1. Answer:

Algorithms:

We use two dictionaries s and p_num. The s will store the shortest path from v to the current node, and p_num stores the number of the shortest path to current node.

The we will use BFS to traverse the graph:

If the node hasn't been visited, we will assume the distance from the node to v shortest;

If the distance to v is less than current path, we will update the path to shorter one;

If the distance to v is the same as current path, we can confirm we find another shortest path, so we add p_num[node] with its parent node's path p_num[start].

```
Path_num (G = (V,E), v,w ∈ V).
    dictionary s{v:0} for v in G.          (s — shortest path to node
    dictionary p_num{v:0} for v in G.       p_num — number of the shortest path)
    Initialize p_num[v] = 1
    queue [v].
    while size(queue) > 0 :
        start = dequeue(queue)
        distance = s[start] + 1            (the start's node to start's distance is 1)
        for node in G[start] :
            If (s[node]==0 and node != v) or distance < s[node]:
                                            (update shortest path)
                s[node] = distance
                p_num[node] = p_num[start]
                enqueue(node)
            elif s[node] == distance :
                p_num[node] = p_num[node] + p_num[start]     (add shortes path's number)
        end for
    end while
    return p_num[w]
```

Running Time: $O(n+2m)$

2. Answer:

Algorithms:

We will implement dijkstra to achieve the question.

Every time we will focus on the node with shortest path to the src.

And we will update all its neighbor nodes's path.

If there is a shorter path, we update the node with shorter one

If there is a path with same distance, means we find another shortest path.

```
Dijkstra ( G = (V, E) , src ∈ V ):
    visited = set (src)
    minv = src                              (minv is the node with shortest path to src)
    dictionary dis {node : INF} for node in G        ( dis — distance from node to src
    dictionary P_num {node : 0} for node in G          P_num — shortes path's number)
    P_num [src] = 1
    dis [src] = 0
    while size (visited) < size (G):
        visited. add (minv)
        for node in G [minv]:                                  (find shorter path)
            If dis [minv] + G [minv] [node] < dis [node]:
                update dis [node] to dis[minv] + G[minv][node]
                P_num [node] = P_num [minv]
            elif dis [minv] + G[minv] [node] == dis [node]:    (find another shortest path)
                P_num [node] = P_num [node] + P_num [minv]
        end for
        new_minv = INF
        for key in dis's keys:                   (find the node with shortest path to src)
            If key in visited:
                keep loop
            If dis[key] < new_minv:               (update minv)
                new_minv = dis [key]
                update minv ~~with~~ to key
        end for
    end while
    return P_num.
```

Running time: ~~O(n + m)~~ O(nm)

3. Answer:

Algorithms:

Assume there are i hotels we can stay.

For each hotel j (from 0 to $i-1$), the minimum penalty is $OPT(j)$, and the cost to i is $[200-(a_j-a_i)]^2$

So $OPT(i) = \min\{OPT(j) + [200-(a_j-a_i)^2]\}$ for j from 0 to $i-1$

Base case: $OPT(0) = 0$

for i from 1 to n:

  for j from 0 to $i-1$:

    $OPT(i) = \min\{OPT(i), OPT(j) + (200-(a_j-a_i)^2)]\}$

return $OPT(n)$.

Running time:

$\Sigma O(i) = O(\frac{n(n-1)}{2}) = O(n^2)$.

4. Answer:

Algorithms:

The total waiting time $= t_1 + (t_1+t_2) + (t_1+t_2+t_3) + \cdots + (t_1+t_2+\cdots+t_n)$

$= nt_1 + (n-1)t_2 + \cdots + t_n = \sum_{i=1}^{n}(n+1-i)t_i$

So we just need to serve the custom in increasing order of $t(i)$

Correctness: Assume there is a customer i served before j and $t(j) < t(i)$ $(j>i)$.

$T_{assume} - T_{opt} = [(n+1-j)t(i) + (n+1-i)t(j)] - [(n+1-i)t(i) + (n+1-j)t(j)]$

$= (i-j)[t(i)-t(j)]$.

$\because i<j,\ t(i) > t(j) \Rightarrow T_{assume} - T_{opt} < 0$.

the assument is incorrect, so the algorithms is optimal

Merge Sort (T):

```
merge(lst1, lst2):
    if len(lst1) == 0:
        return lst2
    elif len(lst2) == 0:
        return lst1
    elif lst1[0] > lst2[0]:
        return [lst2[0]] + merge(lst1, lst2[1:])
    else:
        return [lst1[0]] + merge(lst1[1:], lst2)
```

```
divide (T):
  if len(T) ==1 or len (T) == 0:
        return T

  mid = len(T) / 2
  left = divide (T[: mid])
  right = divide (T [mid: ])
  return merge(left, right)
return divide (T) .
```

∴ Answer:

Algorithms: At first we create a n×n matrix. I. I[i][j] represents the ~~imbalance~~ deviation of the array A[i : j+1].

Then we need another matrix OPT (k+1) × n. OPT[i][j] represents the imbalance ~~of~~ with i groups and j numbers.

$$OPT[i][j] = \begin{cases} I[0][j] & \text{if } i = 0 \\ \min\{\{\max(OPT[i-1][j_k],\ I[j_k+1][j])\}\} & \text{for } i < j_k < j \end{cases}$$

Then OPT[k][n-1] will be the imbalance of A.

```
Imbalance (A, n, k):
  Initialize I = n×n zero matrix
                                  (update I with total of A[i: j+1])
  for i in range (n):
     for j in range(n):
        if i == j:
           I[i][j] = A[j] ~~- average~~

        else:
           I[i][j] = A[j] + I[i][j-1] ~~- average~~
  average = sum (A) / (k+1)
  I = |I - average| for all element in I.   (update I with imbalance for A[i: j+1])
  Initialize OPT = (k+1) × n zero matrix.
  for j in range (n):
     OPT[0][j] = I[0][j]                 (If there are only one group, we just put
                                           all data there).
  for i in range (k+1):
     for j in range (i+1, i+n+1 -k):
```

```
m = 0
for jk in range(i, n-1):
    m = min(m, max(OPT[i-1][jk], I[jk+1][j]))
  OPT[i][j] := m.
 end for.
end for
return OPT[k][n-1]
```

(If there are more than one group we compare OPT[i-1][jk] with I[jk+1][j], then choose the minimum one between this and old m)

Running time: $O(kn^2)$

(b) Algorithms: Implement the same I matrix.

$$OPT[i][j] = \begin{cases} I[0][j] & \text{if } i = 0. \\ \min\{OPT[i-1][j_k] + I[j_k+1][j]\} & \text{for } i < j_k < j. \end{cases}$$

```
for j in range(n):
    OPT[0][j] = I[0][j]
for i in range(k+1):
    for j in range(i+1, i+n+1-k):
        sum = 0
        for jk in range(i, n-1):
            sum = min(sum, OPT[i-1][jk] + I[jk+1][j])
return sum
```

Running time: $O(kn^2)$

Haopeng Zhang
hz2558