Project 4

Isaac Dudney

April 11, 2018

1 Psuedocode

A general description of how my code works is simple. Each of the cache replacement algorithms is represented by a Pure Function which takes two lists as input and returns a single list. The two list inputs are the pattern (eg `ABCDEABC`) and the available slots (which are initialized to None and replaced as the algorithm fills them). Each function initializes a special list of strings which store the text output, which is returned at the end of the function. Every function iterates over the pattern character by character and performs comparisons on the slots to determine behavior. It then appends output text to the list of strings according to the behavior; for example, if it gets a cache hit, it appends a `+` to the appropriate slot.

The main thread starts by taking user input; it exits if the input isn't integers, or if the input is out of bounds. It then initializes each necessary item (pattern and slots) using the input integers. The functions are then defined, along with a utility function to properly print and wrap the output.

Each algorithm is then called and printed using the utility function, after which the program exits.

However, I will also write psuedocode for each algorithm, including the printing utility function, here in a set of listings. Starting with `FIFO`.

1.1 FIFO

```
def FIFO(pattern,slots):
    # index in slots to replace
    index = 0
    # return special printing struct
    # inits with "FIFO  #: " for number of slots + 1
    # where # is the slot number
    r = ["FIFO  " + i + ": " for i in range(len(slots) + 1)]
    # redefine first line to have the input pattern
    # with 1 space after every char as per assignment
    r[0] = "Ref Str: " + " ".join(pattern)

    # for each item in the pattern, do stuff
    for f in pattern:

        # if the item is in the slots
        # wri():
        # magic write to proper place function for psuedocode
        # it writes a "+ " to the printing struct when it hits
        # and the proper output (f + " ") if it misses
        if f in slots: wri("hit")

        # if there's an empty slot
        elif None in slots:
            # magic write
            wri(f)
            # insert at first available empty slot
            slots[slots.index(None)] = f

        # if no empty slots exist
        else:
            wri(f)
            # replace slot at index
            slots[index] = f
            # if index is smaller than number of slots, increment
            if index < len(slots) - 1:
                index++
            # otherwise, reset to 0
            else:
                index = 0
    # return print struct
    return(r)
```

As stated in the comments, `wri()` is a magic function purely for psuedocode; looking in the actual code, each instance is replaced by three lines that depend on the algorithm for their logic. In the end, it ends up doing the same thing, producing good output, as can be seen here:

```
Ref Str: D C D A B D B C C D
FIFO  1: D   + A   D       +
FIFO  2:   C     B   + C +
```

The output will also wrap nicely, but we will cover that portion more extensively in the output section. For now, we will take a look at `LRU`.

1.2 LRU

```
1  def LRU(pattern,slots):
2      # here, instead of a plain index, we keep a time index of each use
3      # the time is relative to the place in pattern
4      # EG in the pattern ABCDEABC, the first B is time 1 (0-indexed)
5      index = [0 for f in slots]
6  
7      # same special printing struct
8      r = ["LRU   " + i + ": " for i in range(len(slots) + 1)]
9      r[0] = "Ref Str: " + " ".join(pattern)
10 
11     # for each item in the pattern, do stuff, by index!
12     for f in range(len(pattern)):
13 
14         # if the item is in the slots
15         if pattern[f] in slots:
16             # same magic write
17             wri("hit")
18             # this time, set the index of the hit to the current time
19             # f, or time, here is the index as defined in line 4
```

```python
            index[slots.index(pattern[f])] = f
            # breaking it down: slots.index() finds the corresponding
            # slot for the index list. We pass in the current item
            # in the pattern (pattern[f]) to find where we hit
            # we then set that to the current time in the index

        # if there's an empty slot
        elif None in slots:
            # magic write
            wri(pattern[f])
            # insert at first available empty slot
            # this time, record the index for reuse in next line
            i = slots.index(None)
            slots[i] = pattern[f]
            index[i] = f

        # if no empty slots exist
        else:
            # lots of indexes here! to put it in english:
            # find the number index of the smallest item in the index var
            i = index.index(min(index))
            # magic write
            wri(pattern[f])
            # set least recently used slot to new item
            slots[i] = pattern[f]
            # set current slot to most recently used
            index[i] = f
    # return print struct
    return(r)
```

The output of this function with the same input as the previously shown FIFO output would look like so:

```
Ref Str: D C D A B D B C C D
LRU   1: D   +   B   +     D
LRU   2:   C   A   D   C +
```

Moving on to the MIN algorithm.

