

Project 4

Isaac Dudney

April 12, 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

```
1 def FIFO(pattern,slots):
2     # index in slots to replace
3     index = 0
4     # return special printing struct
5     # inits with "FIFO  #: " for number of slots + 1
6     # where # is the slot number
7     r = ["FIFO  " + i + ": " for i in range(len(slots) + 1)]
8     # redefine first line to have the input pattern
9     # with 1 space after every char as per assignment
10    r[0] = "Ref Str: " + " ".join(pattern)
11
12    # for each item in the pattern, do stuff
13    for f in pattern:
14
15        # if the item is in the slots
16        # wri():
17        # magic write to proper place function for psuedocode
18        # it writes a "+ " to the printing struct when it hits
19        # and the proper output (f + " ") if it misses
20        if f in slots: wri("hit")
21
```

```

22         # if there's an empty slot
23         elif None in slots:
24             # magic write
25             wri(f)
26             # insert at first available empty slot
27             slots[slots.index(None)] = f
28
29         # if no empty slots exist
30         else:
31             wri(f)
32             # replace slot at index
33             slots[index] = f
34             # if index is smaller than number of slots,
35             # increment
36             if index < len(slots) - 1:
37                 index++
38             # otherwise, reset to 0
39             else:
40                 index = 0
41         # return print struct
42         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:

```

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

```

```
3 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
3     # each use
4     # the time is relative to the place in pattern
5     # EG in the pattern ABCDEABC, the first B is time 1
6     # (0-indexed)
7     index = [0 for f in slots]
8
9     # same special printing struct
10    r = ["LRU " + i + ": " for i in range(len(slots) + 1)]
11    r[0] = "Ref Str: " + " ".join(pattern)
12
13    # for each item in the pattern, do stuff, by index!
14    for f in range(len(pattern)):
15
16        # if the item is in the slots
17        if pattern[f] in slots:
18
19            # same magic write
20            wri("hit")
21
22            # this time, set the index of the hit to the
23            # current time
24
25            # f, or time, here is the index as defined in line
26            4
```

```

20         index[slots.index(pattern[f])] = f
21         # breaking it down: slots.index() finds the
           corresponding
22         # slot for the index list. We pass in the current
           item
23         # in the pattern (pattern[f]) to find where we hit
24         # we then set that to the current time in the index
25
26     # if there's an empty slot
27     elif None in slots:
28         # magic write
29         wri(pattern[f])
30         # insert at first available empty slot
31         # this time, record the index for reuse in next
           line
32         i = slots.index(None)
33         slots[i] = pattern[f]
34         index[i] = f
35
36     # if no empty slots exist
37     else:
38         # lots of indexes here! to put it in english:
39         # find the number index of the smallest item in
           the index var
40         i = index.index(min(index))
41         # magic write
42         wri(pattern[f])
43         # set least recently used slot to new item
44         slots[i] = pattern[f]
45         # set current slot to most recently used

```

```

46         index[i] = f
47     # return print struct
48     return(r)

```

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

```

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

```

Moving on to the MIN algorithm.

1.3 MIN

```

1 def MIN(pattern, slots):
2     # despite it being the technically impossible one,
3     # it actually has the simplest implementation, given the
      pattern
4
5     # printing struct
6     r = ["MIN   " + i + ": " for i in range(len(slots) + 1)]
7     r[0] = "Ref Str: " + " ".join(pattern)
8
9     # for each item in pattern by index
10    for f in range(len(pattern)):
11
12        # if item exists, cache hit
13        if pattern[f] in slots: wri("hit")
14

```

```

15         # if empty slot exists
16     elif None in slots:
17         wri(pattern[f])
18         slots[slots.index(None)] = pattern[f]
19
20     # if no empty slots exist
21     else:
22         # here's where the logic happens!
23         index_list = []
24         # iterate over the slots
25         # find the next index (by pattern) for each item
26         for g in slots:
27             # if it exists, append its next index
28             # to the index_list
29             if g in pattern[f:]:
30                 index_list.append(pattern[f:].index(g) + f)
31             # otherwise append an impossible index
32             # (as len(pattern) != 100)
33             else:
34                 index_list.append(101)
35
36         # set index in slot to the index
37         # of the greatest item in index_list
38         if 101 in index_list: i = index_list.index(101)
39         else: i = slots.index(pattern[max(index_list)])
40
41         # write cache miss
42         wri(pattern[f])
43         # set slot to missed item
44         slots[i] = pattern[f]
45
46     # return print struct
47     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:

```

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

```

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

1.4 RAND

```

1 def RAND(pattern, slots):
2     # printing struct
3     r = ["RAND " + i + ": " for i in range(len(slots) + 1)]
4     r[0] = "Ref Str: " + " ".join(pattern)
5
6     # for each in pattern
7     for f in pattern:
8         # if cache hit
9         if f in slots: wri("hit")
10        # if empty slot exists
11        if None in slots:
12            wri(f)
13            slots[slots.index(None)] = f
14        # if no empty slots available
15        else:
16            # pick a random one
17            slots[random(len(slots))] = f

```



```

18         wri(f)
19
20     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.

