CS 145 – PSET 3 – Sample Answers – 2016

NOTE: there may be more possible answers than those listed here

1a.i	`io_split_sort = 360` `# 160 pages / 20 buffer pages = 8 runs` `# 8 runs * (20 / 4) = 40 IO reads` `# (Alternatively, 160 pages / 4 page per read = 40 IO reads)` `# 160 pages * 2 = 320 IO writes` `# 320 + 40 = 360 total IOs`
1a.ii	`merge_arity = 4` `# Reads are always read in 4-page chunks.` `# In a 4-way merge, we use 16 + 1 = 17 buffer pages.`
1a.iii	`merge_passes = 2` `# ceiling(log_4(8)) = 2` `# first pass: 8 runs => 2 runs` `# second pass: 2 runs => 1 run`
1a.iv	`merge_pass_1 = 360` `# 160 pages / 4 page per IO = 40 read IOs` `# 160 * 2 = 320 IO writes` `# 320 + 40 = 360 total IOs`
1a.v	`total_io = 1080` `# split_and_sort = 360 IOs` `# merge_pass_1 = 360 IOs` `# merge_pass_2 = 360 IOs`
1b.i	`def cost_initial_runs(B, N, P):` `    # We make each run equal to the size of the buffer` `    # since this is the largest amount we can sort in memory` `    # The IO cost to read in the run is (B+1)/P.` `    # The IO cost to write the run is (B+1)*2` `    # There are N/(B+1) of these runs - no floor / ceiling` `    # is needed due to the assumption that N%(B+1) == 0` `    return (N*2)+(N/P)`
1b.ii	`def cost_per_pass(B, N, P):` `    # On each merge pass we will read in exactly N/P chunks = N/P IO` `    # The total write per merge pass will N*2` `    return (N*2)+(N/P)`

1b.iii	<pre>def num_passes(B, N, P): # at each step, how many blocks are we joining # B / P gets us the number of blocks we can merge(since we need 1 to output), # we need to floor it because B might not be divisible by P # final: floor(B/P) # we have num_of_passes = log_base(floor(B/P))(N/(B + 1))) # need to ceiling it since might not be perfect merge # final: ceiling(log_base(floor(B/P))(N/B+1)) # whew! return math.ceil(math.log(N/(B + 1), math.floor(B/float(P))))</pre>

1c	``` B = 99 N = 900 feasible_p_range = range(1, B/2) #if divisibility assumptions were carried over from part b: # feasible_p_range = [] # for i in range(1, B/2): # if 100 % i == 0: # feasible_p_range.append(i) p1_points = [(p, external_merge_sort_cost(B, N, p)) for p in feasible_p_range] # Save the optimal value here P = 11 # P can also be P = 10 depending on if divisibility assumptions from #(b) were carried over. We will accept those. # Save a list of tuples of (P, io_cost) here, for all feasible P's points = p1_points ```

2a

```
# Since (B - 1)(B - 2) = 930 >> min(P_R, P_S):
# P_R fits completely in memory, no partition phase needed
# IO(join1) = 10 + 100 + 50 OUT = 160
# Similarly:
# IO(join2) = 3(50 + 1000) + 250 OUT = 3400
# Total: 3400 + 160 = 3560
```

IO_Cost_HJ_1 = 3560

```
# (B - 1)(B - 2) = 930 >> min(P_S, P_T)
# IO(join1) = 3(100 + 1000) + 500 OUT = 3800
# IO(join2) = 500 + 10 + 250 OUT = 760
# Total: 4560
```

IO_Cost_HJ_2 = 4560

```
# join1:
# 1 pass(R/W) to sort R (2 * 10 = 20)
# 2 pass(R/W) to sort S (4 * 100 = 400)
# 1 pass(R) to merge (10 + 100 = 110)
# Total = 20 + 400 + 110 + 50 = 580

# join2:
# 2 pass(R/W) to sort RS(4 * 50 = 200)
# 3 pass(R/W) (B * (B - 1) = 992 < 1000) to sort T (6 * 1000 =
6000)
# 1 pass(R) to merge (50 + 1000 = 1050)
# OUT = 250
# Total = 200 + 6000 + 1050 + 250 = 7500

# Total = 8080
```

