Question 1

Generate random addresses with the following arguments: -s 0 -n 10, -s 1 -n 10, and -s 2 -n 10.
Change the policy from FIFO, to LRU, to OPT.
Compute whether each access in said address traces are hits or misses.

```
-s 0 -n 10 -p FIFO

ARG numaddrs 10
ARG policy FIFO
ARG cachesize 3
ARG seed 0

Access: 8  Miss  [8]
Access: 7  Miss  [8,7]
Access: 4  Miss  [8,7,4]
Access: 2  Miss  [7,4,2]
Access: 5  Miss  [4,2,5]
Access: 4  Hit   [4,2,5]
Access: 7  Miss  [2,5,7]
Access: 3  Miss  [5,7,3]
Access: 4  Miss  [7,3,4]
Access: 5  Miss  [3,4,5]

10% hit rate
```

```
-s 0 -n 10 -p LRU

ARG numaddrs 10
ARG policy LRU
ARG cachesize 3
ARG seed 0

Access: 8  Miss  [8]
Access: 7  Miss  [8,7]
Access: 4  Miss  [8,7,4]
Access: 2  Miss  [7,4,2]
Access: 5  Miss  [4,2,5]
Access: 4  Hit   [2,5,4]
Access: 7  Miss  [5,4,7]
Access: 3  Miss  [4,7,3]
Access: 4  Hit   [7,3,4]
Access: 5  Miss  [3,4,5]

20% hit rate
```

```
-s 0 -n 10 -p OPT

ARG numaddrs 10
ARG policy OPT
ARG cachesize 3
ARG seed 0

Access: 8  Miss  [8]
Access: 7  Miss  [8,7]
Access: 4  Miss  [8,7,4]
Access: 2  Miss  [7,4,2]
Access: 5  Miss  [7,4,5]
Access: 4  Hit   [7,4,5]
Access: 7  Hit   [7,4,5]
Access: 3  Miss  [4,5,3]
Access: 4  Hit   [4,5,3]
Access: 5  Hit   [4,5,3]

40% hit rate
```

Question 2

For a cache of size 5, generate worst-case address reference streams for each of the following policies: FIFO, LRU, and MRU (worst-case reference streams cause the most misses possible).
For the worst case reference streams, how much bigger of a cache is needed to improve performance dramatically and approach OPT?

```
python3 paging-policy.py -C 5 -p FIFO -a 1,2,3,4,5,6,1,2,3,4 -c
python3 paging-policy.py -C 5 -p LRU -a 1,2,3,4,5,6,1,2,3,4 -c
python3 paging-policy.py -C 5 -p MRU -a 1,2,3,4,5,6,5,6,5,6 -c
```

If the cache size is equal to the maxpage we don't need to throw anything out and get the same result as OPT with every policy

Question 3

Generate a random trace (use python or perl). How would you expect the different policies to perform on such a trace?

The different policies perform as expected and like they did before.

```python
import random

numbers = [random.randint(1, 10) for _ in range(100)]
with open("no-locality.txt", "w") as file:
    for number in numbers:
        file.write(str(number) + "\n")
```

```
OPT:     FINALSTATS hits 55   misses 45   hitrate 55.00
LRU:     FINALSTATS hits 38   misses 62   hitrate 38.00
FIFO:    FINALSTATS hits 34   misses 66   hitrate 34.00
CLOCK:   FINALSTATS hits 34   misses 66   hitrate 34.00
MRU:     FINALSTATS hits 31   misses 69   hitrate 31.00
RAND:    FINALSTATS hits 33   misses 67   hitrate 33.00
```

Question 4

Now generate a trace with some locality.
How can you generate such a trace?
How does LRU perform on it?
How much better than RAND is LRU?
How does CLOCK do?
How about CLOCK with different numbers of clock bits?

```python
import random

frequent_numbers = [random.randint(1, 5) for _ in range(80)]
infrequent_numbers = [random.randint(6, 10) for _ in range(20)]
numbers = frequent_numbers + infrequent_numbers
random.shuffle(numbers)
with open("locality-80-20.txt", "w") as file:
    for number in numbers:
        file.write(str(number) + "\n")
```

```
OPT:     FINALSTATS hits 62   misses 38   hitrate 62.00
RAND:    FINALSTATS hits 48   misses 52   hitrate 48.00
LRU:     FINALSTATS hits 47   misses 53   hitrate 47.00
CLOCK:   FINALSTATS hits 45   misses 55   hitrate 45.00
FIFO:    FINALSTATS hits 44   misses 56   hitrate 44.00
MRU:     FINALSTATS hits 38   misses 62   hitrate 38.00
```

```
-f locality-80-20.txt -p CLOCK  -c -N -b 0    hitrate 34.00
-f locality-80-20.txt -p CLOCK  -c -N -b 1    hitrate 37.00
-f locality-80-20.txt -p CLOCK  -c -N -b 2    hitrate 45.00
-f locality-80-20.txt -p CLOCK  -c -N -b 3    hitrate 41.00
```

Question 5

Use a program like valgrind to instrument a real application and generate a virtual page reference stream.
For example, running
`valgrind --tool=lackey --trace-mem=yes ls`
will output a nearly-complete reference trace of every instruction and data reference made by the program ls.
To make this useful for the simulator above, you'll have to first transform each virtual memory reference into a virtual page-number reference (done by masking off the offset and shifting the resulting bits downward).

```python
import subprocess
import platform
import argparse

# Parse the command line argument
parser = argparse.ArgumentParser()
parser.add_argument("filename", help="the name of the file to trace")
args = parser.parse_args()
policy = args.filename

PAGE_SIZE = 4096

system = platform.system()

# check which system we are running and set the page size correct
if system == 'Windows':
    import ctypes
    sysinfo = ctypes.windll.kernel32.GetSystemInfo()
    PAGE_SIZE = sysinfo.dwPageSize
elif system == 'Linux' or system == 'Darwin':
    import resource
    PAGE_SIZE = resource.getpagesize()
else:
    PAGE_SIZE = None
    print('Unknown operating system, PAGE_SIZE not set')

print('PAGE_SIZE:', PAGE_SIZE)

# Run the valgrind command and save the output to a file
subprocess.run("valgrind --tool=lackey --trace-mem=yes ls &> ls_trace.txt", shell=True)


# Convert virtual memory references to virtual page numbers in decimal and write to file
with open("ls_trace.txt", "r") as f:
    with open("ls_trace_vpn.txt", "w") as fout:
        for line in f:
            if not line.startswith("=") and "," in line:
                address = int(line[3:line.index(",")], 16)
                page_number = address // PAGE_SIZE
                fout.write(str(page_number) + "\n")

# Call paging_policy.py with the given policy
command = "python3 paging-policy.py -p " + policy + " -f ls_trace_vpn.txt -c -N"
subprocess.run(command, shell=True)

subprocess.run("rm ls_trace_vpn.txt", shell=True)
subprocess.run("rm ls_trace.txt", shell=True)
```

This code can take a long time, depending on your system.
Mainly because the simulator is really inefficient

How big of a cache is needed for your application trace in order to satisfy a large fraction of requests?
Plot a graph of its working set as the size of the cache increases.