CS150A Homework2 -- Coding

Instructions / Notes:

Read these carefully

- **Please read all the points of the "Notes" sections- they're important for this homework!!!**
- You **are not required to do any plotting in this homework - only in certain problems to provide the tuples that would generate a plot.** You can then optionally plot (in the notebook with matplotlib, in Excel, wherever works)
- You **may** create new IPython notebook cells to use for e.g. testing, debugging, exploring, etc.- this is encouraged in fact!- **just make sure that your final answer for each question is *in its own cell* and *clearly indicated***
- *Have fun!*

Problem 1: Double Trouble

[25 points total]

In this problem we'll explore an optimization often referred to as ***double buffering***, which we'll use to speed up the **external merge sort algorithm**.

Although we haven't explicitly modeled it in many of our calculations so far, recall that *sequential IO* (i.e. involving reading from / writing to consecutive pages) is generally much faster that *random access IO* (any reading / writing that is not sequential). Additionally, on newer memory technologies like SSD reading data can be faster than writing data (if you want to read more about SSD access patterns look here.

In other words, for example, if we read 4 consecutive pages from file A, this should be much faster than reading 1 page from A, then 1 page from file B, then the next page from A.

In this problem, we will begin to model this, by assuming that 4 sequential *READS* are "free", i.e. the total cost of 4 sequential reads is 1 IO. Sequential writes are never free, therefore the cost of N writes is always N.

Other important notes:

- **NO REPACKING:** Consider the external merge sort algorithm using the basic optimizations, but do not use the repacking optimization.
- **ONE BUFFER PAGE RESERVED FOR OUTPUT:** Assume we use one page for output in a merge, e.g. a B -way merge would require $B + 1$ buffer pages
- **REMEMBER TO ROUND:** Take ceilings (i.e. rounding up to nearest integer values) into account in this problem for full credit! Note that we have sometimes omitted these (for simplicity).
- **Consider worst case cost:** In other words, if 2 reads *could happen* to be sequential, but in general might not be, consider these random IO

Part (a)

[15 points]

Consider a modification of the external merge sort algorithm where **reads are always read in 4-page chunks (i.e. 4 pages sequentially at a time)** so as to take advantage of sequential reads. Calculate the cost of performing the external merge sort for a setup having $B + 1 = 20$ buffer pages and an unsorted input file with 160 pages.

Show the steps of your work and make sure to explain your reasoning by writing them as python comments above the final answers.

Part (a.i)

What is the **exact** IO cost of spliting and sorting the files? As is standard we want runs of size $B + 1$.

In [159…
```
import math
```

In [160…
```
# 1 I/O can read 4 pages
# the size of each run is the size of the buffer, which is 20
# the cost of read for each run is ceil(20/4)
# multiply by 8 since there are 8 runs
# the cost of write needs sequential writes, which is the size of the input file
io_split_sort = math.ceil(20/4)*8+20*8
io_split_sort
```

Out[160… 200

Part (a.ii)

After the file is split and sorted, we can merge n runs into 1 using the merge process. What is largest n we could have, given reads are always read in 4-page chunks? Note: this is known as the arity of the merge.

In [161…
```
# the number of pages in the buffer which is available for us to merge is 19
# each read will cause 4 pages of a same run to be read into the buffer
# so the number of different runs is floor(19/4)
merge_arity = math.floor(19/4)
merge_arity
```

Out[161… 4

Part (a.iii)

How many passes of merging are required?

In [162…
```
# since we can merge 4 runs for each pass
# for the first pass of merging, we can get ceil(8/4)=2 runs
# since 2<4, then the 2 runs can both be mergrd into 1 run in one pass (we can also get this by ceil(2/4
# then totally 2 passes
merge_passes = 2
merge_passes
```

Out[162… 2

Part (a.iv)

What is the IO cost of the first pass of merging? Note: the highest arity merge should always be used.

In [163…
```
# the first pass of merging can merge 8 runs into 2 runs
# for 4 runs of the total 8 runs
# the number of read is 4*(20/4)=20 (the first 4 is beacuse we need 4 read to load data into the buffer,
# and the total number we need for each run is 20/4)
# multiply by 2 to include other 4 runs
# plus 160 to include the cost of read
merge_pass_1 = 4*math.ceil(20/4)*2+160
merge_pass_1
```

Out[163… 200

Part (a.v)

What is the total IO cost of running this external merge sort algorithm? **Do not forget to add in the remaining passes (if any) of merging.**

In [164…
```
# after the first pass of merging, we have 2 runs of 80 pages for each
# the cost of read for the second pass of merging is 2*(80/4)
# plus 160 to get the cost of write
# then we get the cost of the 2nd pass is 2*(80/4)+160=200
# finally plus all the cost above
# (include the io_split_sort,merge_pass_1 and 200)
total_io = io_split_sort+merge_pass_1+200
total_io
```

Out[164… 600

Part (b)

[5 points]

Now, we'll generalize the reasoning above by writing a python function that computes the *approximate** cost of performing this version of external merge sort for a setup having $B+1$ buffer pages, a file with N pages, and where we now read in P-page chunks (replacing our fixed 4 page chunks in Part (a)).