```

1 Ref Str: D C D A B D B C C D
2 RAND  1: D   +       +   C + D
3 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:

```

1 # take user input
2 pattern_length = input("pattern length: ")
3 unique_characters = input("unique characters: ")
4 number_of_slots = input("number of slots: ")
5
6 # if any errors occur
7 if not int: exit("not numbers")
8 if 10 > pattern_length > 101: exit("out of range")

```

```

9 if 2 > unique_characters, number_of_slots > 11: exit("out of
    range")
10
11 # initialize the pattern
12 pattern = [choose(lowercase_ascii[:unique_characters])
13             for f in range(pattern_length)]
14 # initialize the slots
15 slots = [None for f in range(number_of_slots)]
16
17 def FIFO(...): ...
18 def LRU(...): ...
19 def MIN(...): ...
20 def RAND(...): ...
21
22 # special printing function, will show detail in output section
23 def pr(...): ...
24
25 pr(FIFO(pattern, slots))
26 pr(LRU(pattern, slots))
27 pr(MIN(pattern, slots))
28 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.

```

1 ptrn lngth: 25
2 uniq chars: 5
3 slot lngth: 3
4
5 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
6 FIFO  1: B          A +    +    E      C +    +    +      + A      E
7 FIFO  2:   D +          C      A          D    +      B
8 FIFO  3:          E      D      B      +      E      D
9 -----
10 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
11 LRU   1: B          A +    +      +      D    +      A      E
12 LRU   2:   D +          C      E      C +    +    +      +      D
13 LRU   3:          E      D      B      +      E      B
14 -----
15 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
16 MIN   1: B          A +    +      + B      +    D    +      +
17 MIN   2:   D +          + E          +      +
18 MIN   3:          E      C          + +    +    +      + A B
19 -----
20 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
21 RAND  1: B          A      C +    + D C D      A      D E
22 RAND  2:   D +          C      D      B      +      +
23 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, let's delve into the output in wrapping form. Here is the pattern output if it has to wrap:

```

1 ptrn lngth: 70
2 uniq chars: 4
3 slot lngth: 2
4
5 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
6 FIFO  1: C   +   D   B       C       D + +   B       A   + +   +
7 FIFO  2:   B   +   C   D +   + B       A   + D   B       +   + +
8 ##### wrapping 1
9 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
10 FIFO  1:   D       + +       C   + + B +       +       A   B
11 FIFO  2: +   + B       + + +   D       + + + +   C + +   D   C
12 ##### wrapping 2
13 Ref Str:  B C B D C B B
14 FIFO  1:    C       + B +
15 FIFO  2:   B   + D
16 -----
17 -----
18 -----
19 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
20 LRU   1: C   +   D   B       C   B       A   +   +   + +   +
21 LRU   2:   B   +   C   D +   +   + + +   B   D   B       +   + +
22 ##### wrapping 1
23 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
24 LRU   1:   D   B       + + +   D       B +       +       A   B

```

```

25 LRU    2: +   +   D +           C   + +   D + + +   C + +   D   C
26 ##### wrapping 2
27 Ref Str: B C B D C B B
28 LRU    1:   +   D   B +
29 LRU    2:  B   +   C

30 -----
31 -----
32 -----
33 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
34 MIN    1: C   +   + B   C   B           +   D   B   +   + +
35 MIN    2:  B   + D   + +   +   + + + A   +   +   + +   +
36 ##### wrapping 1
37 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
38 MIN    1: +   + B   + + + C   + + B +           + C + + A   C
39 MIN    2:  +   + +           +           + + + +           + B
40 ##### wrapping 2
41 Ref Str: B C B D C B B
42 MIN    1:   +   + B +
43 MIN    2:  B   + D

44 -----
45 -----
46 -----
47 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
48 RAND   1: C   +   D C   D +   + B D + +   B   A   + +   +
49 RAND   2:  B   +   +   C           A   + D   B   +   + +
50 ##### wrapping 1
51 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
52 RAND   1:  +   B   + + +   C + B +           +   A
53 RAND   2: +   +   D +           C D           + + + +   C + +   D B C
54 ##### wrapping 2

