Threads: A sequence of instructions given to the CPU by a program or application
- It shares heap, static data, code (address space)
- Useful for lower memory overhead
- Threads belonging to the same process can communicate via shared state.

Sockets: Allows processes on different machines or the same machine to communicate with each other over a network
- Sockets can be used by processes to communicate on the same machine

PageRank: (1-d)/N + d*(PR(others)/L + …)   * L : # of arrows pointing out
- It is not dependent on the query
- Determined by incoming links

HITS algorithm:
- Authority scores of root set ultimately used for ranking (Not the hub scores)
- Authority scores consider only incoming links
- H/A Calculate A first and then H (Set 1/1 all nodes, then calculate A: count incoming links, H: add authority scores of the node that this node is pointing to)

Boolean retrieval: queries are formulated as Boolean expressions
- Boolean retrieval is generally more efficient than using tf-idf when searching for documents that contain exactly the same terms as the query.

TF-IDF: Inverse Document Frequency (Total / include term => lower is better)
- requires more work compared to Boolean retrieval
- uses angles between vectors (cosine similarity)

TCP sliding window: Fast sender overloads a slow receiver / **congestion window**: Many senders overloads a network router

CAP theorem: Consistency, Availability, Partition-tolerance
- NoSQL databases: Availability over consistency

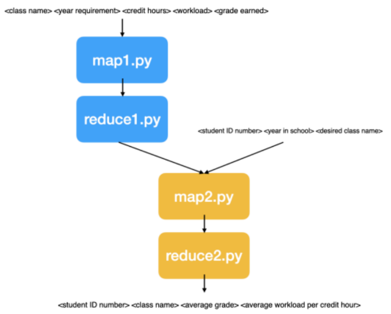
Precision and Recall formula:
Precision = TP/(TP+FP) , Recall = TP/(TP+FN)

CDN: Content Delivery Network, geographically distributed group of servers that caches content close to end users
- quick transfer of img, vid
- Traditional CDNs can support client-side dynamic pages because javascript files are static
- store data closer to clients, leading to reduced bandwidth costs for reading static files

DNS with load balancer: load balancer is the middleman to forwards requests to backend servers (same domain name but can be different IP address)

```
class Manager:
    """Manager class handles all register/join messages, handles chat room
    logic
    """
    def __init__(self, host, port):
        """Initialize Manager instance."""
        self.host = host
        self.users = {} # {<username>: (<host>, <port>)}
        self.rooms = {} # {<room_name>: {
                        #       'port': <port num>,
                        #        members: <set or list of usernames>}}
        self.free_port = 6001

        # listen for messages on manager host/port in main thread
        tcp_server(host, port, self.handle_message)

    def handle_room_message(self, msg):
        """Handle user messages sent to a chat room.

        If msg type is 'leave_room' remove specified user from the room.
        If msg type is 'send_message' send message to all users in the room
        except the message sender.
        """
        sender = msg['username']
        room_name = msg['room_name']

        if msg['type'] == 'leave_room':
            self.rooms[room_name]['members'].remove(sender)
            return

        for to_user in self.rooms[room_name]['members']:
            if to_user != sender:
                host, port = self.users[to_user]
                tcp_client(host, port, msg)
    def handle_message(self, msg):
        """Handle user messages to the server.

        If message_type is "register" create and keep track of user and the
        host and port that user is listening on. Send register_ack message.

        If message_type is "join", create room if room_name doesn't exist
        and start tcp_server thread for that room. Then add a user as a
        member of the room. Send join_ack message.
        """
        if msg['type'] == 'register':
            username = msg['username']
            user_host = msg['host']
            user_port = msg['port']

            # register new user
            self.users[username] = (user_host, user_port)

            # send register ack
            tcp_client(user_host, user_port, {
                'type': 'register_ack'
            })

        elif msg['type'] == 'join_room':
            username = msg['username']
            room_name = msg['room_name']
            user_host = self.users[username][0]
            user_port = self.users[username][1]

            # if room doesn't exist, create it. Increment available port
            if room_name not in self.rooms:
                room_thread = threading.Thread(
                target=tcp_server,
                   args=[
                       self.host, self.free_port, self.handle_room_message
              ])

            # start thread and forget about it because no shutdown logic
            room_thread.start()

            # initialize room port and member list for room
            self.rooms[room_name] = {
                'port': self.free_port,
                 'members': set()
            }

            # increment free port
            self.free_port += 1

        # add user to room, send join_ack
        self.rooms[room_name]['members'].add(username)
        tcp_client(user_host, user_port, {
            'type': 'join_ack',
            'room_host': self.host,
            'room_port': self.rooms[room_name]['port']
        })
```

```
                _, id, student_year = line
                students[id] = int(student_year)
            else:
                _, class_year, grade, work_credit_ratio = line
                found_class = True

        class_year = int(class_year)

        if found_class:
            for id, student_year in students.items():
                if student_year >= class_year:
                    print(f"{id} {key} {grade} {work_credit_ratio}")
```