1.3 MIN

```
def MIN(pattern,slots):
    # despite it being the technically impossible one,
    # it actually has the simplest implementation, given the
        pattern

    # printing struct
    r = ["MIN   " + i + ": " for i in range(len(slots) + 1)]
    r[0] = "Ref Str: " + " ".join(pattern)

    # for each item in pattern by index
    for f in range(len(pattern)):

        # if item exists, cache hit
        if pattern[f] in slots: wri("hit")

        # if empty slot exists
        elif None in slots:
            wri(pattern[f])
            slots[slots.index(None)] = pattern[f]

        # if no empty slots exist
        else:
            # here's where the logic happens!
            index_list = []
            # iterate over the slots
            # find the next index (by pattern) for each item
            for g in slots:
                # if it exists, append its next index
                # to the index_list
                if g in pattern[f:]:
                    index_list.append(pattern[f:].index(g) + f)
                # otherwise append an impossible index
                # (as len(pattern) !> 100)
                else:
                    index_list.append(101)
            # set index in slot to the index
            # of the greatest item in index_list
            if 101 in index_list: i = index_list.index(101)
            else: i = slots.index(pattern[max(index_list)])
            # write cache miss
            wri(pattern[f])
            # set slot to missed item
            slots[i] = pattern[f]
    # return print struct
    return(r)
```

Hopefully the explanations in the last `else` statement made sense. They are an accurate description of what occurs. The output of this function, with the same input as the others, would look like so:

```
Ref Str: D C D A B D B C C D
MIN   1: D   +     +       +
MIN   2:   C   A B   + C +
```

Now, finally, the easiest one to describe, `RAND`.

1.4 RAND

```
def RAND(pattern,slots):
    # printing struct
    r = ["RAND  " + i + ": " for i in range(len(slots) + 1)]
    r[0] = "Ref Str: " + " ".join(pattern)

    # for each in pattern
    for f in pattern:
        # if cache hit
        if f in slots: wri("hit")
        # if empty slot exists
        if None in slots:
            wri(f)
            slots[slots.index(None)] = f
        # if no empty slots available
        else:
            # pick a random one
            slots[random(len(slots))] = f
```

```
                wri(f)

    return(r)
```

As the `RAND` function is non-deterministic, here is the output of one iteration of the same input as the others. However, most outputs of `RAND` should look generally similar.

```
Ref Str: D C D A B D B C C D
RAND  1: D   +     +   C + D
RAND  2:   C   A B   +
```

1.5 Main

Finally for the psuedocode, we have to go over the main code; I will indicate the above function definitions with a simple `def NAME(...): ...` to indicate the function has been shrunk for size concerns. Here is the main body of the code in psuedocode:

```
# take user input
pattern_length = input("pattern length: ")
unique_characters = input("unique characters: ")
number_of_slots = input("number of slots: ")

# if any errors occur
if any not int: exit("not numbers")
if 10 > pattern_length > 101: exit("out of range")
```

```python
if 2 > unique_characters,number_of_slots > 11: exit("out of
    range")

# initialize the pattern
pattern = [choose(lowercase_ascii[:unique_characters])
           for f in range(pattern_length)]
# initialize the slots
slots = [None for f in range(number_of_slots)]

def FIFO(...): ...
def LRU(...): ...
def MIN(...): ...
def RAND(...): ...

# special printing function, will show detail in output section
dep pr(...): ...

pr(FIFO(pattern,slots))
pr(LRU(pattern,slots))
pr(MIN(pattern,slots))
pr(RAND(pattern,slots))
```

