dictionary to translate into Chinese is technically correct, the fact that they don't actually understand Chinese determines that they don't actually speak Chinese.

Similarly, the fact that AI can generate images that are creative or unique doesn't mean that the machines themselves are creative, because they lack the special factor that would make them sentient/creative/understanding.

Parallels can be drawn here to GPT-3: real life applications of it have included using it to write blog articles that fooled large sections of the online community; however, and very crucially, these articles didn't contain anything original, anything that was inspired or out-of-track, it was just a recombination of words which the AI thought sufficiently imitated articles by real life humans. This is exactly what the face-generating AI is doing: ordering pixels in a way that it thinks resembles a human face.

Part (b)

Humans do have feelings, and they are also programmed to act like they do. The two statements are not mutually incompatible. If we're comparing the human body to a machine, it's sufficient to say that it's a very complex machine that's unlike any of our present architectures. At some point of complexity, it ceases to be relevant whether a certain state in a human is caused by a mixture of chemicals (the organic version of the electrical impulses that simulate state in our machines) or by genuine, inherently human non-programmed inspiration. The fact that we can operate with sentience, and ask questions about whether or not we feel feelings, is proof enough that we do feel feelings and that even if we are programmed to behave in certain ways, we have either free will itself, or the illusion of free will which is pragmatically good enough.

If a robot was a sufficiently good actor, it would still not have actual feelings unless it hit the point of singularity where it became sentient and had sufficiently complex internal states to be indistinguishable from reality. However, no robots conceived by human (outside fiction) have even approached this level; all machines known to us at present are simple tools of logic and binary that can't have feelings. Also, acting wouldn't comprise of having feelings no matter how good it is, it would only be relevant if the robot actually did feel those things, which it can't.

Karnav Popat - Exam 2 - Question 3

```
import random


def generate_list(size):

    random_list = [x for x in range(0, size)]
    random.shuffle(random_list)
    print(f"random list: {random_list}")
    return random_list


def odd_even_sort(random_list):
    even_nums = [random_list[x] for x in range(len(random_list)) if x % 2 == 0]
    odd_nums = [random_list[x] for x in range(len(random_list)) if x % 2 == 1]

    even_nums, odd_nums = sorted(even_nums), sorted(odd_nums)

    sorted_evens_list = [item for sublist in list(zip(even_nums, odd_nums)) for
item in sublist]

    return sorted_evens_list


def chunk_sort(random_list, k):
    chunks = []

    for i in range(0, len(random_list)-k, k):
        chunks.append([random_list[i], random_list[i+1], random_list[i+2]])

    if len(random_list) % k != 0:
        chunks.append([*(random_list[i+k:])])

    chunks = [sorted(chunk) for chunk in chunks]
    sorted_chunk_list = [item for sublist in chunks for item in sublist]

    return sorted_chunk_list


def main():
    generated_list = generate_list(10)  # part A

    even_sorted = odd_even_sort(generated_list)
    print(f"even sorted: {even_sorted}")  # part B

    chunk_sorted = chunk_sort(even_sorted, 3)
    print(f"chunk sorted: {chunk_sorted}")  # part C
```

```python
if __name__ == '__main__':
    main()
```

Karnav Popat - Exam 2 - Question 4

Part (a)

For angles that are multiples of 90 degrees, I would use the following (I know the question said code wasn't necessary but it's easier to express with):

```
# turn the array left/right by 90 degrees
def transpose(A):

    B = [[(255, 255, 0) for row in A] for column in A[0]]

    assert(len(B[0]) == len(A))
    assert(len(B) == len(A[0]))

    for row in range(len(A)):  # height-wise
            for column in range(len(A[row])):  # width-wise

                        # A[row][column] rotates 90 left
                        # A[len(A)-row-1][column] rotates 90 right
                        B[column][row] = A[row][column]

    return B


# rotate by 180 degrees
def xflip(A):

    B = [[(255, 255, 0) for column in row] for row in A]

    for row in range(len(A)):  # height-wise
            for column in range(len(A[row])):  # width-wise

                        B[len(A)-row-1][column] = A[row][column]

    return B
```

Essentially I'd iterate over each row in the image array and use it as a column in a new array (for 90) or flip it and use it as the height - ith row (for 180).
I have no idea if it's even possible to represent rotations of non-90-multiple angle rotations in regular arrays and dataframes without a lot of diagonal/trigonometric math. But if I had to, I'd use https://pythontic.com/image-processing/pillow/rotate

Part (b)

Part ©

```
import math
from images_fcp import *

array = readImage("xi.jpg")
width, height = len(array[0]), len(array)  # 2560, 1707

kernel = [
  [0, -1, 0],
  [-1, 5, -1],
  [0, -1, 0]
]


# blur or sharpen A with K
def apply_kernel(A, K):
  B = [[(255, 255, 0) for column in row] for row in A]

  for row in range(len(A)):  # height-wise
     for column in range(len(A[row])):  # width-wise

        B[row][column] = multiply(A, K, row, column, len(A), len(A[row]))

  return B


if __name__ == '__main__':
  # Note that the original lenna.png is square but this one isn't
  q2 = array

  # Subpart 1
  reverse_q2 = [[(255 - (q2[row][column][0]), 255 - (q2[row][column][1]), 255 -
(q2[row][column][2])) for column in
                 range(len(q2[0]))] for row in range(len(q2))]

  # Subpart 2
  detected_reverse_q2 = apply_kernel(reverse_q2, kernel)

  writeImage(reverse_q2, "q2_reverse.jpg")
  writeImage(detected_reverse_q2, "q2_detected.jpg")
```

Part (d)

The results shouldn't change whether or not the colours are inverted. The point of the kernel is to discover unequal rises and falls in the colour values of the boxes it takes (3x3 here), and for two given pixels with say x and x+y values, the difference will still be y whether they stay x and x+y or whether its 255-x and 255-x-y .