<class name> <year requirement> <credit hours> <workload> <grade earned>
map1.py
reduce1.py
<student ID number> <year in school> <desired class name>
map2.py
reduce2.py
<student ID number> <class name> <average grade> <average workload per credit hour>

map1.py

```
#!/usr/bin/env python3
# map1.py

# Assume all packages are imported

# input:
# <class_name> <year_requirement> <credit_hours> <workload> <grade>
# output:
# <class_name>\t<year_requirement> <credit_hours> <workload> <grade>

for line in sys.stdin:
    line = line.rstrip().split()
    print(f"{line[0]}\t{' '.join(line[1:])}")
```

reduce1.py

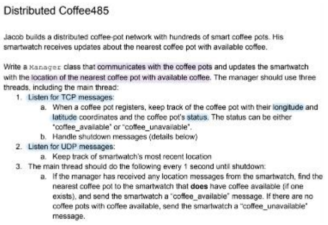
```
# Assume all packages are imported

# input:
# <class_name>\t<year_requirement> <credit_hours> <workload> <grade>
# output:
# <class> <year_requirement> <average_grade> <workload_total/credit_total>


def reduce_1_group(key, group):
    group = list(group)
    grade_total = 0
    workload_total = 0
    credit_total = 0
    year_requirement = 0

    for line in group:
        _, year, credits, workload, grade = line.rstrip().split()
        year_requirement = year
        credit_total += int(credits)
        workload_total += float(workload)
        grade_total += float(grade)

    print(f"{key} {year_requirement} {grade_total/len(group)} "
          f"{workload_total/credit_total}")
```

map2.py

```
# input:
# <student_id> <year> <class_name>
# <class_name> <year_requirement> <average_grade> <workload/credit>
# output:
# <class_name>\t<student_id> <year>
# <class_name>\t<year> <average_grade> <workload/credit>


for line in sys.stdin:
    line = line.rstrip().split()
    if len(line) == 3:
        student_id, year, class_name = line
        print(f"{class_name}\t{student_id} {year}")
    else:
        class_name, year, grade, work_credit_ratio = line
        print(f"{class_name}\t{year} {grade} {work_credit_ratio}")
```

reduce2.py

```
#!/usr/bin/env python3
# reduce2.py

# Assume all packages are imported


# input:
# <class_name>\t<student_id> <year>
# <class_name>\t<year> <average_grade> <workload/credit>
# output:
# <student_id> <class_name> <average_grade> <workload/credit>


def reduce_2_group(key, group):
    class_year = 0
    grade = 0
    work_credit_ratio = 0
    students = {}
    found_class = False

    for line in group:
        line = line.rstrip().split()
        if len(line) == 3:
```

