# HW1 Solutions: CS425 FA23

*Recommended solutions for HW1 (FA23) are below. For many questions, alternate solutions are possible and reasonable correct solutions will be accepted during grading. So please refrain from asking questions about solutions until you receive your HW grades back.*

1. (Solution and Grading by Shubhi Jain.)
   (Variants of these answers are also accepted, as long as they are correct)
   a. While AWS EC2 allows you to choose VM instances as “servers” and you pay by VM-hour (fixed price per instance type), AWS Lambda is “Function as a Service” (FAAS) and it allows one to invoke functions (like in a program) in a “serverless” manner. The pricing for AWS Lambda is by usage only, and so it is typically cheaper than AWS EC2 for hosting web services that are invoked only when requests are received.
   b. While AWS Lambda is serverless FAAS, AWS Spot Instances are closer to AWS EC2 (i.e., full VM instances), with the exceptions that (1) the pricing is not fixed in spot instances but depends on market prices, and (2) if the “bid” price falls below the market price, the instance can be killed unilaterally by AWS (often AWS will send a warning before killing it). AWS EC2 instances typically cannot be killed unilaterally by AWS, and its prices are fixed (per instance type).
   c. Microsoft Azure Functions, and (Google) Cloud Functions.
   d. AWS EC2 is useful for compute applications, e.g., a Mapreduce job for an important application. AWS Lambda is useful for web services that service requests (which in turn invoke functions to service the request), which gives the service automatic scalability.

2. (Solution and Grading by: Chaitanya Bhandari.)

We will accept solutions that consider either main-memory or GPU memory. Additionally, since the question was open to interpretation, we will also accept solutions that use custom VM types. However, please note that the solutions below consider only preset machine types.

a. Maximum Memory
   i. ***Considering main memory only: Across all cloud platforms, the preset VM instance with one GPU and the maximum main memory is Azure's*** *Standard_NV36adms_A10_v5* ***(880GiB)***
      1. GCP: *n1-highmem-96* *(581.145GiB)*
         a. Most VMs in the N1 series (except shared core) can be attached with GPUs [Reference]
      2. Azure: *Standard_NV36adms_A10_v5* *(880GiB)*
      3. AWS: *g5.16xlarge* */(g4dn.16xlarge) (256GiB)*
   ***ii.*** ***Considering GPU memory only, not main memory: Across all cloud platforms, the preset VM instance with one GPU and the maximum GPU memory is GCP's*** a2-ultragpu-1g (74.5GiB)
      1. AWS: G5-series with 1 GPU (24GiB)
      2. Azure: Standard_NG32ads_V620_v1/Standard_NG32adms_V620_v1 (32GiB)
      3. GCP: a2-ultragpu-1g (74.5GiB)

b. Minimum memory
   i. ***Considering main memory only: Across all cloud platforms, the VM instance with one GPU and the minimum main memory is GCP's*** *n1-highcpu-2* ***(1.67GiB).*** These are the results for each cloud platform:
      1. GCP: *n1-highcpu-2* *(1.67 GiB)*
         a. Most VMs in the N1 series (except shared core) can be attached with GPUs [Reference]
      2. Azure: *Standard_NC4as_T4_v3* *(28GiB)*
      3. AWS: *g5g.xlarge* *(8GiB)*
   *ii.* ***Considering GPU memory only, not main memory: Across all cloud platforms, the preset VM instance with one GPU and the minimum GPU memory is Azure's*** Standard_NV12s_v3 **(8GiB)**
      1. GCP: g2-standard-4 (14.9GiB)
      2. Azure: Standard_NV12s_v3 (8GiB)
      3. AWS: P2 Instances (12GiB)

c. GPUs were originally meant for graphics processing, but have also been useful for ML operations since they allow thousands of parallel multiplications and additions. However, they incur expensive memory accesses while performing these operations. On the other hand, Tensor Processing Units (TPUs) are

Google's specialized matrix processor for ML operations. They have on-chip high-bandwidth memory, which allows TPUs to avoid expensive memory accesses. GCP provides Cloud TPUs, but Azure & AWS do not provide specialized hardware to accelerate ML workloads.

3. (Solution and Grading by: Lavanya Puri, Shulin Pan)

```
MapReduce 1:
M1 Input (a,b): // For each input line (a,b) , i.e., a follows b
        output (key=a, value=(OUT , b))
        output (key=b, value=(IN, a))

R1 Input (key=user, value=list of (IN/OUT, x) tuples):
        followerList =  list(value[1] in values where value[0] == IN)
        followingList = list(value[1] in values where value[0] == OUT)

    If |followerList| >= 100 million and |followingList| <10:
          output (key=user, value="#")
     // if this user has at least 100M followers and follows less than 10 people
     // then the user satisfies first 2 conditions, output user as key with value "#"

     If |followerList| >= 10 million:
          For each u in followerList:
             output (key=u, value="*")
     // each account u satisfies the third condition since the key user account it follows
     // has at least 10 million followers
```

```
MapReduce 2:
M2: Input (key=u, value=v) // u, v is the output of R1
          output(key=u, value=v)

R2: Input (key=u, value=list of v)
        If # and * are in value:  // indicate that all three conditions are satisfied by u
           output(u, -)
```

4. (Solution and Grading by: Christina Youn, Shubhi Jain.)