Part (e)

```
import math
from images_fcp import *

array = readImage("xi.jpg")
width, height = len(array[0]), len(array)   # 2560, 1707

kernel = [
   [0, -1, 0],
   [-1, 5, -1],
   [0, -1, 0]
]


# blur or sharpen A with K
def apply_kernel(A, K):
  B = [[(255, 255, 0) for column in row] for row in A]

  for row in range(len(A)):   # height-wise
      for column in range(len(A[row])):   # width-wise

          B[row][column] = multiply(A, K, row, column, len(A), len(A[row]))

  return B


# generate colour pattern
def create_pattern(pattern, height, width):
  for row in range(width):
      for col in range(height):
          nb = ((255 - (row + col) * 1.2) / 4) % 255

          pattern[col][row] = (pattern[col][row][0] - nb, pattern[col][row][1] -
nb, nb)

  return pattern


def surround(A, k):
  B = [[(0, 0, 0) for column in range(len(A[0]) + (k))] for row in range(len(A)
+ (k))]
```

```
  for row in range(k * 2, len(A[0]) - k - 1):
      for col in range(k * 2, len(A) - k - 1):
          B[col][row] = A[col][row]

  return B


if __name__ == '__main__':
  # Note that the original lenna.png is square but this one isn't
  q2 = array

  # Subpart 1
  reverse_q2 = [[(255 - (q2[row][column][0]), 255 - (q2[row][column][1]), 255 -
(q2[row][column][2])) for column in
                        range(len(q2[0]))] for row in range(len(q2))]

  # Subpart 2
  detected_reverse_q2 = apply_kernel(reverse_q2, kernel)

  # Subpart 3
  blue = create_pattern(q2, len(q2), len(q2[0]))

  # Subpart 4
  k = 10
  bordered = surround(blue, int(k / 4))

  writeImage(reverse_q2, "q2_reverse.jpg")
  writeImage(detected_reverse_q2, "q2_detected.jpg")
  writeImage(blue, "blue.jpg")
  writeImage(bordered, "bordered.jpg")
```

Part (f)

```
import math
from images_fcp import *

array = readImage("xi.jpg")
width, height = len(array[0]), len(array)  # 2560, 1707

kernel = [
  [0, -1, 0],
  [-1, 5, -1],
  [0, -1, 0]
]
```

```
# blur or sharpen A with K
def apply_kernel(A, K):
  B = [[(255, 255, 0) for column in row] for row in A]

  for row in range(len(A)):  # height-wise
     for column in range(len(A[row])):  # width-wise

        B[row][column] = multiply(A, K, row, column, len(A), len(A[row]))

  return B


# generate colour pattern
def create_pattern(pattern, height, width):
  for row in range(width):
     for col in range(height):
        nb = ((255 - (row + col) * 1.2) / 4) % 255

        pattern[col][row] = (pattern[col][row][0] - nb, pattern[col][row][1] -
nb, nb)

  return pattern


def surround(A, k):
  B = [[(0, 0, 0) for column in range(len(A[0]) + (k))] for row in range(len(A)
+ (k))]

  for row in range(k * 2, len(A[0]) - k - 1):
     for col in range(k * 2, len(A) - k - 1):
        B[col][row] = A[col][row]

  return B


if __name__ == '__main__':
  # Note that the original lenna.png is square but this one isn't
  q2 = array

  # Subpart 1
  reverse_q2 = [[(255 - (q2[row][column][0]), 255 - (q2[row][column][1]), 255 -
(q2[row][column][2])) for column in
                 range(len(q2[0]))] for row in range(len(q2))]

  # Subpart 2
  detected_reverse_q2 = apply_kernel(reverse_q2, kernel)

  # Subpart f
  blue = create_pattern(q2, len(q2), len(q2[0]))
```

```
# Subpart g
k = 10
bordered = surround(blue, int(k / 4))

writeImage(reverse_q2, "q2_reverse.jpg")
writeImage(detected_reverse_q2, "q2_detected.jpg")
writeImage(blue, "blue.jpg")
writeImage(bordered, "bordered.jpg")
```

Karnav Popat - Exam 2 - Question 5

Karnav Popat - Exam 2 - Question 6

Part (a)

I would use the naive approach, because frankly I don't have time to think of a better algorithm.

Make (n * (n-2) / 2) / 2 passes on the array.

On each pass, compare each successive element with the minimum element found so far and the maximum element found so far. Remember the minimum element so far and the maximum element so far.

At the end of every pass you should have one pair of elements. It's trivial to arrange the two in the correct order. The pairs are already in the correct order because of how the list is created.