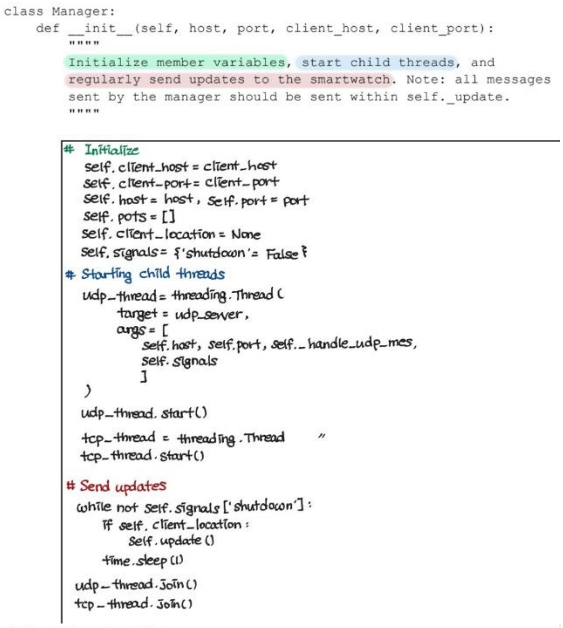


Distributed Coffee485

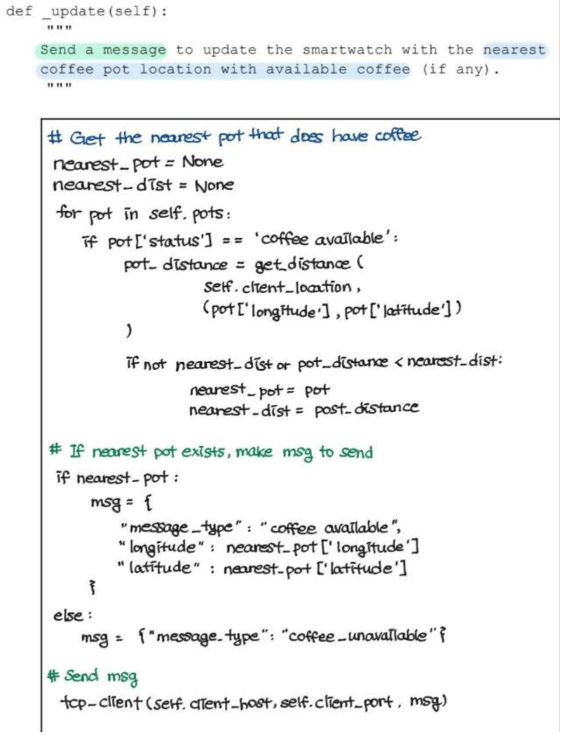
Jacob builds a distributed coffee-pot network with hundreds of smart coffee pots. His smartwatch receives updates about the nearest coffee pot with available coffee.

Write a `Manager` class that communicates with the coffee pots and updates the smartwatch with the location of the nearest coffee pot with available coffee. The manager should use three threads, including the main thread:

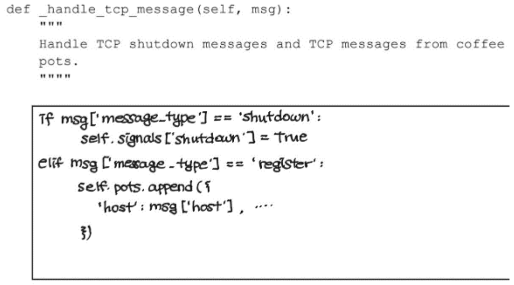
1. Listen for TCP messages:
    a. When a coffee pot registers, keep track of the coffee pot with their longitude and latitude coordinates and the coffee pot's status. The status can be either "coffee_available" or "coffee_unavailable".
    b. Handle shutdown messages (details below)
2. Listen for UDP messages:
    a. Keep track of smartwatch's most recent location
3. The main thread should do the following every 1 second until shutdown:
    a. If the manager has received any location messages from the smartwatch, find the nearest coffee pot to the smartwatch that **does** have coffee available (if one exists), and send the smartwatch a "coffee_available" message. If there are no coffee pots with coffee available, send the smartwatch a "coffee_unavailable" message.