**Note: our approximation will be a small one- for simplicity, we'll assume that each pass of the merge phase has the same IO cost, when actually it can vary slightly... Everything else will be exact given our model!*

We'll call this function `external_merge_sort_cost(B,N,P)` , and we'll compute it as the product of the cost of reading in and writing out all the data (which we do each pass), and the number of passes we'll have to do.

Even though this is an approximation, **make sure to take care of floor / ceiling operations- i.e. rounding down / up to integer values properly!**

Importantly, to simplify your calculations: Your function will only be evaluated on cases where the following hold:

- **(B + 1) % P == 0** (i.e. the buffer size is divisible by the chunk size)
- **N % (B + 1) == 0** (i.e. the file size is divisible by the buffer size)

Part (b.i)

First, let's write a python function that computes the **exact** total IO cost to create the initial runs:

In [165…
```
def cost_initial_runs(B, N, P):
    # Calculate the number of runs
    runs = N / (B + 1)

    # Calculate the read cost: every run needs to be read in chunks of size P
    # Given the number of pages in a run (B+1), the read cost for one run:
    read_cost_per_run = math.ceil((B + 1) / P)

    # Since each run is read and then written completely:
    # Write cost for one run is exactly B+1 (no chunk optimization for writes)
    write_cost_per_run = B + 1

    # Total cost for one run:
    total_cost_per_run = read_cost_per_run + write_cost_per_run

    # Total cost for all runs:
    total_cost = total_cost_per_run * runs
    return total_cost
```

Part (b.ii)

Next, let's write a python function that computes the *approximate** total IO cost to read in and then write out all the data during one pass of the merge:

In [166…
```
def cost_per_pass(B, N, P):
    # Number of pages read and written in each pass:
    # We read the entire file and write it back
    read_pages = N
    write_pages = N

    # Reading cost, given that it happens in chunks of P:
    read_cost = math.ceil(read_pages / P)

    # Write cost is simply the number of pages since every page written costs 1 IO
    write_cost = write_pages

    # Total cost per pass:
```

```
    total_cost = read_cost + write_cost
    return total_cost
```

**Note that this is an approximation: when we read in chunks during the merge phase, the cost per pass actually varies slightly due to 'rounding issues' when the file is split up into runs... but this is a small difference*

Part (b.iii)

Next, let's write a python function that computes the **exact** total number of passes we'll need to do

In [167…
```
def num_passes(B, N, P):
    # Start with the number of initial runs
    initial_runs = N / (B + 1)

    # Calculate the number of passes required to merge all runs into a single run
    passes = 0
    while initial_runs > 1:
        # Calculate maximum number of runs that can be merged in each pass:
        merge_arity = math.floor((B + 1 - 1) / P)  # minus 1 for output buffer
        if merge_arity < 1:
            merge_arity = 1  # ensure at least two-way merge if buffer settings are very low
        initial_runs = math.ceil(initial_runs / merge_arity)
        passes += 1
    return passes
```

Finally, our total cost function is:

In [168…
```
def external_merge_sort_cost(B, N, P):
    return cost_initial_runs(B,N,P) + cost_per_pass(B,N,P)*num_passes(B,N,P)
```

In [169…
```
# Testing with the example provided in Part (a)
# B+1 = 20, N = 160, P = 4
B = 19
N = 160
P = 4
test_cost = external_merge_sort_cost(B, N, P)
test_cost
```

Out[169…
```
600.0
```

Part (c)

[10 points]

For $B + 1 = 100$ and $N = 1000$, find the optimal P according to your IO cost equation above. Return both the optimal P value (`P_opt`) and the list of tuples ***for feasible values of*** P that would generate a plot of P vs. IO cost, at resolution $= 1$(every value of P), stored as `points` :

In [170…
```
# Save the optimal value here
P = 10

# Save a list of tuples of (P, io_cost) here, for all feasible P's
points = []
for p in range(1,50):
        points.append((p,external_merge_sort_cost_simple(99,1000,p)))
points
```