IO_Cost_SMJ_1 = 8080

2a

```
# join1:
# 3 pass(R/W) (B * (B - 1) = 992 < 1000) to sort T (6 * 1000 =
6000)
# 2 pass(R/W) to sort S (4 * 100 = 400)
# 1 pass(R) to merge (100 + 1000 = 1100)
# OUT = 500
# Total = 6000 + 400 + 1100 + 500 = 7500

# join2:
# 2 pass(R/W) to sort ST (4 * 500) = 2000
# 1 pass(R/W) to sort R (2 * 10 = 20)
# 1 pass(R) to merge (10 + 500 = 510)
# OUT = 250
# Total = 2000 + 20 + 510 + 250 = 2780

# Total: 10280

IO_Cost_SMJ_2 = 10280

# From lecture: P(R) + P(R)P(S)/B + OUT
# Where one should use smaller of two relations as R

#join1: 10 + ceiling(10/30) * 100 + 50 = 160

#join2: 50 + ceiling(50/30) * 1000 + 250 = 2300

#Total: 160 + 2300 = 2460

IO_Cost_BNLJ_1 = 2460

#join1: 100 + ceiling(100/30) * 1000 + 500 = 4600
#join2: 10 + ceiling(10/30) * 500 + 250 = 760
#Total: 4600 + 760 = 5360

IO_Cost_BNLJ_2 = 5360
```

2b	see code below

```
P_R = 10
P_S = 50
P_T = 20
P_RS = 20
P_RST = 25
B = 30

# HJ 1
# B^2 > min(P_R, P_S)
#join 1 = 3*(10 + 50) + 20 = 200
HJ_IO_Cost_join1 = 200
# SMJ 2
# sortRS = 2 * 20 = 40
# sortT = 2*20
# total = 40 + 40 + 20 + 20 + 25 = 145
SMJ_IO_Cost_join2 = 145
# totalIO = 200 + 145 = 345

#SMJ 1
# sortR = 2* 10 = 20
# sortS = 2* 50 * 2 = 200
# total = 300
SMJ_IO_Cost_join1 = 20 + 200 + 10 + 50 + 20

# HJ 2 B^2 > min( 20, 20)
HJ_IO_Cost_join2 = 3*(P_RS + P_T) + P_RST

print HJ_IO_Cost_join1 + SMJ_IO_Cost_join2
print HJ_IO_Cost_join2 + SMJ_IO_Cost_join1

345
445
```

```
# Possible Idea:
# Have P_R be small while P_S be large. This will result in HJ
for join1 being much cheaper using HJ than SMJ for join1

P_R = 10
P_S = 10000
P_T = 100
P_RS = 50
P_RST = 25
B = 20

HJ_IO_Cost_join1 = 10060
SMJ_IO_Cost_join2 = 775

SMJ_IO_Cost_join1 = 90080
HJ_IO_Cost_join2 = 475

#For reference: function calculating HJ, SMJ for sanity-check
def HJ_cost_calc(input1, input2, buf, out):
    #From lecture notes, note B is B + 1 in notes
    B = buf - 1
    smaller = min(input1, input2)
    return 2 * math.ceil(math.log(math.ceil(float(smaller)/(B -
1)), B)) * (input1 + input2) + (input1 + input2) + out

def SMJ_cost_calc(input1, input2, buf, out):
    #From lecture notes, note buf is B + 1 in notes
    B = buf - 1
    return 2 * input1 * (1 +
math.ceil(math.log(math.ceil(float(input1)/(B + 1)), B))) + \
           2 * input2 * (1 +
math.ceil(math.log(math.ceil(float(input2)/(B + 1)), B))) + \
           input1 + input2 + out

plan1 = HJ_cost_calc(P_R, P_S, B, P_RS) \
    + SMJ_cost_calc(P_RS, P_T, B, P_RST)
plan2 = SMJ_cost_calc(P_R, P_S, B, P_RS) \
    + HJ_cost_calc(P_RS, P_T, B, P_RST)

print HJ_cost_calc(P_R, P_S, B, P_RS), SMJ_cost_calc(P_RS, P_T,
B, P_RST)
print SMJ_cost_calc(P_R, P_S, B, P_RS), HJ_cost_calc(P_RS, P_T,
B, P_RST)
print plan1, plan2
```

```
P_R = 10
sort_R = 2
P_S = 100
sort_S = 4
P_T = 100
sort_T = 4
P_RS = 50
sort_RS = 4
P_RST = 25
B = 30

HJ_IO_Cost_join1 = 3*(P_R+P_S)+P_RS
SMJ_IO_Cost_join2 = sort_RS*P_RS+sort_T*P_T+P_RS+P_T+P_RST
print "TOTAL: " + str(HJ_IO_Cost_join1+SMJ_IO_Cost_join2)

SMJ_IO_Cost_join1 = sort_R*P_R+sort_S*P_S+P_R+P_ST+P_RS
HJ_IO_Cost_join2 = 3*(P_RS+P_T)+P_RST
print "TOTAL: " + str(SMJ_IO_Cost_join1+HJ_IO_Cost_join2)

TOTAL: 1155
TOTAL: 1455
```

