HW4-IO-HANDOUT (5)

October 22, 2021

1 Problem Set #4

1.1 CSCI 3287

======= ### Instructions / Notes:

Read these carefully

There are three problems. #1 is 40 points, #2 is 35 points and #3 is 25 points. Each subproblem indicates the number of points. Although part #1 is the most points, it's actually the easiest problem – but the solution of #2 depends on being careful and understanding #1 completely.

- **Please read all the points of the "Notes" sections- they're important for this PS!!!**
- You **are not required to do any plotting in this PS- only in certain problems to provide the tuples that would generate a plot.** You can then optionally plot (in the notebook with matplotlib)
- 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!*

1.2 Problem 1: Double Trouble

[15 + 15 + 10 = 40 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.

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 3/4 sequential *READS* are "free", i.e. the total cost of 4 sequential reads is 1 IO. * We will also assume that the writes are always twice as expensive as a read. Sequential writes are never free, therefore the cost of N writes is always $2N$.

1.2.1 Other important notes:

- **NO REPACKING:** Consider the external merge sort algorithm using the basic optimizations we present in lecture, but do not use the repacking optimization covered in lecture.
- **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. In Python, you can use `math.ceil()` to compute the ceiling (next largest integer).
- **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

[1]:
```
import math
```

1.2.2 Part (a) [14 pts total]

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

[2]:
```
N = 160
B = 19
```

[3]:
```
### BEGIN SOLUTION
# Provide a description of your solution as comments (BELOW)
io_split_sort = 360
io_split_sort
### END SOLUTION
```

[3]:
```
360
```

Description of solution to a.i

First, we note the problem assumptions: * #1. The cost of a read (r) is N, while the cost of a write (w) is $2N$ #2. A B way merge requires B+1 buffer pages * #3. No optimization with repacking.

Specific Assumptions for this problem: * #1. Reads are in 4 page sequential chunks. Important assumption: Writes are in 1 page chunks (not explicity stated!) * #2. B+1=20 buffer pages * #3. The input file is 160 pages.

Calculations, explanations: * So, there are files of 160 pages * (1 pass/20 pages processed)= 8 runs required. * Each write costs 2*N, each read costs N, and this is unoptimized. **However,**

note an important distinction: Reads are always in 4 page chunks, while writes are always in 1 page chunks (I ASSUME).

- Each pass involves reading and writing out all the pages once each.
- However, for a write the cost of a 4 page chunk is 1/4 total cost, so there is 1*N IO operation per read.
- For a write, there are 2*N IO operations.
- Now, calculate the number of IO ops for a read:
- ((20 pages processed/run)*(1 read/4 page chunks))* (1 IO) = 5 IO/run
- Now, calculate the number of IO ops for a write, which should be much higher:
- (20 pages processed/run)* (1 write/1 page chunks) * (2 IO)=40 IO/run
- Therefore, the total I/O cost and splitting and sorting the files is:
- (40+5 IO/run)* 8 run=
- **360 IO ops.**

Part (a.ii) [3 pts] After the file is split and sorted, we can merge n runs into 1 using the merge process. What is the largest n we could have, given that reads are always read in 4-page chunks? Note: this is known as the *arity* of the merge.

[4]:
```
merge_arity = 4
merge_arity
```

[4]: 4

Description of solution to a.ii

- B+1=20, so B=19
- B, not B+1, must be used in the calcuation because one page is always reserved for output.
- As given in the problem description, reads are in 4 page chunks, an important detail.
- Buffers of size B could potentially reduce the overall cost a lot.

- However, the merging step is limited by the number of chunks that can be read into the buffer at a time, because of divisibility.
- That is, having 19 available buffers and reading in 4 page chunks means that only 4 runs at a time can be merged (4*4=16, 16<=19).
- The calculation for arity is: floor(B/c) where c=chunk size of reading.
- floor(19/4)=4
- **n= 4, the arity of the merge.**

Part (a.iii) [3 pts] How many passes of merging are required?

[5]:
```
merge_passes = 2
merge_passes
```

[5]: 2

Description of solution to a.iii * This is adapted from the unoptimized portion of the lecture, and combines logic from a.i and a.ii * Merging N pages requires ceil((Log$_n$(N/B+1)) total merge passes. * N=160, n=4, and B=19, * So: ceil((Log$_4$(160/20))= * 2 passes required.

- To explain further: The initial split and sort process (call it pass 0) results in N/(B+1)=160/20=
- 8 sorted runs of 20 pages each.
- From aii., we know that 4 runs can be merged at once because of the arity (divisibility) constraint of reads.
- So the forst merge pass results in 8/4=2 merged files of 80 pages each.
- Merge pass 2 results in 160 (N) pages, after the two groups of 80 are merged into one final file.
- Note that merging and sorting happens incrementally (fractionally), so that a large file can fit into a smaller buffer size, a key point.

Part (a.iv) [3 pts] What is the IO cost of the first pass of merging? Note: the highest arity merge should always be used.

