CS577: Introduction to Algorithms (Spring 2018) **Scribe:** Xiahe Liu

HW2 Solutions **Date:** 2/8/2018

1(a) In this problem, we will assume that addition on peg labels is mod 3, so e.g. when we say move from peg P to peg $P + 2$, we mean $(P + 2)$ *mod* 3.

As the hint mentioned, to move n disks from peg P to peg $P + 2$, we can write two different procedures, one of which moves disks one place counterclockwise, and the other two places counterclockwise. We will make recursive calls to each one from the other, as necessary. Let us call these functions MoveOnePlace and MoveTwoPlaces respectively.

Let's begin by trying to move disks two places counterclockwise from peg P to peg $P+2$. At some point in our procedure we will have to move the nth disk from P to $P + 1$. In order for this to happen, the first $n - 1$ disks should all be on $P + 2$. This is a recursive call to MoveTwoPlaces on $n - 1$ disks. Then, we will have to move the nth disk from $P + 1$ to $P + 2$. In order for this to happen, the first $n-1$ disks should all be on P. But moving the $n-1$ disks from their most recent position $P+2$, to the new position P will need a recursive call to MoveOnePlace on $n-1$ disks. We can then complete the process by moving the top $n-1$ disks from P to $P+2$ using a recursive call to MoveTwoPlaces.

Note that all of our recursive calls are made to move the top $n-1$ disks. As long as these recursive calls are internally consistent (i.e. they don't make any illegal moves), they are consistent with the rest of the algorithm. In particular, we don't need to worry about the position of the nth disk while performing the recursive calls because any of the $n-1$ can be placed legally on the nth disk.

We now describe the procedure formally.

```
MoveTwoPlaces(n, P, P+2):
1. If $n = 0$, then we are done
2. MoveTwoPlaces(n-1, P, P+2): recursively move top $(n-1)$ disks from $P$ to $P+2$ via $P+1$
3. Move the n-th disk from P to P+1
4. MoveOnePlace(n-1,P+2, P): recursively move top $(n-1)$ disks from $P+2$ to $P$ via $P+1$
5. Move the n-th disk from P+1 to P+2
6. MoveTwoPlaces(n-1, P, P+2): recursively move top $(n-1)$ disks from $P$ to $P+2$ via $P+1$
```

The procedure MoveOnePlace can be described similarly:

```
MoveOnePlace(n, P, P+1):
1. If $n = 0$, then we are done
2. MoveTwoPlaces(n-1, P, P+2): recursively move top $(n-1)$ disks from $P$ to $P+2$ via $P+1$
```

3. Move the n-th disk from P to P+1
4. MoveTwoPlaces(n-1, P+2, P+1): recursively move top $(n-1)$ disks from $P+2$ to $P+1$ via P

1(b) Let $T_1(n)$ represent the running time of MoveOnePlace with size n and $T_2(n)$ is the running time of MoveTwoPlaces, which is the running time of our algorithm. MoveTwoPlaces has two recursive calls of MoveTwoPlaces, one recursive call of MoveOnePlace and two moves. MoveOnePlace has two recursive calls of MoveTwoPlaces and one move. We have the following recurrences:

$$T_2(n) = 2T_2(n-1) + T_1(n-1) + 2$$

$$T_1(n) = 2T_2(n-1) + 1$$

thus, we have

$$T_2(n) = 2T_2(n-1) + 2T_2(n-2) + 3$$

The recurrence above can be solved by "guess and verify". The base cases are $T_2(1) = 2$ and $T_2(2) = 7$. We will prove that $T_2(n) \leq a \cdot 3^n$ for some $a > 0$

Proof: By induction on $n-1$:

- Base case: When $n = 1$, we have $T_2(1) = 2$, $T_2(1) = 2 \leq a \cdot 3^1$, for $a \geq \frac{2}{3}$ and when $n = 2$, we have $T_2(2) = 7$, $T_2(2) = 7 \leq a \cdot 3^2$, for $a \geq \frac{7}{9}$
- Inductive Hypothesis: Assume that for all $k < n$, $T_2(k) < a \cdot 3^k$, where $n \geq 3$ and a is a constant number
- Inductive Step:

$$\begin{aligned} T_2(n) &= 2T_2(n-1) + 2T_2(n-2) + 3 \\ &\leq 2 \cdot a \cdot 3^{n-1} + 2 \cdot a \cdot 3^{n-2} + 3 \\ &= 8 \cdot a \cdot 3^{n-2} + 3 \end{aligned}$$