```

```

55 Ref Str:  B C B D C B B
56 RAND  1:  +      + D
57 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 pseudocode with detailed explanations as to what's going on.

2.1 Disgusting Hack

```

1 # necessary imports
2 from subprocess import run, PIPE
3
4 def pr(s):
5     # disgusting hack to print according to spec on Linux
6     wb = run(["tput", "cols"], stdout=PIPE)
7     ws = wb.stdout.decode("utf-8")
8     w = int("".join(ws.split()))
9     for f in range((len(s[0]) // w) + 1):
10         if f == 0:
11             for g in s:
12                 print(g[:w])
13         else:
14             print("##### wrapping", f)
15             for g in s:

```

```
16         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:

```
1 def pr(s):
2     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:

```
1 ['Ref Str: B B D A A C C A D D',
2  'FIFO  1: B +   A +       + D + ',
3  '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.

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

```

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

```
1 scores_counter = [fifo,lru,m,rand]
2 count = [0] * 4
3 for f in scores_counter:
4     for g in f: count[scores_counter.index(f)] += g.count("+")
5
6 scores = [f / p for f in count]
7
8 print("\nCache Hit Rates:\n")
```

```

9 # count stores count of hits, scores stores score (hits/total
   length)
10 print("FIFO : " + str(count[0]) + " of " + str(p) + " = " +
      str(scores[0]))
11 print("LRU  : " + str(count[1]) + " of " + str(p) + " = " +
      str(scores[1]))
12 print("MIN  : " + str(count[2]) + " of " + str(p) + " = " +
      str(scores[2]))
13 print("RAND : " + str(count[3]) + " of " + str(p) + " = " +
      str(scores[3]))
14 # takes advantage of the fact that scores_counter stores a
   list of lists
15 # [1] accesses second line, [:4] accesses name strings
16
17 winners = [f == max(count) for f in count]
18 losers = [f == min(count) for f in count]
19 win, lose = "", ""
20 for f in range(len(winners)):
21     if winners[f]: win += scores_counter[f][1][:4] + " "
22     if losers[f]: lose += scores_counter[f][1][:4] + " "
23 print("\nBest:  " + win)
24 print("Worst: " + lose)

```

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. To finish off, here's the full and finished output with every bell and whistle that the printing is capable of triggered:

```
1 ptrn lngth: 35
2 uniq chars: 10
3 slot lngth: 3
4
5 Ref Str: A G H D A E G E H J D B E G E A C A A J A
6 FIFO  1: A      D      G      D      G      J
7 FIFO  2:  G      A      H      B      A  +  +  +
8 FIFO  3:      H      E  +  J      E  +  C
9 ##### wrapping 1
10 Ref Str: F B A F F D E H D D
11 FIFO  1:      F  +      H
12 FIFO  2: + B      D      +  +
13 FIFO  3:      A      E
14 -----
15 -----
16 Ref Str: A G H D A E G E H J D B E G E A C A A J A
17 LRU   1: A      D      G      J      E  +      J
18 LRU   2:  G      A      H      B      A  +  +  +
19 LRU   3:      H      E  +      D      G      C
20 ##### wrapping 1
21 Ref Str: F B A F F D E H D D
```

```

22 LRU    1: F      + +      H
23 LRU    2:   B      D      + +
24 LRU    3:   A      E
25 -----
26 -----
27 Ref Str: A G H D A E G E H J D B E G E A C A A J A
28 MIN    1: A      + E      +      +      C
29 MIN    2:   G      +      H J      +
30 MIN    3:   H D      + B      G      A      + + +
31 ##### wrapping 1
32 Ref Str: F B A F F D E H D D
33 MIN    1: +      + + D      + +
34 MIN    2:   B      E H
35 MIN    3:   +
36 -----
37 -----
38 Ref Str: A G H D A E G E H J D B E G E A C A A J A
39 RAND    1: A      +      J      A      + + +
40 RAND    2:   G      E      + H      B      G      C      J
41 RAND    3:   H D      G      D      E      +
42 ##### wrapping 1
43 Ref Str: F B A F F D E H D D
44 RAND    1:   A
45 RAND    2:   B      E H
46 RAND    3: F      + + D      + +
47
48 Cache Hit Rates:
49
50 FIFO   : 10 of 35 = 0.2857142857142857
51 LRU    : 10 of 35 = 0.2857142857142857

```

```
52 MIN   : 17 of 35 = 0.4857142857142857
53 FIFO  : 11 of 35 = 0.3142857142857143
54
55 Best:  MIN
56 Worst: FIFO LRU
```