CS 145 PS3

November 18, 2015

Instructions / Notes:

- Using the IPython version of this problem set is **strongly recommended**, however you can use only this PDF to do the assignment, or replicate the functionality of the IPython version by using this PDF + your own python interface

- See Piazza for submission instructions

- Have fun!

1 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** we saw in *Lecture 13*.

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 on the physical disk) is generally much faster that *random access IO* (any reading / writing that is not sequential).

In other words, for example, if we read 3 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 (assuming that the file layouts represent the actual physical layout on disk, as we will). Conceptually, recall the image of the disk head having to move around the physical disk, versus just staying in place and reading sequentially as the disk spins.

In this problem, we will begin to model this, by assuming that sequential READS are "free", i.e. the cost of N sequential reads is 1 IO for any $N > 0$.

Note that we do not make this assumption for writes, i.e. writing N pages sequentially still costs N IO (this is to simplify our calculations below).

OTHER IMPORTANT NOTES

- **NO REPACKING:** Consider the external merge sort algorithm using the basic optimizations we present in section 1 of lecture 13, but do not use the repacking optimization (Lecture 13 : Slide 28 and on)

- **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) in lecture.

- **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

- **The pages of files and runs are arranged sequentially on the physical disk** i.e. you can assume that sequential reads from a run / file are always sequential!

1.1 Part (a)

5 points Consider a modification of the external merge sort algorithm where **during the merge phase, runs are always read in 3-page chunks (i.e. 3 pages sequentially at a time).** Calculate the cost of performing this version of external merge sort for a setup having $B+1 = 10$ buffer pages and an unsorted input file with 90 pages, showing the steps of your work.

Make sure to explain your reasoning in 2-3 sentences for full / partial credit. Write your explanations as python comments, and save your final IO cost as pla_cost.

1.2 Part (b)

15 points Next, you'll generalize the reasoning above by writing a python function that computes the cost of performing this version of external merge sort for a setup having $B + 1$ buffer pages, an unsorted input file with N pages, and where **we always read in P-page chunks during the merge phase**.

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 (plus the cost of initially sorting the runs).

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)

1.2.1 Part (b.i)

5 points First, write a python function that computes the total IO cost to create the initial runs. The function should look as such:

```
def cost_initial_runs(B, N, P):
    # YOUR CODE HERE
```

1.2.2 Part (b.ii)

5 points Next, write a python function that computes the total IO cost to read in and then write out all the data during *one* merge pass; note that, as in lecture, this should apply for any pass during our merge phase. The function should look as such:

```
def cost_per_pass(B, N, P):
    # YOUR CODE HERE
```

1.2.3 Part (b.iii)

5 points Next, write a python function that computes the total number of passes we'll need to do. The function should look as such:

```
def num_passes(B, N, P):
    # YOUR CODE HERE
```

Finally, our total cost function is:

```
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)
```

1.3 Part (c)

5 points For $B = 29$ and $N = 900$, find the optimal P according to your IO cost equation above. Return both the optimal P value (stored as 'P_opt') and the list of tuples **for feasible* values of** P that would generate a plot, at resolution $= 1$, stored as 'points', using the starter code below.

*By *feasible values*, we mean values of P for which the algorithm can run; this should include values for which the divisibility assumptions in part (b) do not hold (those were just to make your cost equation in that part simpler!)

Additionally, plot your points *using some plotting software** and describe what you see in one sentence (stored as a python comment).

*Below we provide starter code for using 'matplotlib' in the notebook, if you were able to install this; however any other software that allows you to visualize the plot (Excel, Google spreadsheets, MATLAB, etc) is fine!

```
# Save the tuples to generate an IO cost plot vs. P here
B = 29
N = 900
feasible_p_range = # FILL THIS IN
p1_points = [(p, external_merge_sort_cost(B, N, p)) for p in feasible_p_range]

# Save the optimal value here
P_opt =

# ONE SENTENCE OF EXPLANATION HERE
```

Below, we provide *optional* starter code for plotting with the 'matplotlib' library:

```
# Shell code for plotting in matplotlib
%matplotlib inline
import matplotlib.pyplot as plt

# Plot: unpack the list of p,cost tuples and input as args to plot
plt.plot(*zip(*p1_points))
plt.show()
```