[6]:
```
merge_pass_1 = 360
merge_pass_1
```

[6]: 360

Description of solution to a.iv

- The merge pass here is the process of both reading and writing each page once, with distinct IO costs for each.
- Recall that each read is done in 4 chunks, while each write may be assumed to be done in 1 chunk.
- Also, recall that the cost per write is twice that of the cost per read.
- The cost for the first pass write merge is therefore: 160 pages * (2 IO/page) = 320 IO ops.
- The cost for the first pass read merge is: 160 pages * (1 page/4 chunks) * (1 IO/page)= 40 IO ops.
- The total number of IO ops= 320 + 40=
- **360 IO Ops**

Part (a.v) [3 pts] 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.**

[7]:
```
total_io =1080
total_io
```

[7]: 1080

Description of solution to a.v

- The total IO cost of the external merge sort algorithm will be calculated as:

- The total IO ops of splitting and sorting + (IO ops of merging per pass * Number of passes necessary)=
- (360 IO + (360 IO/pass * 2 passes)=
- **1080 IO ops**
- To explain further: The total cost= The IO cost of split and sort (from a.i)=360 IOs
- +IO cost of first merge (360 IOs) * Number of merges (2) (from a.iv)
- So, Total IO = 360 + (360)*2 = 1080
- Note once again that the split and sort, and each merge pass must read and write every page once.

1.2.3 Part (b) [15 pts]

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

[8]:
```
def cost_initial_runs(B, N, P):
    #Reading the files in: N pages and P chunks available each time (formerly␣
 ↪4).
    #=floor(N/P) accounts for the worst case where we round down for read size
    #Writing the files incurs a cost of 2*N, as described above.
    #Take the ceil of the whole thing to account for possible worst case.
    return math.ceil(math.floor(N/P)+(2*N))
```

Part (b.ii) [5 pts] 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:

[9]:
```
def cost_per_pass(B, N, P):
    #According to the logic from part a, b.i and b.ii should be the same IO␣
 ↪costs...
    return math.ceil(math.floor(N/P)+(2*N))
```

[10]:
```
#Explanation of b.ii

#N/P gives us the gives us the number of P page chunks that we can merge at␣
 ↪each run, as one output page is needed as before. This is the read cost.
#This must be floored, to B may not be evenly divisible by P.
#The write cost, one page at a time, is always 2*N, or 2*Read 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) [5 pts] Next, let's write a python function that computes the **exact** total number of passes we'll need to do

[11]:
```
def num_passes(B, N, P):
    #In this section the task is to reproduce the equation from a.iii

    #The log base should be B/P, as we merge max B pages/ P chunks at a time␣
 ↪because of divisibility.
    #floor(B/P) must be taken in case B%P!=0, and floor is used rather than␣
 ↪ceiling because it is worst case.
    #The argument of the logarithm, as in a.iii, is (N/B+1), as in lecture.
    #A ceil() must also be added to account for the final worst case.

    #so num_passes=ceil(log_(floor(B/P))(N/B+1))
    #the +1 above is for the initial pass

    #NOTE: syntax of math.log(a,Base)
    return math.ceil(math.log(N/(B+1), math.floor(B/P)))
```

Finally, our total cost function is:

[12]:
```
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)
```

1.2.4 Part (c) [5 points]

For $B + 1 = 100$ and $N = 900$, 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`:

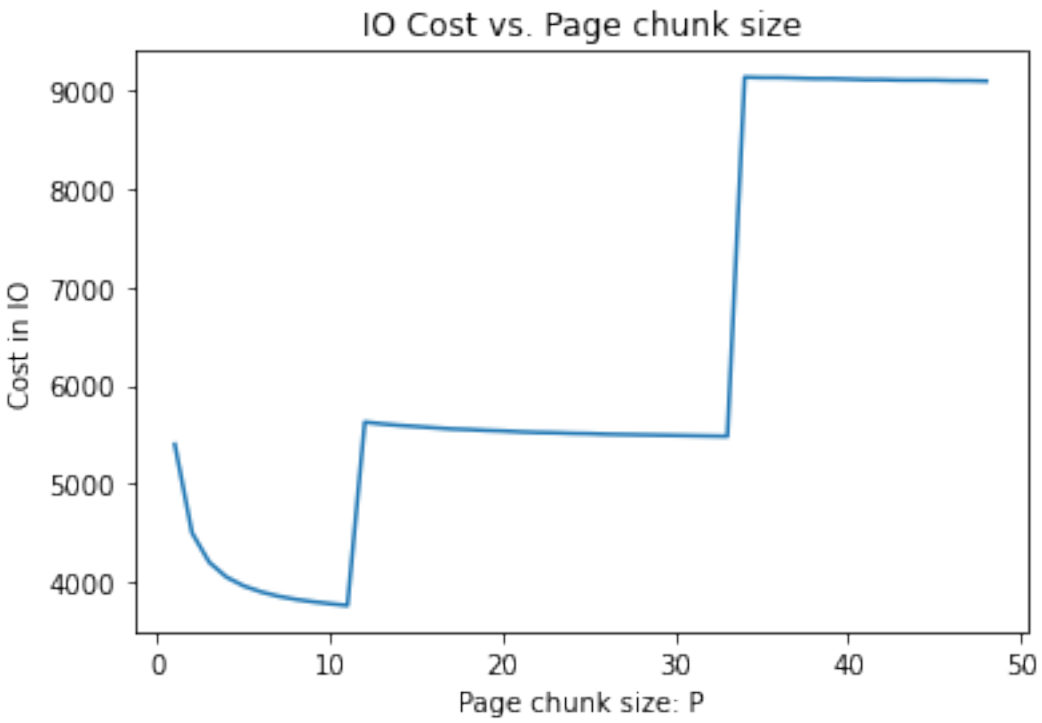
[13]:
```
# Save the optimal value here
P = 11