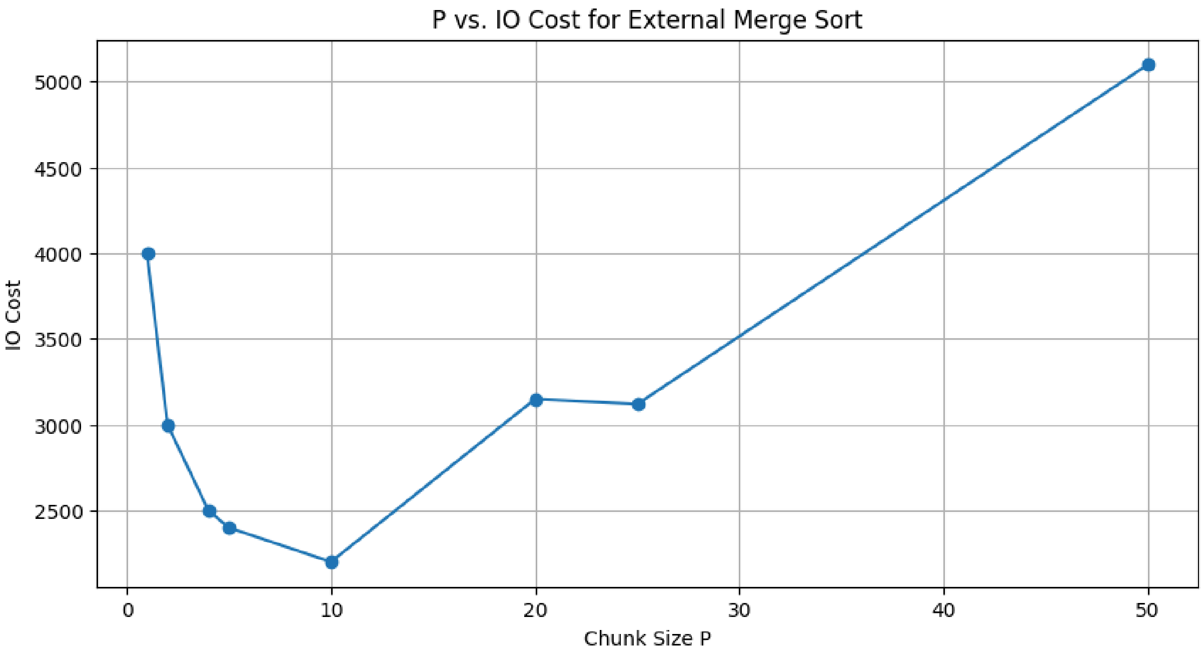
Out[170…
```
[(1, 4000.0),
 (2, 3000.0),
 (3, 2674.0),
 (4, 2500.0),
 (5, 2400.0),
 (6, 2337.0),
 (7, 2293.0),
 (8, 2255.0),
 (9, 2232.0),
 (10, 2200.0),
 (11, 3282.0),
 (12, 3258.0),
 (13, 3234.0),
 (14, 3224.0),
 (15, 3204.0),
 (16, 3196.0),
 (17, 3178.0),
 (18, 3172.0),
 (19, 3166.0),
 (20, 3150.0),
 (21, 3146.0),
 (22, 3142.0),
 (23, 3138.0),
 (24, 3134.0),
 (25, 3120.0),
 (26, 4157.0),
 (27, 4154.0),
 (28, 4148.0),
 (29, 4145.0),
 (30, 4142.0),
 (31, 4139.0),
 (32, 4136.0),
 (33, 4133.0),
 (34, 5150.0),
 (35, 5146.0),
 (36, 5142.0),
 (37, 5142.0),
 (38, 5138.0),
 (39, 5134.0),
 (40, 5130.0),
 (41, 5130.0),
 (42, 5126.0),
 (43, 5126.0),
 (44, 5122.0),
 (45, 5122.0),
 (46, 5118.0),
 (47, 5118.0),
 (48, 5114.0),
 (49, 5114.0)]
```

*Below we provide starter code for using `matplotlib` in the notebook, if you want to generate the graph of P vs. IO cost; however any other software that allows you to visualize the plot (Excel, Google spreadsheets, MATLAB, etc) is fine!

In [171…
```
# Shell code for plotting in matplotlib
%matplotlib inline
import matplotlib.pyplot as plt

# Plot
# plt.plot(*zip(*points))
# plt.show()
```

In [172…
```
B = 99  # since B+1 = 100
N = 1000
feasible_Ps = [P for P in range(1, B+1) if (B + 1) % P == 0]  # feasible P values satisfying (B + 1) % P

def num_passes_simple(B, N, P):
    # Simpler and more efficient calculation of the number of passes
    initial_runs = N / (B + 1)
    passes = 0
    while initial_runs > 1:
        merge_arity = (B + 1) // P  # number of chunks we can fit into the buffer
        initial_runs = math.ceil(initial_runs / merge_arity)
```

```python
            passes += 1
    return passes

# Redefine the main cost function to use the simplified passes calculation
def external_merge_sort_cost_simple(B, N, P):
    initial_cost = cost_initial_runs(B, N, P)
    per_pass_cost = cost_per_pass(B, N, P)
    total_passes = num_passes_simple(B, N, P)
    return initial_cost + per_pass_cost * total_passes

# Calculate the IO cost for each feasible P again and find the optimal P using the simplified model
points_simple = []
min_cost_simple = float('inf')
P_opt_simple = None

for P in feasible_Ps:
    io_cost = external_merge_sort_cost_simple(B, N, P)
    points_simple.append((P, io_cost))
    if io_cost < min_cost_simple:
        min_cost_simple = io_cost
        P_opt_simple = P

P_opt_simple, points_simple

# Plotting P vs. IO cost from the simplified calculation
plt.figure(figsize=(10, 5))
plt.plot(*zip(*points_simple), marker='o')
plt.title("P vs. IO Cost for External Merge Sort")
plt.xlabel("Chunk Size P")
plt.ylabel("IO Cost")
plt.grid(True)
plt.show()
```



Problem 2: IO Cost Models

[15 points total]

In this problem we consider different join algorithms when joining relations $R(A, B)$,$S(B, C)$, and $T(C, D)$. We want to investigate the cost of various pairwise join plans and try to determine the best join strategy given some conditions.