To achieve $T_2(n) \leq a \cdot 3^n$, we need to guarantee that

$$\begin{aligned} 8 \cdot a \cdot 3^{n-2} + 3 \leq a \cdot 3^n = 9 \cdot a \cdot 3^{n-2} \\ \frac{1}{3^{n-3}} \leq a \end{aligned}$$

We choose $a = 1$, then for any $n \geq 3$, we can conclude that $T_2(n) \leq a \cdot 3^n$, thus, $T_2(n) \in O(3^n)$

■

2(a) Part (a) can be solved by reducing the problem to the problem of counting inversions. For any two line segments $l_i = (p_i, q_i)$ and $l_j = (p_j, q_j)$, suppose $p_i < p_j$, then there will be an intersection *iff* $q_i > q_j$. Likewise, if $p_i > p_j$, there is an intersection between l_i and l_j *iff* $q_i < q_j$. This leads to the following algorithm, where $P = \{p_1, p_2, ..., p_n\}$ and $Q = \{q_1, ..., q_n\}$ are the sets of points. And Step 3 is implemented using the algorithm for counting inversions done in class.

```
INTERSECTIONS(P, Q):
1. Sort P in ascending order
2. Rearrange $Q$ according to sorted $P$, make sure that $p_i$ and $q_i$ have the same index
3. Return number of inversions on $Q$
```

Proof of Correctness:
First of all, the first two steps will not influence the result of this problem, since we still have the same line segments as that before sorting and rearrangement.
We will now prove that the number of inversions on rearranged list Q is exactly the number of intersections among all the line segments. As we mentioned before, P is already sorted in ascending order, so we have $p_i < p_j$ for any $i < j$, and there will be an intersection if $q_i > q_j$. Thus, we can conclude that there will be an intersection *iff* $i < j$ and $q_i > q_j$, which is exactly the number of inversions on rearranged Q.

Running Time:
The running time of sorting and rearrangement is $O(n \log n)$ and $O(n)$, respectively. And $O(n \log n)$ is required to get the number of inversions. Overall, the running time of the algorithm is $O(n \log n)$.

2(b) Part (b) can be solved by leveraging the algorithm from Part (a), the basic idea is to split the circle into two parts C_1 and C_2, each of which contains half the points. As shown in Fig. 2.1, C_1 contains the points 1 to n start from the origin, and C_2 contains points $(n + 1)$ to $2n$. Let L_1 represent all the line segments with both endpoints in C_1. Let L_2 contain all the line segments with both endpoints in C_2. Let L represent the remaining line segments, these are lines with one endpoint in C_1 and the other endpoint in C_2.

Given the sets of line segments L, L_1, L_2, intersections can be found in 4 cases,
1. Both line segments are in L_1
2. Both line segments are in L_2
3. Both line segments are in L
4. One line segment is in L, and the other is in either L_1 or L_2

Observe that each of L_1 and L_2 contains at most $\frac{n}{2}$ line segments. So the recursive calls will work on smaller subproblems.
For case 1, we can count the intersections between lines in L_1 by performing a recursive call on this set of line segments. Similarly, we can count the intersections between lines in L_2 for case 2.
For case 3, imagine taking the arcs of the circle in either half and stretching it out into a line. Then, this problem looks like the problem we encountered in Part (a).
For case 4, we need to count the intersections which occur between a line segment in L and another

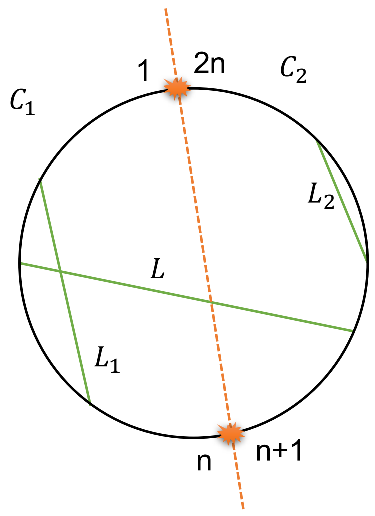

Figure 0.1: An Example of Circle Separation.