# Work Done below
```

[14]:
```
B=99
N=900
FeasiblePs=range(1,math.floor(B/2))
#Pythonic syntax-list comprehension for iteration to create list of tuples, as␣
 ↪read below.
points=[(p,external_merge_sort_cost(B,N,p)) for p in FeasiblePs]

print(points)
```

```
[(1, 5400), (2, 4500), (3, 4200), (4, 4050), (5, 3960), (6, 3900), (7, 3856),
(8, 3824), (9, 3800), (10, 3780), (11, 3762), (12, 5625), (13, 5607), (14,
5592), (15, 5580), (16, 5568), (17, 5556), (18, 5550), (19, 5541), (20, 5535),
(21, 5526), (22, 5520), (23, 5517), (24, 5511), (25, 5508), (26, 5502), (27,
5499), (28, 5496), (29, 5493), (30, 5490), (31, 5487), (32, 5484), (33, 5481),
(34, 9130), (35, 9125), (36, 9125), (37, 9120), (38, 9115), (39, 9115), (40,
9110), (41, 9105), (42, 9105), (43, 9100), (44, 9100), (45, 9100), (46, 9095),
(47, 9095), (48, 9090)]
```

Below we provide starter code for using `matplotlib` in the notebook, if you want to generate the graph of P vs. IO cost. Your value for `points` should be tuples of the form (P, io_cost)

[15]:
```
# Shell code for plotting in matplotlib
%matplotlib inline
import matplotlib.pyplot as plt

# Plot
plt.plot(*zip(*points))
plt.title("IO Cost vs. Page chunk size")
plt.ylabel("Cost in IO")
plt.xlabel("Page chunk size: P")
plt.show()
```



What is the optimal Page chunk size, P_{opt}? Explain the graph's shape and what happens at $P_{opt}+1$.

[16]:
```
# Optimal page chunk size
P_opt = 11

# EXPLAIN WHAT HAPPENS AT P_OPT+1 HERE


print('num passes with P_opt:', num_passes(B, N, P_opt))
print('num passes with P_opt_plus_1:', num_passes(B, N, P_opt + 1))
```

```
num passes with P_opt: 1
num passes with P_opt_plus_1: 2
```

Description of Graph

- Up until P(opt)=11, the IO cost falls in a U-shape, where only 1 pass of merging is required.
- At P(opt)+1=12, the cost in IOs greatly increases from the min of 3762 IOs.
- This is because at at P(opt)+1, the number of passes of merging requires becomes 2, with the increase in IOs associated.
- At P=34, the cost increases greatly again because the number of merging passes required jumps from 2 to 4.
- Finally, at increased P values the cost plateaus once again as the number of passes remains

constant.

- The optimal p value (P(opt)) is where the cost in IO is the lowest due to the least number of merge passes required.
- This occurs at exactly:
- **P(opt)=11.**

1.3 Problem 2: IO Cost Models [35 pts 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) RS
- `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 abbreviations for the joins used are Sort-Merge Join (SMJ), Hash Join (HJ), and Block Nested Loop Join (BNLJ). **

1.3.1 Part (a)

Given: * `P_R`: 10 * `P_S`: 100 * `P_T`: 1000 * `P_RS`: 50 * `P_ST`: 500 * `P_RST`: 250 * `B`: 32

Compute the IO cost for the query plans. Each query plan contains multiple joins. Include 1-2 sentences (as a python comment) above each answer explaining the performance for each algorithm/query plan. You can express your answer using the values `P_R` *etc* that are defined below and you can use your `external_merge_sort_cost` function to check or compute your results if you believe it's correct

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

[17]:
```
P_R = 10
P_S = 100
P_T = 1000
```