Specifically, for each part of this question, we are intereseted determining some (or all) of the following variables:

- `P_R` : Number of pages of R
- `P_S` : Number of pages of S

- `P_RS` : Number of pages of output (and input) RS
- `P_T` : Number of pages of T
- `P_RST` : Number of pages of output (and input) RST
- `B` : Number of pages in buffer
- `IO_cost_join1` : Total IO cost of first join
- `IO_cost_join2` : Total IO cost of second join

Note:

- ** The output of join1 is always feed as one of the inputs to join 2 **
- **Use the "vanilla" versions of the algorithms as presented in lecture, *i.e. without any of the optimizations we mentioned***
- **Again assume we use one page for output, as in lecture!**
- ** The abbreviates for the joins used are Sort-Merge Join (SMJ), Hash Join (HJ), and Block Nested Loop Join (BNLJ). **

Part (a)

[8 points]

Given:

- `P_R` : 10
- `P_S` : 100
- `P_T` : 1000
- `P_RS` : 25
- `P_ST` : 250
- `P_RST` : 125
- `B` : 16

Compute the IO cost for the following query plans:

- IO_Cost_HJ_1 where only hash join is used, $join1 = R(a,b), S(b,c)$ with hash table based on $S(b,c)$ and $join2 = join1(a,b,c), T(c,d)$ with hash table based on $join1(a,b,c)$
- IO_Cost_HJ_2 where only hash join is used, $join1 = T(c,d), S(b,c)$ with hash table based on $S(b,c)$ and $join2 = join1(b,c,d), R(a,b)$ with hash table based on $join1(a,b,c)$
- IO_Cost_SMJ_1 where only sort merge join is used, $join1 = R(a,b), S(b,c)$ and $join2 = join1(a,b,c), T(c,d)$
- IO_Cost_SMJ_2 where only sort merge join is used, $join1 = T(c,d), S(b,c)$ and $join2 = join1(b,c,d), R(a,b)$
- IO_Cost_BNLJ_1 where only block nested loop join is used, $join1 = R(a,b), S(b,c)$ and $join2 = join1(a,b,c), T(c,d)$
- IO_Cost_BNLJ_2 where only block nested loop join is used, $join1 = T(c,d), S(b,c)$ and $join2 = join1(b,c,d), R(a,b)$

Note: again, be careful of rounding for this problem. Use ceiling/floors whenever it is necessary.

Include 1-2 sentences (as a python comment) above each answer explaining the performance for each algorithm/query plan.

In [173…

```
from math import ceil,log

P_R = 10
P_S = 100
P_T = 1000
P_RS = 25
P_RST = 125
P_ST = 250
B = 16

# Compute IO costs for different join strategies

# Hash Join 1
```

```python
IO_Cost_HJ_1 = 3 * (P_R + P_S) + P_RS + 3 * (P_RS + P_T) + P_RST

# Hash Join 2
IO_Cost_HJ_2 = 3 * (P_T + P_S) + P_ST + 3 * (P_ST + P_R) + P_RST

# Sort-Merge Join 1
# Sorting costs for each relation using hypothetical logarithmic factors
sort_cost_R = 2 * P_R * (1 + ceil(log(ceil(P_R / B), B-1)))
sort_cost_S = 2 * P_S * (1 + ceil(log(ceil(P_S / B), B-1)))
sort_cost_T = 2 * P_T * (1 + ceil(log(ceil(P_T / B), B-1)))
sort_cost_RS = 2 * P_RS * (1 + ceil(log(ceil(P_RS / B), B-1)))

IO_Cost_SMJ_1 = sort_cost_R + sort_cost_S + P_R + P_S + P_RS + sort_cost_T + sort_cost_RS + P_RS + P_T +

# Sort-Merge Join 2
sort_cost_ST = 2 * P_ST * (1 + ceil(log(ceil(B / 2), 2)))

IO_Cost_SMJ_2 = sort_cost_T + sort_cost_S + P_T + P_S + P_ST + sort_cost_ST + sort_cost_R + P_ST + P_R +

# Block Nested Loop Join 1
IO_Cost_BNLJ_1 = P_R + ceil(P_R / (B - 2)) * P_S + P_RS + P_RS + ceil(P_RS / (B - 2)) * P_T + P_RST

# Block Nested Loop Join 2
IO_Cost_BNLJ_2 = P_S + ceil(P_S / (B - 2)) * P_T + P_ST + P_R + ceil(P_R / (B - 2)) * P_ST + P_RST

IO_Cost_HJ_1, IO_Cost_HJ_2, IO_Cost_SMJ_1, IO_Cost_SMJ_2, IO_Cost_BNLJ_1, IO_Cost_BNLJ_2
```

Out[173…

```
(3555, 4455, 7805, 10155, 2285, 8735)
```

Part (b)

For the query plan where $join1 = R(a,b), S(b,c)$ and $join2 = join1(a,b,c), T(c,d)$ find a configuration where using SMJ for $join1$ and HJ for $join2$ is cheaper than HJ for $join1$ and SMJ for $join2$. The output sizes you choose for P_RST and P_RS must be non-zero and feasible (e.g. the maximum output size of $join1$ is P_R*P_S).