2 Output

Here, I will discuss the the printing function for both unwrapped (exactly to spec) and wrapped (not to spec, but hopefully in good taste). Firstly, I will show pattern output of the program for an unwrapped print.

```
ptrn lngth: 25
uniq chars: 5
slot lngth: 3

Ref Str: B D D E A A C A D E A B C C B C D C D E C A B D E
FIFO  1: B       A +   +   E     C +   +   +     + A     E
FIFO  2:   D +       C       A           D   +       B
FIFO  3:       E         D     B     +         E       D
---------------------------------------------------------
Ref Str: B D D E A A C A D E A B C C B C D C D E C A B D E
LRU   1: B       A +   +     +           D   +     A     E
LRU   2:   D +       C     E     C +   +   +     +     D
LRU   3:       E         D     B     +         E     B
---------------------------------------------------------
Ref Str: B D D E A A C A D E A B C C B C D C D E C A B D E
MIN   1: B       A +   +     + B     +   D   +         +
MIN   2:   D +           + E                   +         +
MIN   3:       E     C           + +   +   +     + A B
---------------------------------------------------------
Ref Str: B D D E A A C A D E A B C C B C D C D E C A B D E
RAND  1: B                   A   C +   + D C D     A   D E
RAND  2:   D +       C   D     B     +               +
RAND  3:       E A +   +   E                   + C
```

As can be seen, each piece of output is according to spec in the assignment sheet. The user's input is accepted and each function takes the appropriate variables to produce correct output. Each function (except `RAND`, due to the random nature, and `pr`, as it prints to the output) is also Pure. This means it edits no global state, only working on its own copies of input and producing unique output (not modifying global state and returning that).

Now that we have gone over the output in its spec-compliant form, now let's delve into the output in wrapping form. Here is the pattern output if it has to wrap:

```
ptrn lngth: 70
uniq chars: 4
slot lngth: 2

Ref Str: C B C B D C B D D C D B D D D A B A D A B A A B A B B
FIFO  1: C   +   D   B     C     D + +   B     A   + +   +
FIFO  2:   B   +   C   D +   + B       A   + D   B     +   + +
######## wrapping 1
Ref Str: A D A B D D B B B C D C C B B D D D D B C C C A D B C
FIFO  1:   D     + +       C   + + B +         +       A   B
FIFO  2: +   + B     + + +   D         + + + +   C + +   D   C
######## wrapping 2
Ref Str:  B C B D C B B
FIFO  1:    C     + B +
FIFO  2:  B   + D
--------------------------------------------------------------
--------------------------------------------------------------
-----------------------
Ref Str: C B C B D C B D D C D B D D D A B A D A B A A B A B B
LRU   1: C   +   D   B     C   B       A   +   +   + +   +
LRU   2:   B   +   C   D +   +   + + +   B   D   B     +   + +
######## wrapping 1
Ref Str: A D A B D D B B B C D C C B B D D D D B C C C A D B C
LRU   1:   D   B     + + +   D     B +         +       A   B
LRU   2: +   +   D +       C   + +     D + + +   C + +   D   C
######## wrapping 2
Ref Str:  B C B D C B B
LRU   1:    +   D   B +
LRU   2:  B   +   C
--------------------------------------------------------------
--------------------------------------------------------------
-----------------------
Ref Str: C B C B D C B D D C D B D D D A B A D A B A A B A B B
MIN   1: C   +     + B     C   B         +   D   B     +   + +
MIN   2:   B   + D     + +   +   + + + A   +   +   + +   +
######## wrapping 1
Ref Str: A D A B D D B B B C D C C B B D D D D B C C C A D B C
MIN   1: +   + B     + + + C   + + B +         + C + + A     C
MIN   2:   +     + +         +         + + + +           + B
######## wrapping 2
Ref Str:  B C B D C B B
MIN   1:    +     + B +
MIN   2:  B   + D
--------------------------------------------------------------
--------------------------------------------------------------
-----------------------
Ref Str: C B C B D C B D D C D B D D D A B A D A B A A B A B B
RAND  1: C   +   D C   D +   + B D + +   B     A   + +   +
RAND  2:   B   +     +     C           A   + D   B     +   + +
######## wrapping 1
Ref Str: A D A B D D B B B C D C C B B D D D D B C C C A D B C
RAND  1:   +   B     + + +     C + B +         +       A
RAND  2: +   +   D +       C D         + + + +   C + +   D B C
######## wrapping 2
```

```
Ref Str:  B C B D C B B
RAND  1:  +   + D
RAND  2:    +     + B +
```

As can be seen, each one wraps around all of its lines to produce a generally nice-looking output with clear indications of what is related to what.

The way this is done relies pretty heavily on the structure of the return values, and also happens to be a disgusting hack. I will post the full code of the printing function here, and then break it down in a separate listing as psuedocode with detailed explanations as to what's going on.

