

# 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

```
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, increment
35             if index < len(slots) - 1:
36                 index++
37             # otherwise, reset to 0
38             else:
39                 index = 0
40         # return print struct
41         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 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
19             # this time, set the index of the hit to the current time
20             # f, or time, here is the index as defined in line 4
21             index[slots.index(pattern[f])] = f
22
23             # breaking it down: slots.index() finds the corresponding
24             # slot for the index list. We pass in the current item
25             # in the pattern (pattern[f]) to find where we hit
26             # 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

FIFO output 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         # set index in slot to the index
36         # of the greatest item in index_list
37         if 101 in index_list: i = index_list.index(101)
38         else: i = slots.index(pattern[max(index_list)])
39         # write cache miss
40         wri(pattern[f])
41         # set slot to missed item
42         slots[i] = pattern[f]
43     # return print struct
44     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 any 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 dep 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 output of the program as a whole 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, now let's delve into the output in wrapping form. Here is the output if it has to wrap:

```

1 ptrn length: 70
2 uniq chars: 4
3 slot length: 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 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
18 LRU   1: C   +   D   B      C   B      A   +   +   + +   +
19 LRU   2:  B   +   C   D +   +   + + +   B   D   B      +   + +
20 ##### wrapping 1
21 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
22 LRU   1:  D   B      + + +   D      B +      +      A   B
23 LRU   2: +   +   D +      C   + +      D + + +   C + +   D   C
24 ##### wrapping 2
25 Ref Str:  B C B D C B B
26 LRU   1:   +   D   B +
27 LRU   2:  B      +   C
28 -----
29 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
30 MIN   1: C   +      + B      C   B      +   D   B      +   + +
31 MIN   2:  B   + D      + +   +   + + +   A   +   +   + +   +
32 ##### wrapping 1
33 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
34 MIN   1: +   + B      + + +   C   + + B +      + C + +   A      C
35 MIN   2:  +      + +      +      + + + +      + B
36 ##### wrapping 2
37 Ref Str:  B C B D C B B
38 MIN   1:   +      + B +
39 MIN   2:  B      + D

```

```

40 -----
41 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
42 RAND  1: C   +   D C   D +   + B D + +   B   A   + +   +
43 RAND  2:   B   +   +   C   A   + D   B   +   + +
44 ##### wrapping 1
45 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
46 RAND  1:   +   B   + + +   C + B +   +   A
47 RAND  2: +   +   D +   C D   + + + +   C + +   D B C
48 ##### wrapping 2
49 Ref Str:   B C B D C B B
50 RAND  1:   +   + D
51 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.