in L_1 or L_2. Consider some line segment $(p_i, q_i) \in L_1$ and some line segment $(p_j, q_j) \in L$. Suppose p_j lies in C_1 and q_j in C_2. Then, (p_i, q_i) and (p_j, q_j) intersect *iff* p_j lies between the points p_i and q_i in C_1. This implies that counting the number of endpoints of line segments in L that lie between p_i and q_i gives us the number of intersection of that line with line segments in L. To count the number of intersections, create an array containing the endpoints from L which appear on C_1, and sort them in the order in which they appear on C_1. Perform a binary search to determine where p_i would be inserted into this array, and then perform another binary search to determine where q_i would appear. This allows us to deduce the number of endpoints from L that are between p_i and q_i, and so we know how many intersections occur between L_1 and L. Repeat this process with the other line segments in L_1 to count the remaining intersections between L_1 and L. Similarly, the intersection between L and L_2 can be counted. More formally:

Count(P, Q):

1. Obtain $P_1, Q_1, P_2, Q_2, P_L, Q_L$. Define $P_1 = \{$all points in $P \leq n$ such that their counterpart in $Q \leq n\}$, $P_L = \{$all points in P or $Q \leq n$ whose counterpart is larger than $n\}$. Likewise, define the other lists. Note that, P_1 and Q_1 are the endpoints of L_1, P_2 and Q_2 are the endpoints of L_2, P_L and Q_L represent L.
2. Initialize counter $\leftarrow 0$
3. counter $\leftarrow$ counter + Count(P_1, Q_1) + Count(P_2, Q_2)
4. counter $\leftarrow$ counter + Intersections(P_L, Q_L), add the number of intersections within L by applying the algorithm in Part (a).
5. counter $\leftarrow$ counter + countCross(P_1, Q_1, P_L, Q_L) + countCross(P_2, Q_2, P_L, Q_L), add the number of intersections between L_1 and L, L_2 and L, and the helper function countCross will be introduced later.

```
6. Return the counter
```

Observe that some of the steps above require renumbering of some of the points, we omit those details. The following shows the procedure to get the number of intersection between L and $L1/L2$, which is case 4. We show the steps on L_1 and L in the following, it will be similar for L_2 and L.

```
countCross($P_1, Q_1, P_L, Q_L$):
1. Create a sorted array $C^{'}$ of all endpoints from $L$ which appear in $C_1$, namely $P_L$
2. Initialize counter $\leftarrow 0$, which records the number of intersections between $L_1$ and $L$.
3. for each $l_i = (p_i, q_i) \in L_1$
    a. Perform a binary search on $C^{'}$ to determine where $p_i$ would be inserted.
    b. Perform a binary search on $C^{'}$ to determine where $q_i$ would be inserted.
    c. Determine the number of endpoints between these two insertion points and add this to the counter
4. Return the counter
```

Proof of Correctness:

Claim 0.1 countCross*(P_1, Q_1, P_L, Q_L)/*countCross*(P_2, Q_2, P_L, Q_L) returns the number of intersections between L_1/L_2 and L.*

Proof: For any line segment $l_i = (p_i, q_i) \in L_1$, we can reasonably assume that $p_i < q_i$. And similarly, assume that $p_j < q_j$ for any line segment $l_j = (p_j, q_j) \in L$. There will be an intersection *iff* $p_i < p_j < q_i$. Step 1 of countCross(L_1, L) returns a sorted list $C^{'}$ that contains all the $p_j \in C_1$, and $(p_j, q_j) \in L$. Then the number of the points in $C^{'}$ that between p_i and q_i is the number of intersections between line $l_i \in L_1$ and L. And the binary searches in Step 3.a-3.b can return the correct number of the intersections above. The total number of intersections between L_1 and L can be obtained by repeating the above process for each line segment in L_1.
Similarly, we can prove that Claim 1 holds for countCross(P_2, Q_2, P_L, Q_L). ∎