[8 points]

In [174…

```python
P_R = 10
P_S = 100
P_T = 1000
P_RS = 25
P_RST = 125
B = 16

# Define sorting costs for SMJ
sort_cost_R = 2 * P_R * (1 + ceil(log(ceil(P_R / B), B-1)))
sort_cost_S = 2 * P_S * (1 + ceil(log(ceil(P_S / B), B-1)))
sort_cost_T = 2 * P_T * (1 + ceil(log(ceil(P_T / B), B-1)))
sort_cost_RS = 2 * P_RS * (1 + ceil(log(ceil(P_RS / B), B-1)))


HJ_IO_Cost_join1 = 3*(P_R+P_S)+P_RS
SMJ_IO_Cost_join2 = sort_cost_RS+sort_cost_T+P_RS+P_T+P_RST



SMJ_IO_Cost_join1 = sort_cost_R+sort_cost_S+P_R+P_S+P_RS
HJ_IO_Cost_join2 = 3*(P_RS+P_T)+P_RST

HJ_IO_Cost_join1 ,SMJ_IO_Cost_join2,SMJ_IO_Cost_join1,HJ_IO_Cost_join2
```

Out[174…

```
(355, 7250, 555, 3200)
```

since 555+3200<355+7250, so it's clear that using SMJ for $join1$ and HJ for $join2$ is cheaper than HJ for $join1$ and SMJ for $join2$.

Problem 3: Sequential Flooding

[10 points total]

Note: Before doing this question, it is highly recommended that you go through Activity 15, which covers eviction policies for buffer managers such as LRU, and why *sequential flooding* can sometimes occurs with LRU.

In the activity accompanying Lecture, we saw something called *sequential flooding* that can occur when a default eviction policy (for example LRU) is used by the buffer manager. We saw that we can achieve much lower IO cost by using a different eviction policy, MRU ("most recently used").

Note that "Most recently used" means most recently accessed, either from buffer or disk, consistent with what we showed in Activity-15.

For this problem, we will take a closer look at the IO cost of different eviction policies when reading the pages of a file sequentially multiple times.

Part (a)

Part (a.i)

[1 point]

Write a python function `lru_cost(N,M,B)` that computes the IO cost of the LRU eviction policy when reading in all the papges of an N-page file sequentially, M times, using a bugger with $B + 1$ pages. Assume that after reading the files, you don't need to write them out (you can just release them, so there is no write IO cost).

In [175…

```python
def lru_cost(N, M, B):
    buffer_capacity = B + 1
    if N > buffer_capacity:
        # If pages exceed buffer capacity, every page reloads each time it's accessed.
        return M * N
    else:
        # If all pages fit within the buffer, they're only loaded once.
        return N
```

Part (a.ii)

[2 points]

Write a python function `mru_cost(N,M,B)` that computes the IO cost of the MRU eviction policy when reading in all the papges of an N-page file sequentially, M times, using a bugger with $B + 1$ pages. Assume that after reading the files, you don't need to write them out (you can just release them, so there is no write IO cost).

In [176…

```python
def mru_cost(N, M, B):
    if N > B + 1:
        buffer = []
        total_cost = 0
        current_position = 0
        for m in range(M):
            for n in range(N):
                if n not in buffer:
                    if len(buffer) < B + 1:
                        buffer.append(n)
                    else:
                        buffer[current_position] = n
                    total_cost += 1
                current_position = buffer.index(n)
        return total_cost
    else:
```

```python
        # If all pages fit within the buffer, they're loaded only once.
        return N
```

Part (a.iii)

[2 points]

Now that you have written these functions, provide the tuples which generate the plot of **M vs. the absolute value of the difference between LRU and MRU in terms of IO cost** for $B = 4$, $N = 7$, and M between 1 and 20 inclusive (saved as the variable `p3_lru_points`)

In [177…

```python
B = 4
N = 7
M = 20

# Provide a list of tuple (m, difference between LRU and MRU in terms of IO cost) here:
p3_lru_points = []
for M in range(1, 21):
    lru = lru_cost(N, M, B)
    mru = mru_cost(N, M, B)
    p3_lru_points.append((M,abs(lru - mru)))

p3_lru_points
```

Out[177…

```
[(1, 0),
 (2, 5),
 (3, 10),
 (4, 15),
 (5, 20),
 (6, 24),
 (7, 28),
 (8, 33),
 (9, 38),
 (10, 43),
 (11, 48),
 (12, 52),
 (13, 56),
 (14, 61),
 (15, 66),
 (16, 71),
 (17, 76),
 (18, 80),
 (19, 84),
 (20, 89)]
```

Again, you can optionally plot your answer to check that it seems reasonable- starter code for doing this in the notebook below:

Part (b)

Recall that the LRU eviction policy removes the least recently used page when the buffer is full and a new page is referenced which is not there in buffer. The basic idea behind LRU is that you timestamp your buffer elements, and use the timestamps to decide when to evict elements. Doing so efficiently, requires some serious book-keeping, this is why in practice many buffer managers try to approximate LRU with other eviction policies that are easier to implement.

Here we will focus on the *CLOCK* or *Second Chance* policy. In the CLOCK eviction policy, the candidate pages for removal are considered left-to-right in a circular manner(with wraparound), and a page that has been accessed between consecutive considerations will not be replaced. The page replaced is the one that - considered in a circular manner - has not been accessed since its last consideration.

