HW6

Intuitively we have two main idea of this problem. One is to find the smallest difference each time, and the other is to pair the two smallest items each time.

1. The following sets are not yielding optimal results with the algorithm described.
   skiers = [3, 4]
   skis = [1, 4, 7]
   With the algorithm of Prof. Lai, it should give a result = |4-3| + |1-4| = 4, while the optimal result should be 2.

2. Prove the conjecture. I'll only provide one analysis according to HW6 question. There are 6 possible cases in total.
   - Case $x < y < s_y < s_x$,
   - Case $x < s_y < s_x < y$,

     If we match s_x to x and s_y to y, the difference equals to $(s_x - x) - (s_y - y) = (s_x - s_y) + (y - x)$

     If we match s_x to y and s_y to x, the difference equals to $(s_y - x) - (s_x - y) = (s_y - s_x) + (y - x)$

     Since $(s_x - s_y) > 0 > (s_y - s_x)$, the upper one is obviously larger than the lower one, which means the second match is better than the first. That is to say, matching longer skis to taller person is better than matching longer to shorter.
   - Case $x < s_y < y < s_x$,
   - Case $s_y < x < y < s_x$,
   - Case $s_y < s_x < x < y$,
   - Case $s_y < x < s_x < y$,

3. If the numbers of skis and skiers are the same, then the basic idea is to sort both arrays and assign each person with the length corresponding with his index.
   Since we've derived from question 2 that assigning shorter skis to shorter person is better than taller person, in arrays with same length, the only way to guarantee the match is to correlate elements in order. Otherwise, we'll run into overlapped order and contradict the conclusion in question 2.
   The runtime for sorting both length n arrays are in 2*O(nlogn), and the matching process need to go through both array which takes 2*O(n). Thus, the total runtime for this algorithm is O(nlogn).

4. Algorithms

4.1 For this recursive approach, I will rename the function ***partC*** as ***minimum_difference***. They take the same input. The implementation of minimum_difference is pasted below

```
1.  def minimum_difference(array1, array2):
2.      array1 = sorted(array1)
3.      array2 = sorted(array2)
4.      sums = 0
5.      for i in range(len(array1)):
6.          sums += abs(array1[i] - array2[i])
7.      return sums
```

The basic idea of this function is to calculate the minimum sum of disparities of two sorted arrays with same length. As we've discussed in question 3, we need to sort both arrays and sum up the differences between each pair with same indices.

Then in the recursion, the base case is to

- Reject invalid cases - the number of skiers is larger than skis and cannot allocate, just return a large number.
- Return valid cases - the number of skiers equals to skis, and we can call function minimum_difference.

And the recursive step is to remove one element from the skis array, judge if its length equals to the length of skiers array, and move forth. The recursion function below is written in Python style, and need to use together with the function minimum_difference above. It'll return the final result value of minimum sum of disparities.

```
1.  def ski_matching_recursive(skiers, skis):
2.      p = len(skiers)
3.      h = len(skis)
4.      # base cases
5.      if p > h:
6.          return 9999  # invalid case
7.      if p == h:
8.          return minimum_difference(skiers, skis)
9.      min_value = 9999
10.     # recursive steps, pretty ugly
11.     for i in range(h):
12.         temp_skis = skis.copy()
13.         temp_skis.pop(i)
14.         min_value = min(min_value, ski_matching_recursive(skiers, temp_skis))
15.     return min_value
```

4.2 Since I've written the code in Python, I'll just keep brief in my description. The DP table should keep track of the minimum difference between skis length and given numbers of people. For example, we have
skiers = [3, 4]
skis = [1, 4, 7]

	0	1	2	3
0	0	0	0	0
1	9999	2	1	1
2	9999	9999	2	2

Where the rows are number m of skiers[0:m] and columns are n number of skis[0:n]. When skiers are more than skis, I just set those invalid cases as 9999 so that they won't be the minimum sum of difference.

The algorithm firstly set row 0 and column 0 with zeros. Then for each row we loop through all of its elements and find its minimum sum. The lower right corner will be the final result. You can just read the code below to get an idea of how it works.

```
1.  def ski_matching_dp(skiers, skis):
2.      p = len(skiers)
3.      h = len(skis)
4.      d = [[0 for _ in range(h+1)] for _ in range(p+1)]  # init
5.      for i in range(p+1):
6.          d[i][0] = 0
7.      for j in range(h+1):
8.          d[0][j] = 0
9.      for i in range(p+1):
10.         for j in range(h+1):
11.             if j < i:
12.                 d[i][j] = 9999
13.             else:
14.                 d[i][j] = min(d[i][j-1], d[i-1][j-1] + abs(skiers[i-1] - skis[j-1]))
15.     print(d)
16.     Find_optimal_path(d, skiers, skis)  # for q4.4
17.     return d[p][h]
```

4.3 The runtime of this dynamic programming algorithm is the sum of all the steps.
For sorting the two arrays, we have O(nlogn+mlogm);
For looping the DP table, we have O(mn).
It does depend on the scale of m or n, but roughly speaking the overall complexity is O(mlogm + nlogn + mn). Still, if we do not loop through invalid cases where number of people is larger than number of skis, we might cut down some computation, but that implementation would be counter-intuitive and I'd rather not delve into

that.

4.4 Like most of the dynamic programming problems, we could use our DP table to back-trace and find a route backwards as the optimal path. The operation is just like an inverse operation of the solver above.
We start from the bottom right corner d[n][m], and then go left and upward. If the left element is same, it means that we did not make that match. Otherwise we keep record of it and then go to upper left element, until we reach the upper left corner. The Python code for this question is:

```
1.  def find_optimal_path(d, skiers, skis):
2.      n = len(skiers)
3.      m = len(skis)
4.      result_dict = dict()
5.      i = n
6.      j = m
7.      while i > 0:
8.          while j > 0:
9.              if d[i][j] == d[i][j-1]:
10.                 j -= 1
11.             else:
12.                 result_dict[skiers[i-1]] = skis[j-1]
13.                 i -= 1
14.                 j -= 1
15.     print(result_dict)
```

A simple test case with code above:
skiers = [5, 7, 8, 9, 10]
skis = [1, 2, 4, 5, 6, 7, 9, 11, 12]
{10: 11, 9: 9, 8: 7, 7: 6, 5: 5}
Sum is 3.