```
P_RS = 50
P_ST = 500
P_RST = 250
B = 32
```

1.3.2 IO_Cost_HJ_1 [5 pts]

where only hash join is used, $join1 = R(a,b), S(b,c)$ and $join2 = join1(a,b,c), T(c,d)$

[18]:
```
#The equation for this solution follows from the lecture complexity of HJ,
#Given enough buffer pages, then partitioning requires 2(P(R)+P(S)) IOs, where each page is read and written,
#While matching (with BLNJ) requires reading each page, or P(R)+P(S) IOs.
#Combining these, it requires 3(P(R)+P(S)) + OUT IOs generically.
#Or for join 1, IOs=3(P_R+P_S)+P_RS IOS
#So join 1=3(10+100)+50=380 IOs.
#join 2= 3((P_RS+P_T))+P_RST OUT
#So join 2 = 3*(50+1000)+250= 3780.
#Join1+Join2=380+3400=3780 IOs
#Equation with variables plugged in is:

#3*(P_R+P_S)+P_RS + 3*((P_RS+P_T))+P_RST
```

[19]:
```
# describe solution (ABOVE)
IO_Cost_HJ_1 = 3*(P_R+P_S)+P_RS + 3*((P_RS+P_T))+P_RST
IO_Cost_HJ_1
```

[19]:
```
3780
```

1.3.3 IO_Cost_HJ_2 [5 pts]

where only hash join is used, $join1 = T(c,d), S(b,c)$ and $join2 = join1(b,c,d), R(a,b)$

[20]:
```
#Plugging in values using the equation above,
#Join 1= 3*(P_S+P_T)+P_ST OUT= 3*(100+1000)+500=3800 IOs
#Join 2= 3*(P_ST+ P_R)+ P_RST OUT=3*(500+10)+250=1780 IOs
#Join1+Join2=3800+1780=5580 IOs

#equation with variables plugged in:

#3*(P_S+P_T)+P_ST + 3*(P_ST+ P_R)+ P_RST
```

[21]:
```
# describe solution (ABOVE)

IO_Cost_HJ_2 = 3*(P_S+P_T)+P_ST + 3*(P_ST+ P_R)+ P_RST
IO_Cost_HJ_2
```

[21]: 5580

1.3.4 IO_Cost_SMJ_1 [5 pts]

where only sort merge join is used, $join1 = R(a,b), S(b,c)$ and $join2 = join1(a,b,c), T(c,d)$

[22]:
```
#NOTE**********: It is assumed that B+1=33 is the max number of available␣
↪buffers.
#*****************************************************************************

#This logic will follow closely from the lecture for SMJ complexity.
#In lecture, it is given that the cost of SMJ is the cost of sorting + merging␣
↪+Output (P(R)(S) OUT) generically,
#And we know that the cost of sorting N pages with no repacking is:
#2N*(ceil(LogB(N/B+1))+1)

#A key insight is that P_R completely fits in the buffer, so it can be split␣
↪and sorted in just one pass.

#So, join 1, where P_R or P_S is exchanged for the N pages:
#P_R requires 1 pass to split and sort:
#Sort(P_R)=2*10=20 IOS
#P_S requires 2 Passes to split and sort:
#Sort(P_S)=2*100*(ceil(Log32(100/33))+1)=2*100*2=400 IOS
#Merge=P_S+P_R=100+10=110 IOS
#Total Cost join1=Sort(P_R)+Sort(P_S)+Merge+P_RS
#Total Cost join1=20+400+10+100+50=580 IOs

#Join 2:
#P_RS requires 2 passes to split and sort:
#Sort(P_RS)=2*50*(ceil(Log32(50/33))+1)=2*50*2=200 IOS
#P_T requires 3 passes to split and sort:
#Sort(P_T)=2*1000*(ceil(Log32(1000/33))+1)=2*1000*3=6000 IOS
#Merge=P_RS+P_T=50+1000=1050 IOS
#Total Cost join2=Sort(P_RS)+Sort(P_T)+Merge+P_RST
#Total Cost join2=200+6000+50+1000+250=7500 IOS

#Join1+Join2=580+7500=8080 IOs

#Formula, with variables plugged in:
#2*P_R + 2*2*P_S + P_R + P_S + P_RS + 2*2*P_RS + 2*3*P_T + P_RS + P_T + P_RST
```

[23]:
```
# describe solution (ABOVE)

IO_Cost_SMJ_1 = 2*P_R + 2*2*P_S + P_R + P_S + P_RS + 2*2*P_RS + 2*3*P_T + P_RS␣
↪+ P_T + P_RST
IO_Cost_SMJ_1
```

[23]: 8080

1.3.5 IO_Cost_SMJ_2 [5 pts]

where only sort merge join is used, $join1 = T(c,d), S(b,c)$ and $join2 = join1(b,c,d), R(a,b)$

[24]:
```
#NOTE**********: It is assumed that B+1=33 is the max number of available␣
 ↪buffers.
#*****************************************************************************

#This logic will follow closely from the lecture for SMJ complexity.
#In lecture, it is given that the cost of SMJ is the cost of sorting + merging␣
 ↪+Output (P(R)(S) OUT) generically,
#And we know that the cost of sorting N pages with no repacking is:
#2N*(ceil(LogB(N/B+1))+1)

#A key insight is that P_R completely fits in the buffer, so it can be split␣
 ↪and sorted in just one pass.