MapReduce1 determines the users that fulfill condition (i) and looks at the list of people these big users follow:

```
map1(a, b):
    output (key=a, value=(OUT, b))
    output (key=b, value=(IN, a))
reduce1(key=user, list(values)):
    out_list = list(value in values where value[0] == OUT)
    in_list = list(value in values where value[0] == IN)
    if len(in_list) >= 100 million:
        for each f in out_list:
            output (key=f.value[1], value=user)
```

MapReduce2 creates pairs of big users who follow the same user:

```
map2(key=user, value=big_user):
    output (key=user, value=big_user)
reduce2(key=user, list(big_users))
    for big_user1 in big_users:
        for big_user2 in big_users:
            output lexographic_sort(big_user1, big_user2)
```

MapReduce3 determines which big users have a following overlap of at least 100 accounts to fulfill condition (ii):

```
map3(key=U, value=V):
    if U != V:
        output (key=(U, V), value=1)
reduce3(key=(U, V), list(values)):
    if len(values) >= 200:                    // pairs were double counted in reduce2
        output (U, V)
```

5. (Solution and Grading by: Lilia Tang, Kshitij Phulare.)

```
MR1 reads from D1 to process directed edges to find who follow each other:
    map1(key=a, value=b):
        output (key=lexicographic_sort(a, b), value=1)
    reduce1(key=(a, b), value=V):
        if |V| == 2:
          output (a, b)
          output (b, a)
```

```
MR2 reads from D2 and combines all times for one user:
    map2(key=a, value=(start_time, end_time)):
        output (key=a, value=(start_time, end_time))
    reduce2(key=a, times):
        output (a, times)
```

```
MR3 reads from MR1's and MR2's outputs and connects the users times with the
filtered followers
    map3(key=a, value=V): // each value is either b or times_a
      output (key=a, value=V)
    reduce3(key=a, value=list of M3 values)
      split M3 into followers_list and times_a
      for all b in followers_list:
         output (key=lexicographic_sort(a, b), value=times_a)
```

```
MR4 reads from MR3's output and outputs the ones that overlap:
    map4(key=(a, b), value=times):
      output (key=(a, b), value=times)
    reduce4(key=(a, b), value=times_list): // value now contains times for a and b
          if times in times_list overlap:
           output (a, b)
```

6. (Solution and Grading by: Maleeha Masood, Aman Khinvasara, Shulin Pan.)

**Solution 1 (of 2)**:

Votes = {“ABC”, “BAC”, “CBA”, …}

```
MR1 determines count of pairwise dominance
   map1(key=empty, value=Votes)
      For vote in Votes:
         if vote.index(‘A’) < vote.index(‘B’):
            output(‘A_dom_B’, 1)
         elif vote.index(‘A’) > vote.index(‘B’):
            output(‘A_dom_B’, -1)
         if vote.index(‘A’) < vote.index(‘C’):
            output(‘A_dom_C’, 1)
         if vote.index(‘A’) > vote.index(‘B’):
            output(‘A_dom_C’, -1)
         if vote.index(‘B’) < vote.index(‘C’):
            output(‘B_dom_C’, 1)
         if vote.index(‘A’) < vote.index(‘B’):
            output(‘B_dom_C’, -1)
   reduce1(key=string, value=int)
      output(str, sum of ints)

MR2 uses output of MR1 to determine overall result
   map2(key=str, value=sum of ints)
      if input(‘A_dom_B’) > 0:
         output(‘A’, 1)
      elif input(‘A_dom_B’) < 0:
         output(‘B’, 1)
      if input(‘A_dom_C’) > 0:
         output(‘A’, 1)
      elif input(‘A_dom_C’) < 0:
         output(‘C’, 1)
      if input(‘B_dom_C’) > 0:
         output(‘B’, 1)
      elif input(‘B_dom_C’) < 0:
         output(‘C’, 1)
   reduce2(key=char, value=0/1)
      if sum(input[key]) == 2:
         output(key)
```

**(Alternative) Solution 2 (of 2)**:

```
Votes = {“ABC”, “BAC”, “CBA”, …}
m = number of candidates

// MR1 determines count of pairwise dominance
   map1(key=empty, value=Votes)
      for i in 1 to m-1:
         for j in i+1 to m-1:
            if vote[i] lexicographically < vote[j]:
               output((vote[i], vote[j]), 1)
            else:
               output((vote[j], vote[i]), -1)
   reduce1(key=(A,B), value=1/-1)
      if sum(values) > 0:
         output(A, B)
      else:
         output(B, A)

// MR2 uses output of MR1 to determine overall result
   map2(key=A, value=B)
     output(1, (A, B))
   reduce2(key=1, value=(A,B))
      create int Carray[m] of candidates
      for each (A, B) pair in value:
         Carray[A]++
      if Carray[A] == m-1:
         output(A, “Condorcet Winner!”)
      else:
         Find max count in Carray
         Find set S of all candidates whose Carray[X ] == max
         output(set S, “No Condorcet Winner, Highest Condorcet Counts”)
```

7. (Solution and Grading by: Taksh Soni.)