In more details the CLOCK policy proceeds maintains a circular list of pages in the buffer and uses an additional *clock (or second chance) bit* for each page to track how often a page is accessed. The bit is set to 1 whenever a page is referenced. When clock needs to read in a new page in the buffer, it sweeps over existing pages in the buffer looking for one with second chance bit set to 0. It basically replaces pages that have not been referenced for one complete revolution of the clock.

A high-level implementation of clock:

1. Associate a "second chance" bit with each page in the buffer. Initialize all bits to ZERO (0).
2. Each time a page is referenced in the buffer, set the "second chance" bit to ONE (1). this will give the page a second chance...
3. A new page read into a buffer page has the second chance bit set to ZERO (0).
4. When you need to find a page for removal, look in left-to-right in a circular manner(with wraparound) in the buffer pages:
    - If the second chance bit is ONE, reset its second chance bit (to ZERO) and continue.
    - If the second chance bit is ZERO, replace the page in the buffer.

You can find more details on CLOCK here.

Part (b.i)

[4 points]

Write a python function `clock_cost(N,M,B)` that computes the IO cost of the CLOCK eviction policy when reading in all the papges of an N-page file sequentially, M times, using a bugger with $B + 1$ pages. Assume that after reading the files, you don't need to write them out (you can just release them, so there is no write IO cost).

In [178…

```
def clock_cost(N, M, B):
    if N>B+1:
        return M*N
    else:
        return N
```

Part (b.ii)

[1 point]

Now that you have written the CLOCK cost function, provide the tuples which generate the plot of **M vs. the absolute value of the difference between LRU and CLOCK in terms of IO cost** for $B = 4$, $N = 7$, and M between 1 and 20 inclusive (saved as the variable `p3_clock_points`).

In [179…

```
B = 4
N = 7
M = 20

p3_clock_points = []
for M in range(1, 21):
    lru = lru_cost(7, M, 4)
    clock = clock_cost(7, M, 4)
    p3_clock_points.append((M,abs(lru - clock)))

p3_clock_points
```

Out[179…

```
[(1, 0),
 (2, 0),
 (3, 0),
 (4, 0),
 (5, 0),
 (6, 0),
 (7, 0),
 (8, 0),
 (9, 0),
 (10, 0),
 (11, 0),
 (12, 0),
 (13, 0),
 (14, 0),
 (15, 0),
 (16, 0),
 (17, 0),
 (18, 0),
 (19, 0),
 (20, 0)]
```

Does the CLOCK eviction policy prevent sequential flooding? How does it perform against LRU? Write a short explanation in the field below.

EXPLANATION GOES HERE

In the case of strict sequential access of more pages than the buffer can hold, neither CLOCK nor LRU prevents sequential flooding, they perform the same as each other, as the buffer is too small to hold all pages, and each page is evicted before being accessed again.

Problem 4: Hash Join Madden

[10 points total]

The NFL season has started strong and Jack Del Rio (The Oakland Raider's coach) wants to find out if Joe Flacco is an elite quarterback. He wants to do this by being more of a sabermetrics guy than a numbers guy. As a first step in doing this he wants to find out which are the colleges each NFL teams prefers drafting players from. We have access to two tables: (i) a table named "teams" which contains (team, player) pairs, and (ii) a table named "colleges" which contains (player, college) pairs. Being all excited about databases you decide that there is no other way but to join the two tables and get the desired results. However, you have no access to a database. Not even a challenge for you who decide to implement your favorite join algorithm on your own. And of course HASH JOIN is the way to go!!!

Load and explore the data

The two tables are stored in files which can be loaded into memory as two lists of **named tuples** using the code below:

In [180…

```
# Load data
import nfl
from nfl import *
teams, colleges = loadData()
```

Named tuples are basically lightweight object types and instances of named tuple instances can be referenced using object like variable deferencing or the standard tuple syntax. The following code prints the first 10 tuples from teams and colleges. *Notice how fields of named tuples are accessed inside the loops.*

In [181…

```
# Print List Entries
print ('Table teams contains %d entries in total' % len(teams))
print ('Table colleges contains %d entries in total' % len(colleges))
print
print ('First 10 entries in teams table')
for i in range(10):
```

```
    team = teams[i]
    print ('Entry %d' %(i+1),':',team.teamname, '|', team.playername)
print
print ('First 10 entries in college table')
for i in range(10):
    college = colleges[i]
    print ('Entry %d' %(i+1),':',college.collegename, '|', college.playername)
```

```
Table teams contains 12720 entries in total
Table colleges contains 12720 entries in total
First 10 entries in teams table
Entry 1 : Houston Texans | Jadeveon Clowney
Entry 2 : St. Louis Rams | Greg Robinson
Entry 3 : Jacksonville Jaguars | Blake Bortles
Entry 4 : Buffalo Bills | Sammy Watkins
Entry 5 : Oakland Raiders | Khalil Mack
Entry 6 : Atlanta Falcons | Jake Matthews
Entry 7 : Tampa Bay Buccaneers | Mike Evans
Entry 8 : Cleveland Browns | Justin Gilbert
Entry 9 : Minnesota Vikings | Anthony Barr
Entry 10 : Detroit Lions | Eric Ebron
First 10 entries in college table
Entry 1 : South Carolina | Jadeveon Clowney
Entry 2 : Auburn | Greg Robinson
Entry 3 : UCF | Blake Bortles
Entry 4 : Clemson | Sammy Watkins
Entry 5 : Buffalo | Khalil Mack
Entry 6 : Texas A&M | Jake Matthews
Entry 7 : Texas A&M | Mike Evans
Entry 8 : Oklahoma State | Justin Gilbert
Entry 9 : UCLA | Anthony Barr
Entry 10 : North Carolina | Eric Ebron
```

Down to business

During the lectures we saw that hash joins consist of two phases: The **Partition Phase** where using a hash function h we split the two tables we want to join into B buckets, and the **Matching Phase** where we iterate over each bucket and join the tuples from the two tables that match. Here you will need to implement a hash join in memory.