2.1 Disgusting Hack

```python
# necessary imports
from subprocess import run, PIPE

def pr(s):
    # disgusting hack to print according to spec on Linux
    wb = run(["tput", "cols"], stdout=PIPE)
    ws = wb.stdout.decode("utf-8")
    w = int("".join(ws.split()))
    for f in range((len(s[0]) // w) + 1):
        if f == 0:
            for g in s:
                print(g[:w])
        else:
            print("######## wrapping", f)
            for g in s:
                print(g[:8],g[(w * f) + 8:(w * f) + w - 2])
```

This code is dense and somewhat confusing. However, if we were to implement the program without wrapping, due to the way that the return variables of each algorithm are structured, the code could be reduced to two lines:

```
def pr(s):
    for f in s: print(f)
```

The longer one can be broken down into a few english descriptions of what's going on. The return structure is each line of the output separated into a list. So, for example, the actual return value of, for example, `FIFO`, given `10,4,2` as input would be:

```
['Ref Str: B B D A A C C A D D',
 'FIFO  1: B +   A +     + D + ',
 'FIFO  2:     D     C +       ']
```

With this structure, it's easy to see how a for loop could print this array properly. However, if we wanted to wrap the output nicely, we have to do something significantly more complex. I will demonstrate this with psuedocode of the `pr()` function.

```
def pr(s):
    # find the width of the current terminal
```

```python
    # this is the first three non-comment lines of the actual function
    # use the unix tool tput to find the cols of the current TTY
    w = width(current_terminal)
    # iterate over the number of chunks that exceed the width
    for f in range((len(s[0]) // w) + 1):
        # the first one should be clean in case it doesn't need to wrap
        if f == 0:
            # do the one-liner, but only print up to the width
            # the [:w] slice returns a substring up to w chars long
            for g in s: print(g[:w])
        else:
            # print the chunk it's on
            print("######## wrapping", f)
            # one liner but disgusting
            for g in s:
                # print up to the indicators (RAND  1: etc) by printing g[:8]
                # then print from 8 + (width * current_chunk) up to
                # next chunk
                print(g[:8],g[(w * f) + 8:(w * f) + w - 2])
```

The hack extensively uses Python slicing. Slicing takes an ordered iterable (strings or lists, for example) and returns a new ordered iterable by these rules: `[start:stop:skip]`. Each one of those variables is optional. Start will default to 0, stop to -1 (the end), and skip will default to 1 (every item).

In our example, we used start and stop, but not skip.

The first and second slice, `g[:w]` and `g[:8]`, only used stop, stopping at the width of the terminal and 8, respectively. The third disgusting one used start and stop. The start was `(w * f) + 8`. `w` here is the width of the terminal, while `f` is the "chunk" of the output we are on. We then add 8 to make sure we are aligned with the previously printed introductory chunk. We stop at `(w * f) + w - 2`. We go to the end of the current chunk by going to the start, adding the width, and subtracting two to ensure no wrapping by the terminal.

2.2 Scoring

Finally, the scoring. I was tired when I wrote this code so I used some pretty brute-force methods; but basically, it simply iterates over every output string, counts the occurences of hits (+), and uses that as the score to divide by the global pattern length. It then prints each one, and then the best and worst using a bit of list comprehension. The code is as follows:

```
scores_counter = [fifo,lru,m,rand]
count = [0] * 4
for f in scores_counter:
    for g in f: count[scores_counter.index(f)] += g.count("+")
    for g in count: g /= p

scores = [f / p for f in count]

print("\nCache Hit Rates:\n")
print("FIFO : " + str(count[0]) + " of " + str(p) + " = " +
    str(scores[0]))
print("LRU  : " + str(count[1]) + " of " + str(p) + " = " +
    str(scores[1]))
print("MIN  : " + str(count[2]) + " of " + str(p) + " = " +
    str(scores[2]))
print("FIFO : " + str(count[3]) + " of " + str(p) + " = " +
    str(scores[3]))
print("\nBest:  " +
    scores_counter[scores.index(max(scores))][1][:4])
print("Worst: " +
    scores_counter[scores.index(min(scores))][1][:4])
```

It isn't pretty, but it produces the correct scoring output on every program run.

3 Conclusion

To wrap it all up, the program does the hard parts (the algorithms) elegantly and the easy parts (printing) in a brutish kind of way. The program could be improved, but overall, the important parts are very elegantly and minimally implemented. Each one is correct for every iteration and test I've thrown at it, and I am proud of what I've done.