3a.i	<pre>def lru_cost(N, M, B): # For N <= B+1, you can read the data in once, # and then loop over it: if N <= B+1: return N # Otherwise, you end up needing to read in each # page each iteration! else: return N*M</pre>

3a.ii

```
def mru_cost(N, M, B):
    if (N <= B + 1):
        return N

    #initial reads
    buf = range(B+1)
    io = B+1
    pos = B
    mru = B
    passes = 0

    while True:
        pos+=1
        if (pos >= N):
            pos = 0
            passes+=1
        if (passes >= M):
            break
        if pos in buf:
            mru = buf.index(pos)
        else:
            buf[mru] = pos
            io+=1

    return io
```

3a.iii	`B = 6` `N = 10` `M = 20` `p3_lru_points = [(m, abs(lru_cost(N, m, B) - mru_cost(N, m, B))) for m in range(1, M+1)]`
	`B = 6` `N = 10` `M = 20` `p3_lru_points = [(m, abs(lru_cost(N, m, B) - mru_cost(N, m, B))) for m in range(1, M+1)]`
	`B = 6` `N = 10` `M = 20` `p3_lru_points = [(m, abs(lru_cost(N, m, B) - mru_cost(N, m, B))) for m in range(1, M+1)]`
3b.i	(code below)

```
def clock_cost(N, M, B):
    # YOUR CODE HERE
    clock = [0 for i in range(B+1)]
    b = [None for i in range(B+1)]
    arm = 0
    reads = 0
    for m in range(M):
        for n in range(N):
            if n not in b:
                # Buffer not full
                if None in b:
                    index = b.index(None)
                    b[index] = n
                    clock[index] = 1
                # Evict
                else:
                    while clock[arm] == 1:
                        clock[arm] = 0
                        arm = (arm + 1) % len(b)
                    b[arm] = n
                reads += 1
                clock[arm] = 0
                arm = (arm + 1) % len(b)
            else:
                clock[index] = 1

    return reads
```

```python
def clock_cost(N, M, B):
    b = [None]*(B+1)
    secondChance = [0]*(B+1)
    clock = 0
    reads = 0
    for i in range(M):
        for x in range(N):
            if x not in b:
                if b[clock] == None:
                    b[clock] = x
                else:
                    while secondChance[clock] == 1:
                        secondChance[clock] = 0
                        clock = (clock + 1) % (B+1)

                    b[clock] = x
                secondChance[clock] = 0
                clock = (clock + 1) % (B+1)
                reads += 1
            else:
                secondChance[b.index(x)] = 1
    return reads
```

```python
def clock_cost(N, M, B, verbose=False):
    """
    Calculate CLOCK cost by just implementing the algorithm's
steps
    NOTE that this is distinct from how the actual algorithm is
implemented!
    Verbose mode included, works for single-digit numbers at
least
    """
    if verbose: print "**CLOCK will have a bar over it**\n"
    b = [None]*(B+1)
    secondChance = [0]*(B+1)
    clock = 0
    reads = 0
    prev_reads = 0
    for i in range(M):
        if verbose: print "Iteration %s:" % i
        for x in range(N):
            if x not in b:
                if b[clock] == None:
                    b[clock] = x
                else:
                    while secondChance[clock] == 1:
                        secondChance[clock] = 0
                        clock = (clock + 1) % (B+1)

                    b[clock] = x
                secondChance[clock] = 0
                clock = (clock + 1) % (B+1)
                reads += 1

            # If x is already in buffer, just read from buffer
and mark
            # the second chance flag to one
            else:
                secondChance[b.index(x)] = 1
            if verbose:
                s = " ".join([" " if i != clock else "_" for i
in range(B+1)]) + "\n" + " ".join(map(str,b))
                s += "  [R]" if (reads - prev_reads) > 0 else
""
                prev_reads = reads
                print s
    return reads
```