You are determined to implement the most efficient hash join possible! This is why you decide to implement your own hash function that will uniformly partition the entries of a table across B buckets so that all buckets have roughly the same number of entries. You decide to use the following hash function:

In [182…

```
# Define hash function
def h(x,buckets):
    rawKey = ord(x[1])
    return rawKey % buckets
```

You use this hash function to partition the tables. To do so you can use the helper method `partitionTable(table,hashfunction,buckets)` for convenience as shown next:

In [183…

```
# Fix the number of buckets to 500
buckets = 500
# Partition the teams table using hash function h
teamsPartition = partitionTable(teams,h,buckets)
```

The output of `partitionTable()` is a dictionary with its keys corresponding to bucket numbers in $[0, B-1]$ and its entries to lists of named tuples.

Part (a)

Part (a.i)

[4 points]

It's now time to implement your own hash join! You only need to implement the merge phase of the hash join. The output of the method should correspond to the result of a join between teams and colleges over the ***playername*** attribute. The partition phase is implemented. You need to fill in the merge phase.

Note: You should only use the two dictionaries t1Partition and t1Partition provide. No other data structures are allowed.

In [184…

```
def hashJoin(table1, table2, hashfunction,buckets):
    # Parition phase
    t1Partition = partitionTable(table1,hashfunction,buckets)
    t2Partition = partitionTable(table2,hashfunction,buckets)
    # Merge phase
    result = []
    for bucket_index in range(buckets):
        bucket1 = t1Partition[bucket_index]
        bucket2 = t2Partition[bucket_index]
        for t1Entry in bucket1:
            for t2Entry in bucket2:
                if t1Entry.playername == t2Entry.playername:
                    result.append((t1Entry.teamname, t1Entry.playername, t2Entry.collegename))

    # ANSWER GOES HERE

    # To populate your output you should use the following code(t1Entry and t2Entry are possible var nam
    # result.append((t1Entry.teamname, t1Entry.playername, t2Entry.collegename))
    return result
```

Part (a.ii)

[1 point]

It time to evaluate your algorithm! The code provided below executes the join between teams and colleges and measures the total execution time. What is the total number of entries output by your algorithm?

Does the runtime of your algorithm seem reasonable? Provide a brief explanation.

In [185…

```
import time
start_time = time.time()
res1 = hashJoin(teams, colleges, h, buckets)
end_time = time.time()
duration = (end_time - start_time)*1000 #in ms
print ('The join took %0.2f ms and returned %d tuples in total' % (duration,len(res1)))

# EXPLANATION GOES HERE
```

```
The join took 1951.16 ms and returned 12740 tuples in total
```

Part (b)

You decide to investigate the performance of `hashJoin( )` further. Since you implemented the merge phase of `hashJoin( )` yourself you focus on the partitioning obtained by using the provided hash function `h( )`. In the lectures we saw that a good hash function should partition entries uniformly across buckets. We will now check if `h( )` is indeed a good function.

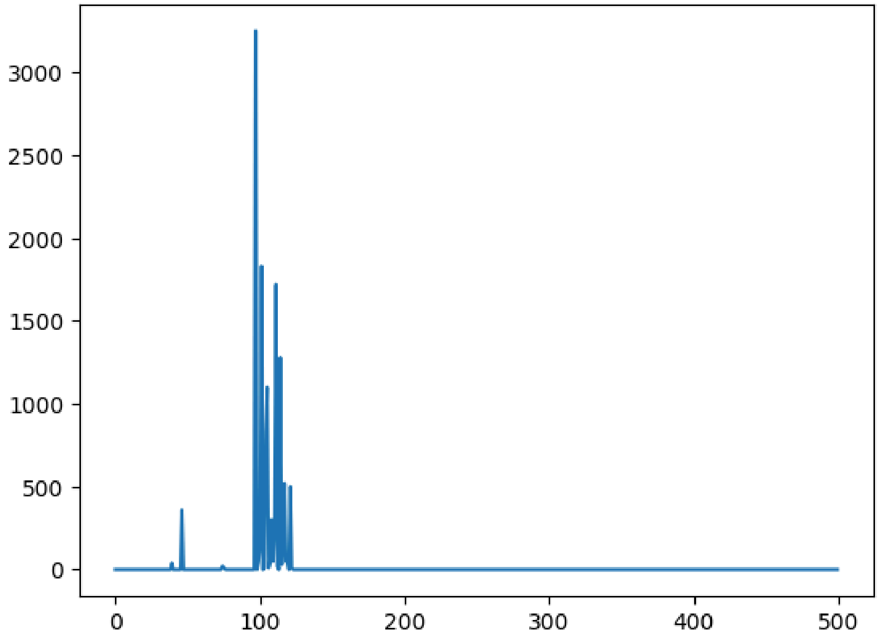
The following code generates a histogram of the bucket sizes for table teams (using the above hash function `h` and 500 buckets) to help figure out what is going wrong.