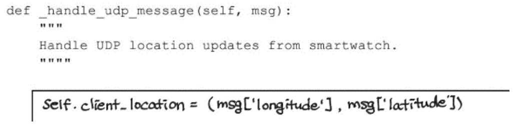
```
class Manager:
    def __init__(self, host, port, client_host, client_port):
        """
        Initialize member variables, start child threads, and
        regularly send updates to the smartwatch. Note: all messages
        sent by the manager should be sent within self._update.
        """

        # Initialize
          self.client_host = client_host
          self.client-port = client_port
          self.host = host, Self.port = port
          self.pots = []
          self.client_location = None
          self.signals = {'shutdown'= False}
        # Starting child threads
          udp_thread = threading.Thread(
                 target = udp_server,
                 args = [
                     self.host, self.port, self._handle_udp_mes,
                     self.signals
                     ]
          )
          udp_thread.start()
          tcp_thread = threading.Thread     "
          tcp_thread.start()

        # Send updates
         while not self.signals['shutdown']:
             If self.client_location:
                 Self.update()
             time.sleep(1)
         udp_thread.Join()
         tcp_thread.Join()
```



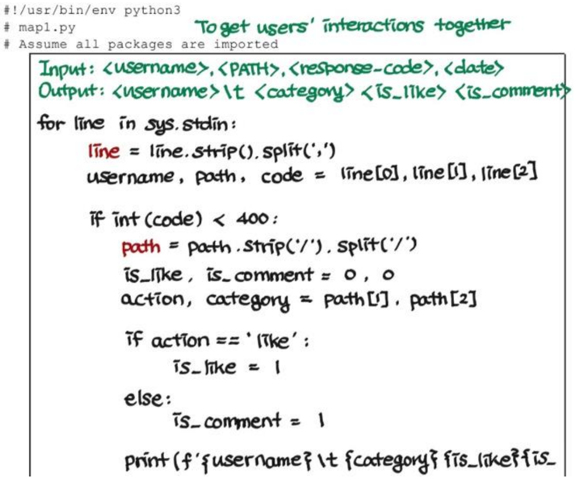
```
def _update(self):
    """
    Send a message to update the smartwatch with the nearest
    coffee pot location with available coffee (if any).
    """

    # Get the nearest pot that does have coffee
    nearest_pot = None
    nearest_dist = None
    for pot in self.pots:
        if pot['status'] == 'coffee available':
            pot_distance = get_distance(
                        Self.client_location,
                        (pot['longitude'], pot['latitude'])
            )
            if not nearest_dist or pot_distance < nearest_dist:
                  nearest_pot = pot
                  nearest_dist = post_distance

    # If nearest pot exists, make msg to send
    if nearest_pot:
        msg = {
            "message_type": "coffee available",
            "longitude": nearest_pot['longitude']
            "latitude" : nearest_pot['latitude']
        }
    else:
        msg = {"message_type": "coffee_unavailable"}
    # Send msg
    tcp_client(self.client_host, self.client_port, msg)
```



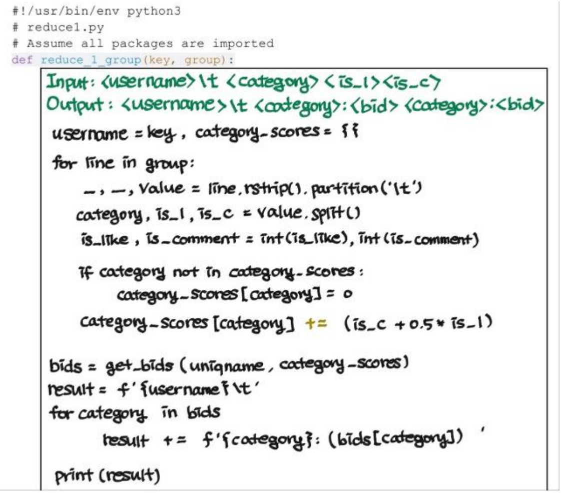
```
def _handle_tcp_message(self, msg):
    """
    Handle TCP shutdown messages and TCP messages from coffee
    pots.
    """

    If msg['message_type'] == 'shutdown':
         self.signals['shutdown'] = True
    elif msg['message_type'] == 'register':
        self.pots.append({
           'host': msg['host'], ....
        })
```