#So, join 1, where P_T or P_S is exchanged for the N pages:
#P_T requires 3 passes to split and sort:
#Sort(P_T)=2*1000*(ceil(Log32(100/33))+1)=2*1000*3=6000 IOS
#P_S requires 2 Passes to split and sort:
#Sort(P_S)=2*100*(ceil(Log32(100/33))+1)=2*100*2=400 IOS
#Merge=P_T+P_S=1000+100=1100 IOS
#Total Cost join1=Sort(P_T)+Sort(P_S)+Merge+P_ST
#Total Cost join1=6000+400+1100+500=8000 IOs

#Join 2:
#P_ST requires 2 passes to split and sort:
#Sort(P_ST)=2*500*(ceil(Log32(500/33))+1)=2*500*2=2000 IOs
#P_R requires 1 pass to split and sort:
#Sort(P_R)=2*10*1=20 IOs
#Merge=P_ST+P_R=500+10=510 IOs
#Total Cost join2=Sort(P_ST)+Sort(P_R)+Merge+P_RST
#Total Cost join2=2000+20+510+250=2780 IOS

#Join1+Join2=8000+2780=10780 IOs

#Formula, with variables plugged in:
#2*3*P_T+ 2*2*P_S + P_S + P_T + P_ST + 2*2*P_ST + 2*P_R + P_R + P_ST + P_RST
```

[25]:
```
# describe solution (ABOVE)

IO_Cost_SMJ_2 = 2*3*P_T+ 2*2*P_S + P_S + P_T + P_ST + 2*2*P_ST + 2*P_R + P_R +␣
 ↪P_ST + P_RST
IO_Cost_SMJ_2
```

[25]: 10780

1.3.6 IO_Cost_BNLJ_1 [5 pts]

where only block nested loop join is used, $join1 = R(a,b), S(b,c)$ and $join2 = join1(a,b,c), T(c,d)$

[26]:
```
#NOTE**********: It is assumed that B+1=33 is the max number of available␣
↪buffers.
#*************************************************************************************

#The equation used here follows closely from lecture, where the complexity of␣
↪BNLJ IS:
#P(R) + ceil(P(R)/(B)) * P(S) + OUT


#Join 1:
#Join1=P_R + ceil(P_R/(B)) * P_S + P_RS
#Join1=10+ceil((10/32))*100 + 50= 160 IOS

#Join 2:
#Join 2=P_RS + ceil(P_RS/(B)) * P_T + P_RST
#Join 2=50+ceil(50/32)*1000+250=50+(2*1000)+250=2300

#Total cost=join1+join2=160+2300=2460

#Formula with variables plugged in:

#P_R + (math.ceil(P_R/(B)) * P_S) + P_RS + P_RS + (math.ceil(P_RS/(B)) * P_T) +␣
↪P_RST
```

[27]:
```
# describe solution (above)
IO_Cost_BNLJ_1 = P_R + (math.ceil(P_R/(B-1)) * P_S) + P_RS + P_RS + (math.
↪ceil(P_RS/(B-1)) * P_T) + P_RST
IO_Cost_BNLJ_1
```

[27]: 2460

1.3.7 IO_Cost_BNLJ_2 [5 pts]

where only block nested loop merge join is used, $join1 = T(c,d), S(b,c)$ and $join2 = join1(b,c,d), R(a,b)$

[28]:
```
#NOTE**********: It is assumed that B+1=33 is the max number of available␣
↪buffers.
#*************************************************************************************


```

```
#The equation used here follows closely from lecture, where the complexity of␣
 ↪BNLJ IS:
#P(R) + ceil(P(R)/(B)) * P(S) + OUT
#IMPORTANT NOTE: It is assumed that the outer loop joined is the smaller␣
 ↪relation to incur a lesser cost.
#This changes the calculation a bit, but is not stated explicitly.

#Join 1:
#Join1=P_S + ceil(P_S/(B)) * P_T + P_ST
#Join1=100+ceil((100/32))*1000 + 500= 4600 IOs

#Join 2:
#Join 2=P_ST + ceil(P_ST/(B)) * P_R + P_RST
#Join 2=500+ceil(500/32)*10+250=500+(17*10)+250=910 IOs

#Total cost=join1+join2=4600+910=5510 IOs

#Formula with variables plugged in:

#P_S + (math.ceil(P_S/(B)) * P_T) + P_ST+ P_ST + (math.ceil(P_ST/(B)) * P_R) +␣
 ↪P_RST
```

[29]:
```
# describe solution (above)

IO_Cost_BNLJ_2 = P_S + (math.ceil(P_S/(B)) * P_T) + P_ST+ P_ST + (math.
 ↪ceil(P_ST/(B)) * P_R) + P_RST
IO_Cost_BNLJ_2
```

[29]:
```
5510
```

1.3.8 Part (b) [5 pts]

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

[8 points]

[30]:
```
#So, the goal here is to find input sizes where (HJ-Join1+ SMJ-Join2) cost <␣
 ↪(SMJ-Join1+ HJ-Join2) cost.