3b.ii

```python
B = 6
N = 10
M = 20
p3_clock_points = [(m, abs(lru_cost(N, m, B) - clock_cost(N, m,
B))) for m in range(1, M+1)]
# Clock algorithm has the same behavior as LRU
```

	`` B = 6 N = 10 M = 20 p3_clock_points = [(m, abs(lru_cost(N, m, B) - clock_cost(N, m, B))) for m in range(1, M+1)] # CLOCK eviction is a form of LRU, which does not prevent sequential flooding ```
	`` #SOLUTION #Exact same behavior as LRU does not prevent sequential flooding.#SOLUTION ```
4a.i	(see code below)

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


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

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

    # ANSWER GOES HERE

    # 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))
    for b in range(buckets):
        for t1Entry in t1Partition[b]:
            for t2Entry in t2Partition[b]:
                if t1Entry.playername == t2Entry.playername:
                    result.append((t1Entry.teamname,
t1Entry.playername, t2Entry.collegename))
    return result
```

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

    # ANSWER GOES HERE

    # To populate your output you should use the following code
    # result.append((t1Entry.teamname, t1Entry.playername,
t2Entry.collegename))
    for b in range(buckets):
        for t1Entry in t1Partition[b]:
            for t2Entry in t2Partition[b]:
                if t1Entry.playername == t2Entry.playername:
                    result.append((t1Entry.teamname,
t1Entry.playername, t2Entry.collegename))
    return result
```

<table>
<tr><td rowspan="3">4a.ii</td><td><pre>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))

The join took 8862.79 ms and returned 12740 tuples in total
The runtime does not seem ideal. It should be faster but my
gut says
that the hash function is not ideal</pre></td></tr>
<tr><td><pre>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))

No, the time of the join seems a bit longer than expected.
part b and c explains why(skewed buckets)!
#The join took 8879.44 ms and returned 12740 tuples in total</pre></td></tr>
<tr><td><pre>The join took 6593.04 ms and returned 12740 tuples in total</pre></td></tr>
<tr><td rowspan="3">4b.i</td><td><pre>skew = np.std([len(partition[i]) for i in
range(len(partition))])
skew = 204.832630213</pre></td></tr>
<tr><td><pre>skew = np.std([len(partition[p]) for p in partition])
skew = 204.832630213</pre></td></tr>
<tr><td><pre>Skew = 204.832630213</pre></td></tr>
<tr><td>4b.ii</td><td><pre>rawKey = hash(x)</pre></td></tr>
</table>

4b.iii	The join took 171.86 ms and returned 12740 tuples in total
	The join took 170.52 ms and returned 12740 tuples in total
	# The join took 172.67 ms and returned 12740 tuples in total