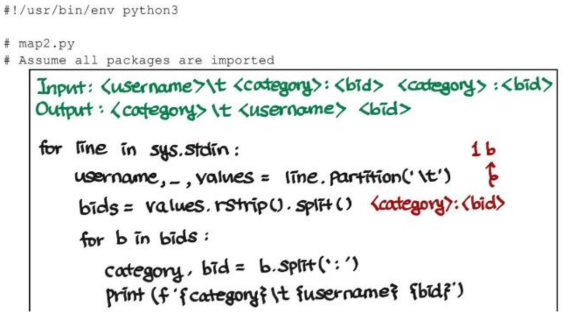
```
def _handle_udp_message(self, msg):
    """
    Handle UDP location updates from smartwatch.
    """

    Self.client_location = (msg['longitude'], msg['latitude'])
```



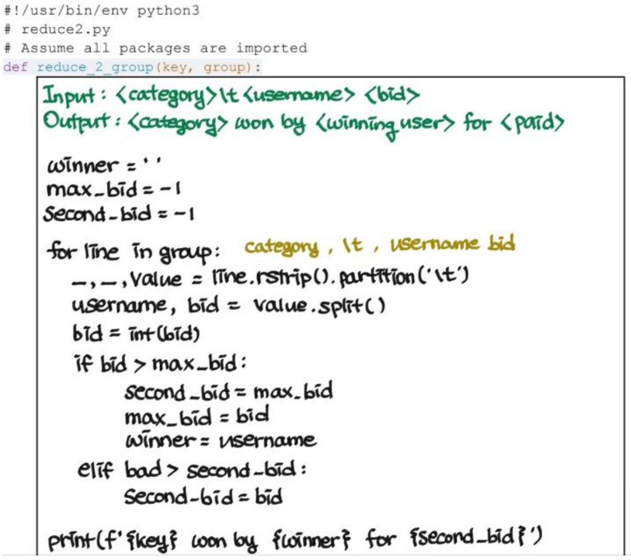
```
#!/usr/bin/env python3
# map1.py                  To get users' interactions together
# Assume all packages are imported
    Input: <username>,<PATH>,<response-code>,<date>
    Output: <username>\t <category> <is_like> <is_comment>
    for line in sys.stdin:
          line = line.strip().split(',')
          username, path, code = line[0], line[1], line[2]

          if int(code) < 400:
              path = path.strip('/').split('/')
              is_like, is_comment = 0, 0
              action, category = path[1], path[2]

              if action == 'like':
                    is_like = 1
              else:
                    is_comment = 1

              print(f'{username}\t {category} {is_like}{is_
```



```
#!/usr/bin/env python3
# reduce1.py
# Assume all packages are imported
def reduce_1_group(key, group):
    Input: <username>\t <category> <is_l><is_c>
    Output: <username>\t <category>:<bid> <category>:<bid>
    username = key, category_scores = {}
    for line in group:
        _, _, Value = line.rstrip().partition('\t')
       category, is_l, is_c = value.split()
       is_like, is_comment = int(is_like), int(is_comment)
       If category not in category_scores:
            category_scores[category] = 0
       category_scores[category] += (is_c + 0.5 * is_l)

    bids = get_bids(uniqname, category_scores)
    result = f'{username}\t'
    for category in bids
         result += f'{category}: (bids[category])  '
    print(result)
```



```
#!/usr/bin/env python3

# map2.py
# Assume all packages are imported
    Input: <username>\t <category>:<bid> <category>:<bid>
    Output: <category> \t <username> <bid>

    for line in sys.stdin:                          1 b
        username, _, values = line.partition('\t')  ↑
        bids = values.rstrip().split()   <category>:<bid>
        for b in bids:
           category, bid = b.split(':')
           print(f'{category}\t {username} {bid}')
```



