Homework #5

Question #7

We will use the divide and conquer approach to solve the closest pair points problem for this question."

```
closestSplitPair(Px, Py, d)
    Sy = []
    for p in Py:
        if Px[n/2][0] - d <= p[0] <= Px[n/2][0] + d
            Sy.append(p)

    for i = 1 to |Sy| - 1
        for j = 1 to 7
            let p, q = Sy[i], Sy[i-j]
            if d(p,q) < best
                best pair = (p,q)
                best = d(p,q)
```

```
closestPair(Px, Py)
    L = Px[:n/2]
    R = Px[n/2:]

    (p1, q1) = closestPair(Lx, Ly)
    (p2, q2) = closestPair(Rx, Ry)
    d = min{d(p1,q1), d(p2,q2)}
    (p3, q3) = closestSplitPair(Px, Py, d)
    return best of (p1,q1), (p2,q2), (p3,q3)
```

Time complexity

$T(n) = 2T(n/2) + O(n) + O(n) + O(n)$

$T(n) = 2T(n/2) + O(n)$

$T(n) = O(n \log n)$

Question #2

we solve the convex hull problem with divide and conquer.

```
def convex_hull(S)
  if len(S) <= 5
      return brutehull(S)

  left, right = [], []
  start = int(len(S)/2)
  for i in range(start):
      left.append(S[i])

  for i in range(start, len(S)):
      right.append(S[i])

  left_hull = convex_hull(left)
  right_hull = convex_hull(right)

  return merge(left_hull, right_hull)
```

Time complexity

The merge of left and right convex hull take O(n) time and as we are dividing the points into two equal parts, so the time complexity of the above algorithm O(nlogn)

recurrence equation

$$T(n) = 2T(n/2) + O(n)$$

Best case $O(n \log n)$

Worst case

Line divides into extremely unbalanced parts and has the recurrence.

$$T(n) = T(n-1) + O(n)$$

$$T(n) = O(n^2)$$

Question #3

we solve the problem with min-cost distance method

```
def minDis(s1, s2, len_str1, len_str2, dp)

    if (len_str1 == 0)
        return len_str2
    if (len_str2 == 0)
        return len_str1

    if (dp[len_str1][len_str2] != -1
        return dp[len_str1][len_str2]


    if (s1[len_str1 - 1] == s2[len_str2 - 1]):
        if (dp[len_str1][len_str2] == -1):
            dp[len_str1][len_str2] = minDis(s1, s2, len_str1 - 1,
                len_str2 - 1, dp)
            return dp[len_str1][len_str2]
        else:
            dp[len_str1][len_str2] = dp[len_str1 - 1][len_str2 - 1]
            return dp[len_str1][len_str2]

    else:
        if (dp[len_str1 - 1][len_str2] != -1):
            m1 = dp[len_str1 - 1][len_str2]
        else:
            m1 = minDis(s1, s2, len_str1 - 1, len_str2, dp)

        if (dp[len_str1][len_str2 - 1] != -1):
            m2 = dp[len_str1][len_str2 - 1]
        else
            m2 = minDis(s1, s2, len_str1, len_str2 - 1, dp)

        if (dp[len_str1 - 1][len_str2 - 1] != -1):
            m3 = dp[len_str1 - 1][len_str2 - 1] + 2
        else :
            m3 = minDis(s1, s2, len_str1 - 1, len_str2 - 1, dp) + 2
            m3 = m3 - 2   // for replace
        dp[len_str1][len_str2] = 1 + (min(m1, min(m2, m3)))
```

// For add and delete

for replace

Time Complexity

m = len(str1)
n = len(str2)

$T(n) = O(n.m)$

Question #4

We solve problem Traveling salesman problem using Dynamic programing

```
def calc_discount(i, mask):
    if mask==(1<<n)-1:          // base case
        return dist[i][0]
    // memorization
    if memo[i][mask] != -1:
        return memo[i][mask]

    res=-1

    // traveling all nodes in mask
    for j in range(1,n+1):
        if(mask&(1<<j)) != and j != 1:
            res = max(res, fun(j,mask&(~(1<<i)))+dist[j][i])

    memo[i][mask]=res          // add to max function
    return res
```

Time complexity: $O(n^2 \cdot 2^n)$ where $O(n^2 2^n)$ are maximum maximum number of unique subproblems/states and $O(n)$ for transition (through for loop as in code) in every states.

Note: We are modifying the function that calculates the minimum to make it maximize.

Question #5

```
def greedy_algorithm(antenna_locations):
    antenna_locations.sort(key=lambda x: x[0])
    selected_antennas = []

    i = 0
    while i < len(antenna_locations):
        current_antenna = antenna_locations[i]
        selected_antennas.append(current_antenna)
        while i+1 < len(antenna_locations) and (antenna_locations[i+1][0]
            <= current_antenna[1]):
            i += 1
            current_antenna = antenna_locations[i]

        i += 1

    return selected_antennas
```

1. First step Sorting: Sorting antennas based on the starting point of their coverage areas. This ensures that the left end of the coverage area of each antenna is greater than that of other antennas.

2. Second step: Coverage Area selection: Start selecting antennas from sorted list. After selecting each antenna record the end point of its coverage area

3. Third step: Selecting the current antenna
Before selecting the next antenna, check if the starting point of this antenna is greater than the endpoint of the coverage area of the previously selected antenna. If it is greater selected this antenna otherwise skip to the next one

(Fourth step: Selecting maximum number of banners

- Repeat these steps until reaching the maximum number of banners.

Time complexity

Algorithm is O(n log n) sorting operations takes n log n time and then banners are selected in a single pass.