In [186…

```
# Examine if this is a good partition function
def histogramPoints(partition):
    ids = range(buckets)
    items = []
    for i in range(buckets):
        if i in partition:
            items.append(len(partition[i]))
        else:
            items.append(0)
    return ids, items

%matplotlib inline
import matplotlib.pyplot as plt
```

```
# Plot bucket histogram
buckets = 500
teamsPartition = partitionTable(teams,h,buckets)
ids, counts = histogramPoints(teamsPartition)
plt.plot(ids, counts)
plt.plot()
```

Out[186… []



My algorithm generated 12,740 entries in total, with notable redundancy. The runtime performance of the algorithm is notably suboptimal, exhibiting excessive slowness. This seems to be caused by the non-uniform distribution of outputs from the hash function.

Part (b.i)

[3 points]

Now find the skew associated with the above histogram. Skew is defined as the standard deviation of the number of entries in the buckets. A uniform hash function produces buckets of equal size, leading to 0 skew, but our candidate hash function h is imperfect so you should observe a positive skew.

In [187…

```
# ANSWER
# partition- a table partition as returned by method partitionTable
# return value - a float representing the skew of hash function (i.e. stdev of chefs assigned to each re:
# def calculateSkew(partition):
#     # ANSWER STARTS HERE
#     skew =
#     # ANSWER ENDS HERE
#     return skew

import statistics

def calculateSkew(partition):
    # Collect the number of entries in each bucket into a list
    num_entries = [len(bucket) for bucket in partition.values()]

    # Calculate the standard deviation of the number of entries in the buckets
    skew = statistics.stdev(num_entries)

    return skew


print (calculateSkew(teamsPartition))
```

```
205.03777060519727
```

Part (b.ii)

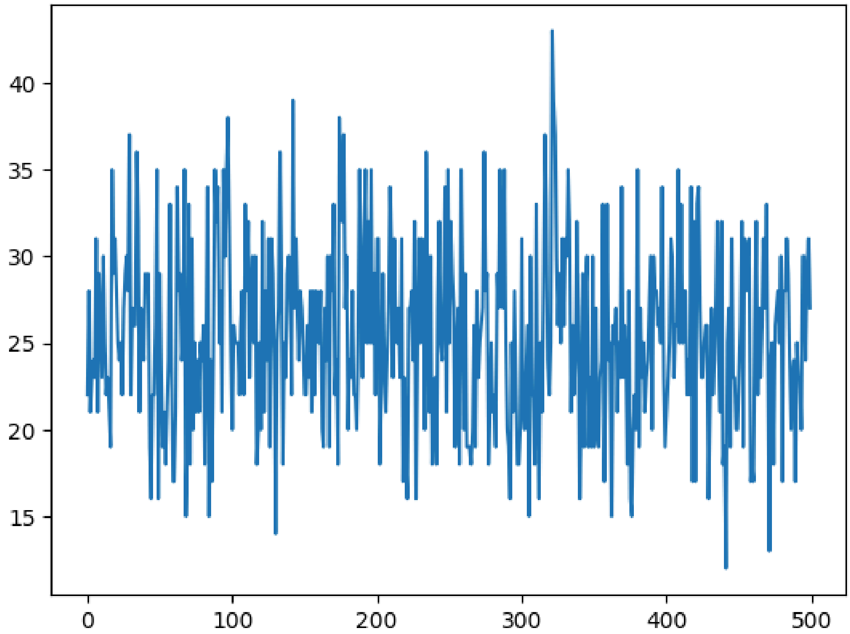
[1 point]

Use python's hash function to see if you can produce a better (aka smaller) runtime for hash join. As at the beginning of part b, make a histogram of the bucket sizes (this time using the new hash function and 500 buckets). You can plot your histogram using the same code provided above.

In [188…

```
# Design a better hash function and print the skew difference for
def hBetter(x,buckets):
    rawKey = hash(x)
    return rawKey % buckets

# Plot bucket histogram
buckets = 500
teamsPartition = partitionTable(teams,hBetter,buckets)
ids, counts = histogramPoints(teamsPartition)
plt.plot(ids, counts)
plt.plot()
```

Out[188… []



Part (b.iii)

[1 point]

Rerun your hash join algorithm with the new hash function you designed and 500 buckets. Does the algorithm run faster? If so what is the speed-up you are observing?

In [189…

```
start_time = time.time()

res1 = hashJoin(teams, colleges, hBetter, buckets)

end_time = time.time()
duration = (end_time - start_time)*1000 #in ms
print ('The join took %0.2f ms and returned %d tuples in total' % (duration,len(res1)))

# WRITE DOWN THE SPEED UP
```

```
The join took 62.58 ms and returned 12740 tuples in total
```

The speedup achieved here is significant, with the runtime reduced to about 60 ms from about 2000 ms, it's over 30 times faster than before. This improvement in algorithm performance can be attributed to the enhancement of the hash function.