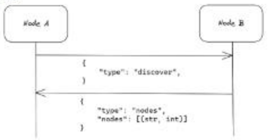
```
#!/usr/bin/env python3
# reduce2.py
# Assume all packages are imported
def reduce_2_group(key, group):
    Input: <category>\t <username> <bid>
    Output: <category> won by <winning user> for <paid>

    winner = ''
    max_bid = -1
    Second_bid = -1

    for line in group:   category , \t , username bid
      _, _, value = line.rstrip().partition('\t')
      username, bid = value.split()
      bid = int(bid)
      if bid > max_bid:
            second_bid = max_bid
            max_bid = bid
            winner = username
      elif bad > second_bid:
            Second_bid = bid

    print(f'{key} won by {winner} for {second_bid}')
```

Discovery Layer

To discover other nodes in the system, your client will:

1. Connect with bootnodes to fetch an initial list of other nodes. We do this for you by calling a `get_bootnodes()` function. This function returns a list of tuples in the format `[(host, port), (host, port), ...]`
2. Send **UDP** `discover` messages to each node, including the bootnodes, so that you can discover new nodes. When you have received at least one response from another node, it becomes "bonded" with your node.
3. When a reply is received from a node, attempt to bond with any nodes in that reply that your client has not already bonded with.
4. Retry sending `discover` requests to a node until you receive a reply. You should wait an appropriate amount of time between sending requests, to allow other nodes to respond.
5. Send the list of bonded nodes to any other client that requests it. You must ignore `discover` requests from clients you have already bonded with.
6. You must not send `discover` messages to nodes you have bonded with.

Here is a diagram of this process, which includes the JSON format used to communicate between the nodes. Note that these are arbitrary nodes; they could be your node, a bootnode, or a neighboring node:



```
def discover(self):
    """
     Continuously send 'discover' messages to nodes that have nc
     yet replied.
    """
```

```
    while True:
        for node in self.to_bond:
            udp_client (node[0], node[1],
                        {
                         "type"= "discover",
                        })
        time.sleep(10)
```

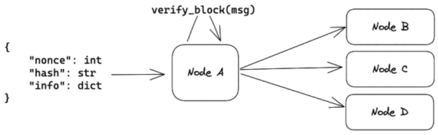
```
def handle_discovery(self, sender_host, sender_port, msg):
       """
       Handle discover messages and node requests
       """
       if msg["type"] == "discover":
```

```
          udp_client (                    sending message
              sender_host,                      back
              sender_port,
              {
               "type" = "nodes",   "우리 이건 bond 야"
               "nodes" = self.bonded
              }
          )

          If (sender_host, port) not in self.bonded:
              Self.bonded.append(               we are going to
                  (sender_host, sender_port))   add that node
                                                that discovered
                                                us
```

```
  elif msg["type"] == "nodes":
```

```
     Self.to_bond.remove ((sender_host, port))
     self.bonded.append ((sender_host, port))

     for node in msg['nodes']:                   sender가
       If node not in self.to_bond               bound된것에
      and node not in self.bonded:               이미 있나 확인해
          Self.to_bond.append(node)
                                                 연결한 노드들과
                                                 연결된 노드들을
                                                 to_bond 에
                                                 넣자.
```

1. When the client receives a block message, it verifies the block and sends the block message to its bonded nodes.
2. We have provided a helper function that takes in a message and verifies it for you.
3. If a block is successfully verified, the client must append the message to the `self.blockchain` member variable and broadcast the message to all nodes it is bonded with.
4. You must ignore messages from nodes you have not bonded with.



```
def handle_p_to_p(self, sender_host, sender_port, msg):
        """
        Receive a message from another node in the network
        """
```

```
  If (sender_host, sender_port) not in self.bonded:
        return

  If verify_block(msg):
      self.blockchain.append(msg)
      for node in self.bonded                  Broadcast !!
          tcp_client (node[0], node[1], msg)
```

Examples of valid inputs:

mbaveja|p5,p2,p3,p1,p4
@280|melodell|p3|4
@485|melodell|p5|10

Output

The final output should be in the following format:

`<project> party: <uniqname1>, <uniqname2>, <uniqname3>`

where uniqnames are listed in descending order by **score**.