Additionally, below is a plot of P **values that meet the divisibility assumptions** (i.e. ones that your function might be evaluated on in 1(b)) that you can check your answer against!
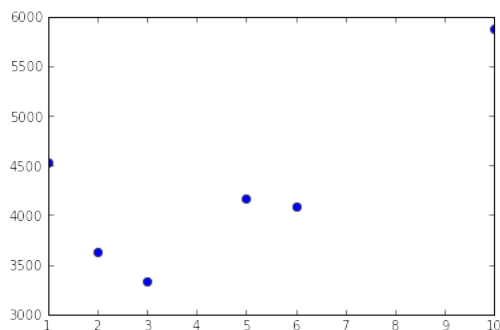
Also, the actual points of the above plot:

```
eval_correct = [(1, 4530.0), (2, 3630.0), (3, 3330.0), (5, 4170.0), (6, 4080.0), (10, 5880.0
```

2 Problem 2: IO Cost Model

15 points total.

In each part of this problem we consider two different join algorithms, joining relations $R(A, B)$ and $R(A, C)$, and you must provide an example scenario where one would be faster than the other. Additionally, you will compute the total IO cost for each join algorithm in your example.

Specifically, for each part, you need to provide:

- <PART>_P_R: Number of pages of R

- <PART>_P_S: Number of pages of S

- <PART >_B: Number of pages in buffer

- <PART>_IO_cost_1: Total IO cost of first join algorithm

- <PART>_IO_cost_2: Total IO cost of second join algorithm

Also include a one sentence explanation for each part (as a python comment)
Notes:

- **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!**

- **Do not include the IO cost of writing the final output (OUT in lecture) in any of your calculations (since OUT is the same, independent of which join algorithm we use!)**

2.1 Part (a)

5 points. Hash Join (HJ) is cheaper than Sort-Merge Join (SMJ) to join R, S

```
A_P_R =
A_P_S =
A_B =
A_IO_cost_1 =
A_IO_cost_2 =

# Explanation:
```

2.2 Part (b)

5 points. HJ is cheaper than BNLJ to join R, S

```
B_P_R =
B_P_S =
B_B =
B_IO_cost_1 =
B_IO_cost_2 =

# Explanation:
```

2.3 Part (c)

5 points. BNLJ is cheaper than SMJ to join R, S

```
C_P_R =
C_P_S =
C_B =
C_IO_cost_1 =
C_IO_cost_2 =

# Explanation:
```

3 Problem 3: Sequential Flooding

10 points total. In the activity we went through in class for Lecture 15, 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 acheive 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 in lecture.

For this problem, specify the IO cost of reading in all the pages of an N-page file sequentially, M times, using a buffer with $B + 1$ pages, using an LRU and MRU eviction policy. 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).

Write an explanation and also create two python functions, 'lru_cost(N,M,B)' and 'mru_cost(N,M,B)'. Finally, provide the tuples which generate the plot of **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_points')

```
def lru_cost(N, M, B):
    # YOUR CODE HERE

def mru_cost(N, M, B):
    # YOUR CODE HERE

# ONE SENTENCE OF EXPLANATION HERE

B = 4
N = 7
M = 20
p3_points = [(m, abs(lru_cost(N, m, B) - mru_cost(N, m, B))) for m in range(1, M+1)]
```

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

```
# Shell code for plotting in matplotlib
%matplotlib inline
import matplotlib.pyplot as plt

# Plot
plt.plot(*zip(*p3_points))
plt.show()
```

4 Problem 4: Don't Drink and Hash Collide

10 points **Imagine the following scenario:** You are the chief culinary training officer of a high-end nation-wide restaurant chain, and you are overwhelmed because today is the end of your latest training program, and you have to assign 10, 000 newly-minted chefs each to one of your chain's 100 restaurants.

You decide that to do this both fairly and uniformly, you'll create a hash function, which takes in a chef's ID and *uniformly* outputs a number between 0 and 99. The chef ids are each the letter "C" followed by four digits, zero-padded, so that ids look like:

- C0000

- C0001

- C0002

- ...

- C9998

- C9999

Each chef has a unique ID. You decide to try the hash function:

```
def h(x, n):
    return sum(map(ord, x)) % n
```

If this hash function works properly, each bucket / restaurant should have ≈ 100 new chefs assigned to it.