#We know that the cost of SMJ is sorting + merging +Output (P(R)(S) OUT)
#and sorting costs 2N*(ceil(LogB(N/B+1))+1).

#The cost of HJ is 3(P(R)+P(S)) + OUT
```

```
#From the problems above, it is clear that SMJ has a much lower IO cost when␣
 ↪the smaller page value comes first.
#So, we want the result of the hash join from join1 to be as small a value as␣
 ↪possible, so that it will be the first term applied in SMJ-Join 2.
#Using the above logic, we want a larger value of P_R than P_S so that using␣
 ↪SMJ for join 1 will be as large a compound resulting value as possible.
#Let's try to keep T be a median value between P_R and P_S.

#Having a large P_R, small P_S, and possibly a median P_T value will lead to␣
 ↪the ultimate disparity of (HJ-Join1+ SMJ-Join2) cost < (SMJ-Join1 +␣
 ↪HJ-Join2) cost.

#*************************************************
#Using the formulas, above, the cost of HJ-Join1 is:
# 3*(P_R+P_S)+P_RS= 3*(1500+100)+100= 4900 IOs.
# Let this new value be C=4900.

#The cost of SMJ Join2 is:
#Sort(C) + Sort(P_T) + C + P_T + P_RST
#=2*C*(Ceil(LogB(C/B+1))+1) + 2*P_T*(Ceil(LogB(P_T/B+1))+1) + C + P_T + P_RST
#Sort C requires 3 passes:
#Sort(C)=2*4900*(Ceil(Log30(4900/31))+1)=2*4900*3=29400 IOs

#Total cost is 4900+29400=34300 IOs

#*****************************************

#The cost of SMJ Join1 is:
#Sort(P_R) + Sort(P_S) + P_R + P_S + P_RS, Let this value = C
#=2*P_R*(Ceil(LogB(P_R/B+1))+1) + 2*P_S*(Ceil(LogB(P_S/B+1))+1) + P_R + P_S +␣
 ↪P_RS
#Sort(P_R)=2*1500*(Ceil(Log30(1500/31))+1)=2*1500*3=9000
#Sort(P_S)=2*100*(Ceil(Log30(100/31))+1)=2*100*2=400
#SMJ cost=9000+400+1500+100+100=
#SMJ cost=11100 IOs Let this =C

#The cost of HJ Join2 is:
# HJ Cost=3*(C+P_T)+P_RST= 3*(11100+500)+50= 34850 IOs.

#Total cost = 11100 + 34850=45950 IOs
#*****************************************

#This ultimately achieves our goal, (HJ-Join1+ SMJ-Join2) cost < (SMJ-Join1+␣
 ↪HJ-Join2) cost
#Or 34300 IOs<45950 IOs.
```

```
#If we made P_S and P_R even bigger it could potentially make the difference␣
↪even larger, but wouldn't be as realistic, and the goal is achieved anyway.
```

[31]:
```
P_R = 1500
P_S = 100
P_T = 500
P_RS = 100
P_RST = 50
B = 30

HJ_IO_Cost_join1 = 4900
SMJ_IO_Cost_join2 = 29400

SMJ_IO_Cost_join1 = 11100
HJ_IO_Cost_join2 = 34850
```

1.4 Problem 4: Hash Join [25 pts]

We want to find out which are the colleges each NFL teams prefers drafting players from.

We have access to two tables: - a table named "teams" which contains (team, player) pairs, and - 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.

And of course HASH JOIN is the way to go!!!

1.5 Load and explore the data

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

[32]:
```
# Load data
import nfl
from nfl import *
teams, colleges = loadData()
```

For example, the first team entry is

[33]:
```
teams[0]
```

[33]:
```
PlayerTeam(teamname='Houston Texans', playername='Jadeveon Clowney')
```

And the first colleges entry is

[34]:
```
colleges[0]
```

[34]:
```
PlayerCollege(playername='Jadeveon Clowney', collegename='South Carolina')
```

Named tuples are basically lightweight object types. Instances of named tuple instances can be dereferences (e.g. `colleges[0].playername` or the standard tuple syntax (`colleges[0][0]`).

[35]:
```
print(colleges[0].playername)
print(colleges[0][0])
```

```
Jadeveon Clowney
Jadeveon Clowney
```

The following code prints the first 10 tuples from teams and colleges. *Notice how fields of named tuples are accessed inside the loops.*

[36]:
```
# 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
```

1.6 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:

[37]:
```
# 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)` defined in `nfl.py` for convenience as shown next:

[38]:
```
# Fix the number of buckets to 500
buckets = 500
# Partition the teams table using hash function h
teamsPartition = partitionTable(teams,h,buckets)
```

[39]:
```
#This prints out a big old list of teamnames, playernames that are very␣
 ↪different but hash to the same bucket...
#print(teamsPartition[97])
```

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.