Any gossip with a fixed fanout will require at least O(log(N)) rounds to get to O(N) (say 50% of N) members, as discussed in class. This is because the “best” dissemination involves a spanning tree with processes receiving duplicate gossips (this is true of push gossip of course, and it is also true of pull gossip on average; for the latter imagine a world where each node limits the number of pulls from it per gossip round). So asymptotically pull cannot be faster than push in getting to O(N) infected. However, as discussed in class, once O(N) nodes already have the gossip, the “tail” of the gossip (getting from O(N) to “high probability” nearly N infected) is much faster in pull gossip (O(loglogN)) than in push gossip (O(logN)).

(This problem was not that hard! Many students overthought the problem…)

8. (Solution and Grading by Shulin Pan.)
   (i) No, it is not O(log(N)), but higher. The gossip takes O(log($\sqrt{N}$)) time to spread inside each subnet (once at least one process is infected), and O(log($\sqrt{N}$)) = O(log(N)) (since log($\sqrt{N}$) = ½ log(N)). Consider any subnet Si that just got infected - it takes O(log(N)) rounds for the gossip to spread inside Si (this is true with inside-subnet target selection, and fun fact -- it would be true even if outside-subnet target selection occurred). Due to the "restriction" it takes O(log(N)) rounds (after Si is first infected) for Si's nodes to start selecting outside-subnet targets from $S((i+1) \bmod \sqrt{N})$. But once the outside-subnet targets start being selected, on average it takes O(1) rounds for at least one gossip to go across from Si to $S((i+1) \bmod \sqrt{N})$ -- the rationale is the same as the topology-aware gossip discussed in lecture. So the $\sqrt{N}$ subnets are "daisy-chained" and the number of rounds it takes for the gossip to spread to all subnets is = (O(log(N)) + O(1) + O(log(N)) + O(1) + … O(log(N)) rounds (wherein O(log(N)) is repeated $\sqrt{N}$ times). This is $O(\sqrt{N} * \log(N))$ rounds to get the gossip across all subnets, still much smaller than O(N), but not as small as O(log(N)).

   (ii) No, the router load is not O(1), but can be O( $\sqrt{N}$) towards the end of the gossip. Each subnet sends on average 1 gossip "out" of it, even when all the subnet's processes are infected. However, when all N nodes are gossiping (towards the end of the gossip spread), since there $\sqrt{N}$ subnets, there will be on average $\sqrt{N}$ gossips going through the router. (The load on the router is O(1) at the start of the gossip, but the question asks "at any time during the gossip spread").

9. (Solution and Grading by Kshitij Phulare.)
    a. One advantage is that it saves space in the short term, and another advantage is that it leads to a shorter time to mark a failed process as failed, because a failed entry is removed immediately. One disadvantage is that the failed servers may remain as ghost entries in the membership list, because every time a member receives gossip containing a server already failed, it treats it as a new entry and will gossip it to other nodes.
    b. One advantage of not using the suspicion feature is that the failure detection time is smaller, because a member is considered to have failed immediately after detection rather than wait for suspicion timeout. One disadvantage is that it will have a higher false positive rate, because the suspicion feature reduces false positives by suspect first and then consider the member as failed.
    c. One advantage is that it removes the overhead of randomizing the list. One disadvantage is that the completeness is not time-bounded; in the worst case a failed node will never be selected as the ping target.

10. (Solution and Grading by: Aman.)
    a. Before beginning, observe that Pi and pj being in the same row/column/aisle is a symmetric relation. For any given Pi, the set of nodes who have a timer for it and can mark it as failed are the same. A monitoring set is of size M+K+R-3. The first completeness violation will be when everyone in my monitoring set fails, and then I fail, so M+K+R-2 failures can violate completeness.
    b. Symmetry observation means also M+K+R-2.
    c. From above, we can always tolerate M+K+R-3 failures. So we want to solve for M such that M+K+R-3 >= 9. Plugging in K=R=2 gives M>=8.
    d. Neither ever satisfies accuracy, because heartbeating and ping-based failure detectors which satisfy completeness can never satisfy accuracy 100% in an asynchronous system.

**========== END of HOMEWORK 1 SOLUTION ==========**