Example output:

p5 party: mbaveja, melodell, reiades

The first stage should calculate a **score** per instructor per project. The second stage should assign the top 3 instructors to each spec release party based on their calculated **score** for each project, breaking ties arbitrarily.

```
Inputs  type1: <uniqname>|<highest_ranked>..., <lowest>
        type2: @<post-number>|<uniqname>|<project>|
                   <endorsements>

  when input type is more than 2, differentiate

for line in sys.stdin:      ↗ 앞뒤 필요없는것 자름
    values = line.strip().split('|')
    type 2 input
    If len(values) == 4:
        _, uniqname, project, endorsements = values
                          key                  val
        Print (f"{uniqname}, {project} \t {endorsements}")
    type 1 input
    else:
        uniqname, project_rankings = values
        Project_rankings = project_rankings.split(',')
        project-ranking을 numerical value로 바꿔야해
        for i, project in enumerate(project_rankings):
            print (f"{uniqname}, {project} \t {i + 1} rank")
                                              왜 i+1 ?
                                         i는 0부터 시작하고
                                     제일높은등수의 proj 앞에1
```

```
: reduce_1_group(key, group): → 같은 key들의 집합
```

```
    Input: <uniqname>, <project> \t <endorsements>
           <uniqname>, <project> \t <numerical_ranking> rank
  uniqname, project = key.split(",")
  num_answers = 0
  num_endorsments = 0          we need to calculate 'score'

  for line in group:                          ↗ input type 1과2
                                                구분하기위해
      Values = line.partition("/t")[2].split()
  ('<uniqname>.<project>', '/t', '<endorsements>'
                                 '<num_ranking> rank')
mel|p4|13  Piazza post (endorsements)
mel|p4|7   If len(values) == 1:
               endorsements = int(values[0])
               num_answers += 1
               num_endorsments += endorsements
           Ranking input
           else:
               rank, _ = values
               rank = int(rank)
  Calculate score
  avg_endorsements = num_endorsements/num_answ
  score = (1/rank) * (num_answ + avg_end **2)
  print (f"{uniqname}, {project} \t {score}")
```

```
: keyfunc(line):
return line.partition("\t")[0]
```

```
def main():
   for key, group in itertools.groupby(sys.stdin, keyfunc):
      reduce_1_group(key, group)

if __name__ == "__main__":
   main()

#!/usr/bin/env python3
# map2.py
# Assume all packages are imported
```

```
    Input: {uniqname}, {project} \t {score}
    for line in sys.stdin:
        key, values = line.split("\t")  (= line.split())
        uniqname, project = key.split(",")
        Score = values

        Print (f"{project} \t {uniqname}, {score}")
```

```
#!/usr/bin/env python3
# reduce2.py
# Assume all packages are imported
def reduce_2_group(key, group):
```

```
    Input: {project} /t {uniqname} {score}
    project = key
    instructors = []

    for line in group:
        Values = line.partition("/t")[2]
        uniqname, score = values.split(",")
        Instructors.append((float(score), uniqname))
                            score를 sort 할거니까 앞에1

    Instructors = sorted (Instructors, reverse = True)
                                            decending
                  uniqname
    top_3 = [ i[1] for i in instructors [0:3] ]
    print(f"{project} party: {top_3[0]}, {top_3[1]}, {top_3[2]})
```

```
def keyfunc(line):
   return line.partition("\t")[0]

def main():
   for key, group in itertools.groupby(sys.stdin, keyfunc):
      reduce_2_group(key, group)

if __name__ == "__main__":
   main()
```