1 Homework 1

1.1 Exercise demo

Implement Fisher-Yates shuffle algorithm (https://en.wikipedia.org/wiki/Fisher%E2%80%93Yates_shuffle (https://en.wikipedia.org/wiki/Fisher%E2%80%93Yates_shuffle)) and measure how the performance scales as the number of cards being shuffled increases. Create a plot to show the result. You can just use a different integer for each unique card and no need to implement a card class by yourself.

1.2 Solution


For benchmarking I look at input sizes of length $2^0$ to $2^{19}$, testing powers of two. At every experiment size (n), the experiment is run 5 times and then results are averaged.

In [1]:

```python
from random import randint

def card_deck_generator(n: int) -> list:
    
    Generates a deck of cards with n elements
    
    cards = []
    for card in range(n):
        # Add plus 1, since I want the cards to start from 1
        cards.append(f'{card+1}')
    return cards
```

In [2]:

```python
def fisher_yates_shuffle(deck: list) -> list:
    
    Creates a new list that is shuffled using Fisher-Yates shuffle algorithm.
    
    shuffled_deck = deck[:]
    card_deck_length = len(shuffled_deck)
    for i in range(card_deck_length-1, 0, -1):
        j = randint(0, i)
        shuffled_deck[j], shuffled_deck[i] = shuffled_deck[i], shuffled_deck[j]
    return shuffled_deck
```
In [3]:

1 # Quick tests! Make sure to test your code.
2 # card_deck = card_deck_generator(n=100)
3 # fisher_yates_shuffle(card_deck) # Seems to work!

In [4]:

1 from time import monotonic
2 def benchmark_fy(n_ranges: list, n_attempts: int=1) -> list:
3     
4     # Benchmark Fisher-Yates algorithm.
5     n_ranges: Sizes of array to use for experiment
6     n_attempts: how many attempts to do for each n.
7     
8     timings = []
9
10     for n in n_ranges:
11         # We do not want to include generation of data within timing procedure!
12         n_times = []
13         for _ in range(n_attempts):
14             start_deck = card_deck_generator(n)
15             start = monotonic()
16             shuffled_deck = fisher_yates_shuffle(start_deck)
17             end = monotonic()
18             # Time taken (in seconds) is end - start
19             n_times.append(end-start)
20
21         # Add average time taken over n attempts as the result
22         timings.append(sum(n_times)/n_attempts)
23
24     return timings

In [5]:

1 # Generate 2-power input params for benchmark
2 experiment_n = [2**i for i in range(20)]
3
4 benchmark_results = benchmark_fy(
5     n_ranges=experiment_n,
6     n_attempts=5
7 )
```python
In [6]:
import matplotlib.pyplot as plt
plt.rcParams["figure.figsize"] = (10,10)
plt.plot(experiment_n, benchmark_results)
plt.grid(True)
plt.xlabel('n (array size)')
plt.ylabel('time (seconds)')
plt.title('Fisher-Yates time taken vs input array size')
plt.show()
```

![Graph showing Fisher-Yates time taken vs input array size](image)

Fisher-Yates time taken vs input array size

- Time (seconds) increases linearly with the input array size.
- The time taken ranges from 0 to 0.6 seconds.
- The experiment n (array size) ranges from 0 to 500,000.
1.2.1 Interpretation of results

As we can see from the plot above, we have linear scaling between input size and time taken to shuffle it. From the graph above, we can see that it takes approximately 0.2 seconds to shuffle an array of size 200 000 elements and 0.4 seconds to shuffle an array of length 400 000.

From the plot, I would say this algorithm belongs to big-O class of $O(n)$. 