Claim 0.2 *All the intersections can be covered by the 4 cases in our algorithm.*

Proof: For any pair of line segments (l_i, l_j), l_i can be in either L_1, L_2 or L, then there could be 9 cases of (l_i, l_j) according to their positions. We can enumerate all the position combinations and conclude that they are covered by the 4 cases in our algorithm.
1. $l_i \in L_1, l_j \in L_1$: counted by case 1
2. $l_i \in L_1, l_j \in L_2$: there is no intersection
3. $l_i \in L_1, l_j \in L$: counted by case 4
4. $l_i \in L_2, l_j \in L_1$: there is no intersection
5. $l_i \in L_2, l_j \in L_2$: counted by case 2
6. $l_i \in L_2, l_j \in L$: counted by case 4
7. $l_i \in L, l_j \in L_1$: counted by case 4
8. $l_i \in L, l_j \in L_2$: counted by case 4

9. $l_i \in L, l_j \in L$: counted by case 3
Thus, we can conclude that all the possible intersections will be counted by the 4 cases in our algorithm. ■

Claim 0.3 *Given a set of $2n$ points C,* Count*(P, Q) returns the number of all the line segment intersections.*

Proof: By induction on n:

- Base case: If $n \leq 1$, the algorithm correctly returns 0.
- Inductive Hypothesis: Assume that for all $k <= n$, the algorithm returns the number of intersections correctly.
- Inductive Step: For the problem with $2(n+1)$ points, assume that all the points are divided into two sets C_1 and C_2, the size of them is $2n_1$ and $2n_2$ separately, and $n_1 + n_2 = n+1$. L_1 represents the line segments that both end points are in C_1, L_2 represents the line segments that both end points are in C_2 and L contains all the line segments that has one end point in C_1 and the other in C_2. For case 1 and 2, the size of the subproblem $|P_1| \leq \frac{|C_1|}{2} \leq n$ and $|P_2| \leq \frac{|C_2|}{2} \leq n$. Appeal to I.H., the line segments within L_1 and L_2 are correctly returned by Count(P_1, Q_1) and Count(P_2, Q_2), when $n_1 \geq 1$ and $n_2 \geq 1$. With Part (a) and Claim 1, the number of intersections in case 3 and 4 can be obtained correctly. And all the intersections are covered by the 4 cases above according to Claim 2. Therefore, the algorithm will return the number of intersections correctly when we have $2(n+1)$ points.

■

Running Time:
Step 1 and 2 in Count cost $O(n)$ in total, because we can do a linear scan on the input lists and place the points into the lists defined above. Step 3 has two recursive calls and the running time will be $2T(\frac{n}{2})$. And the running time of Step 4 is $O(n \log n)$ according to Part (a). For Step 5, as described in countCross, we have $O(n)$ line segments in L_i, and each line segment in L_i performs 2 binary searches, whose running time is $O(\log n)$. So the running time of Step 5 is $O(n) \cdot O(\log n) = O(n \log n)$. Therefore, we can conclude that overall running time is $T(n) = 2T(\frac{n}{2}) + O(n \log n)$. The recurrence can be solved by unrolling

$$\begin{aligned}
T(n) &\leq 2T(\frac{n}{2}) + cn\log n = 2(2T(\frac{n}{4}) + c\frac{n}{2}\log\frac{n}{2}) + cn\log n \\
&= 4T(\frac{n}{4}) + cn\log\frac{n}{2} + cn\log n \leq 4T(\frac{n}{4}) + 2cn\log n \\
&= 8T(\frac{n}{8}) + cn\cdot(\log n + \log\frac{n}{2} + \log\frac{n}{4}) \leq 8T(\frac{n}{8}) + 3cn\log n \\
&\dots \\
&\leq 2^i T(\frac{n}{2^i}) + i \cdot cn\log n
\end{aligned}$$

The process stops when it reaches the base case $T(1) = 0$ (no intersection for only one line segment), then we have $\frac{n}{2^i} = 1$ and $i = \log n$. Therefore we get $T(n) \leq 0 + cn\log^2 n$, or $T(n) \in O(n\log^2 n)$.