Furthermore, as often does in fact happen in practice (with actual algorithms... roughly), the speed of your restaurant chain is proportional to the *sum of the squared number of chefs in each restaurant (i.e. the "bucket sizes")*. In other words, if in restaurants '1, 2, 3, ...' there are '100, 150, 92, ...' chefs respectively, then your restaurant chain's overall speed will be $\propto 100^2 + 150^2 + 92^2 + ...$

Intuitively, at a rough level, this is saying that too many chefs in one kitchen slows things down greatly...

You expect your hash function to distribute the chefs uniformly though, so you don't expect this to be an issue. However, your restaurant chain ends up running much more slowly than expected!! You decide to try to figure out why...

4.1 Part (a)

9 points

4.1.1 Part (a.i)

3 points Make a histogram of the bucket sizes (using the above hash function h) to help figure out what is going wrong. You can plot your histogram with 'matplotlib' if you have it installed, but all that we require is the list of tuples that would make the histogram (the bucket / restaurant ID, number of chefs assigned pairs), saved as 'points'.

```
# C- total number of chefs (10000 in our example)
# R- number of restaurants (100 in our example)
# return value - list of tuples representing (restaurant ID, number of chefs assigned
# to that restaurant) points
def histogramPoints(C,R):
```

```
    points = []
    # Start code here

    # End code here
    return points

# Shell code for plotting in matplotlib (don't need to complete or turn in)
%matplotlib inline
import matplotlib.pyplot as plt
chefPoints = histogramPoints(10000, 100)

ids = [point[0] for point in chefPoints]
chefCounts = [point[1] for point in chefPoints]

# Plot
plt.plot(ids, chefCounts)
plt.show()
```

4.1.2 Part (a.ii)

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.

```
# C- total number of chefs (10000 in our example)
# R- number of restaurants (100 in our example)
# return value - a float representing the skew of hash function (i.e. stdev of
# chefs assigned to each restaurant)
def calculateSkew(C,R):
    # Start code here

    # End code here

skew = calculateSkew(10000,100)
```

4.1.3 Part (a.iii)

3 points Suppose that your application is quadratic in the number of items in a bucket, i.e. if your buckets have sizes 112,150,80,... your runtime will be $112^2+150^2+80^2$,.... Compute how much larger your running time is than what you'd expect if the buckets were in fact uniform.

Note: you may approximate by assuming that $C\%R == 0$ here, if this is helpful

```
# C- total number of chefs (10000 in our example)
# R- number of restaurants (100 in our example)
# return value - difference between real runtime (with some skew) and ideal (no skew)
def runtimeDiff(C,R):
    # Start code here

    # End code here
```

4.2 Part(b)

1 point Use python's hash function to see if you can produce a better (aka smaller) runtime. As in part (a), make a histogram of the bucket sizes (this time using the new hash function). You can plot your histogram with 'matplotlib' if you have it installed, but all that we require is the list of tuples that would make the histogram (the bucket / restaurant ID, number of chefs assigned pairs), saved as 'points'. Hint: This solution should look *very* similar to (a.i)

```
# C- total number of chefs (10000 in our example)
# R- number of restaurants (100 in our example)
# return value - list of tuples representing (restaurant ID, number of chefs
# assigned to that restaurant) points
def betterHistogramPoints(C,R):
    points = []
    # Start code here

    # End code here
    return points


# Shell code for plotting in matplotlib (don't need to complete or turn in)
%matplotlib inline
import matplotlib.pyplot as plt
betterChefPoints = betterHistogramPoints(10000, 100)

betterIds = [point[0] for point in betterChefPoints]
betterChefCounts = [point[1] for point in betterChefPoints]

# Plot
plt.plot(betterIds, betterChefCounts)
plt.show()
```

5 Bonus Problem: Hash Away

5 points total

Design a better hash function that partitions the chefs more uniformly. Return the tuples of the new histogram (again, plot if you want to), and the new runtime difference as in the previous part. Note that there is no *best* solution here- feel free to be creative, just make sure you see an improvement of some sort!

Hint: Feel free to check out other hash functions out on the internet -e.g. Java's, python's- for inspiration

```
# Your better hash function
def bonusH(x, n):
    # Start code here

    # End code here


# Histogram points based off of your better hash function
def bonusHistogramPoints(C,R):
    points = []
    # Start code here

    # End code here
```

```python
    return points

# Difference bewteen the runtime with your hash function and the idea runtime
# (with no skew)
def bonusRuntimeDiff(C,R):
    # Start code here

    # End code here
```