1.7 Part (a)

1.7.1 Part (a.i) [5 pts]

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.

[40]:
```
def hashJoin(table1, table2, hashfunction,buckets):
    # Partition phase
    t1Partition = partitionTable(table1,hashfunction,buckets)
    t2Partition = partitionTable(table2,hashfunction,buckets)
    # Merge phase
    result = []

    # ANSWER GOES HERE

    #As in lecture, using NLJ is a good way to merge
    #This method is crucial: access keys with t_Partition.keys() method, and␣
 ↪key should match in both partitions (join key!)
    for k1 in t1Partition.keys():
        for t1Entry in t1Partition[k1]:
            for t2Entry in t2Partition[k1]:
                #Join condition
                if (t1Entry.playername == t2Entry.playername):
                    result.append((t1Entry.teamname, t1Entry.playername,␣
 ↪t2Entry.collegename))

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

    #Print a little output to show it in action
    for i in range(0,10):
        print(result[i])
    print('\n')

    return result
```

1.7.2 Part (a.ii) [5 pts]

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.

[41]:
```
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, BELOW
```

```
('Minnesota Vikings', "D'Aundre Reed", 'Arizona')
('Jacksonville Jaguars', "D'Anthony Smith", 'Louisiana Tech')
('Arizona Cardinals', "O'Brien Schofield", 'Wisconsin')
('Chicago Bears', "J'Marcus Webb", 'West Texas A&M')
('Minnesota Vikings', "D'Aundre Reed_1", 'Arizona')
('Jacksonville Jaguars', "D'Anthony Smith_1", 'Louisiana Tech')
('Arizona Cardinals', "O'Brien Schofield_1", 'Wisconsin')
('Chicago Bears', "J'Marcus Webb_1", 'West Texas A&M')
('Minnesota Vikings', "D'Aundre Reed_2", 'Arizona')
('Jacksonville Jaguars', "D'Anthony Smith_2", 'Louisiana Tech')
```

```
The join took 2132.94 ms and returned 12740 tuples in total
```

[42]:
```
#Explanation of part a.ii
#The total number of tuples output was 12740, requiring just over 2202 ms.
#This seems reasonable, considering each table had 12720 tuples in total, which␣
 ↪is a lot to iterate over.
#It may not be the best or most optimal method, but NLJ is more than fast␣
 ↪enough for our purposes.
#Note that each table is the same size, so switching the outer and inner table␣
 ↪would have no effect.
```

1.8 Part (b)

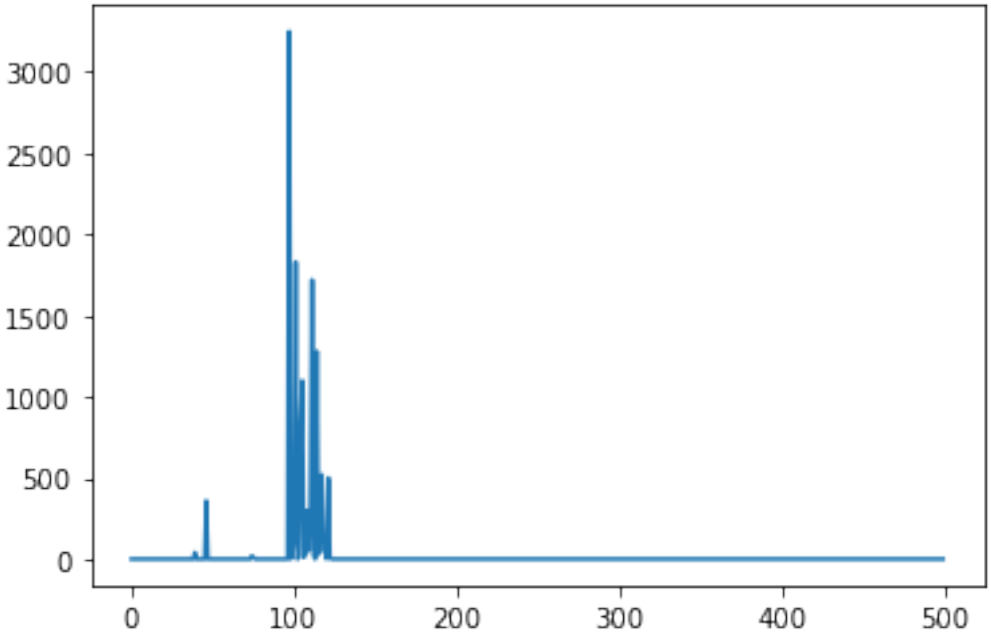
You decide to investigate the performance of `hashJoin( )` further. Since you implemented the merge phase of `hashJoin( )` yourself you focus on the partitioning obtained by using the provided hash function `h( )`. In the lectures we saw that a good hash function should partition entries uniformly across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

[43]:
```
# 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()
```

[43]: []



Part (b.i) [5 pts] 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.

[44]:
```
#Description for b.i:
# I could tell by printing out the PartitionTable for Teams that the skew is␣
 ↪great.
# Most buckets were unused, with a great amount of entries clustering within a␣
 ↪few buckets.
# The standard deviation is SQRT(Sum((x-mean)^2)/n-1) where x is an individual␣
 ↪value.
```

[45]:
```
# ANSWER
# partition- a table partition as returned by method partitionTable
```

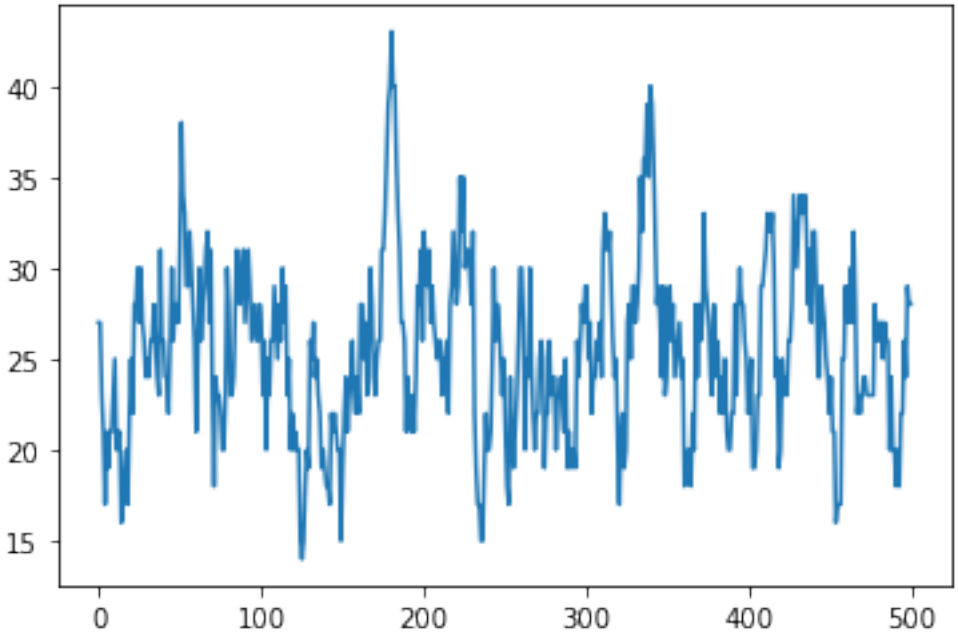
```
# return value - a float representing the skew of hash function (i.e. stdev of␣
 ↪players assigned to each team)
import statistics

def calculateSkew(partition):
    # ANSWER STARTS HERE
    #we need to store the length of each bucket to do stdev, including len=0␣
 ↪values to get correct average.
    #we access the values within each bucket by using .values() method
    temp=[len(i) for i in partition.values()]
    skew = statistics.stdev(temp)
    # ANSWER ENDS HERE
    return skew

print (calculateSkew(teamsPartition))
```

```
205.03777060519727
```

Part (b.ii) [**5 pts**] 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.

[46]:
```
# Design a better hash function and print the skew difference for it
# Here is the best time to use experimentation with different modulo values!
# The randomized values included in the hash below seem to work pretty well.

#Original Hash function:
#def h(x,buckets):
    #rawKey = ord(x[1])
    #return rawKey % buckets

def hBetter(x,buckets):
    #initialize rawkey, but then make it ultimately very variable
    rawKey = 0
    #iterate over each char in the string, x, and do some random␣
 ↪transformations to it.
    for c in x:
        rawKey=(rawKey*3)+1+(ord(c)+9)
    return rawKey % buckets

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

[46]: []



Part (b.iii) [5 pts] 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?

[47]:
```
start_time = time.time()

buckets=500

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
```

```
('Arizona Cardinals', 'Kareem Martin', 'North Carolina')
('New England Patriots', 'Steve Beauharnais', 'Rutgers')
('Cincinnati Bengals', 'Jordan Shipley', 'Texas')
('San Francisco 49ers', 'Phillip Adams_1', 'South Carolina State')
('Houston Texans', 'Nick Mondek_2', 'Purdue')
('New York Jets', 'Vladimir Ducasse_3', 'Massachusetts')
('Jacksonville Jaguars', 'Mike Harris_4', 'Florida State')
('Buffalo Bills', 'Aaron Williams_4', 'Texas')
('Minnesota Vikings', 'Christian Ballard_4', 'Iowa')
```

```
('New York Jets', 'Joe McKnight_4', 'USC')
```

```
The join took 111.07 ms and returned 12740 tuples in total
```

[49]:
```
#This is a great speed up!
# The original took 2132.94 ms, while this took 111.07 ms, making a speedup  of
#(2132.94/111.07)= 19X faster!

#The result is over 19x faster using the new hash function!
```