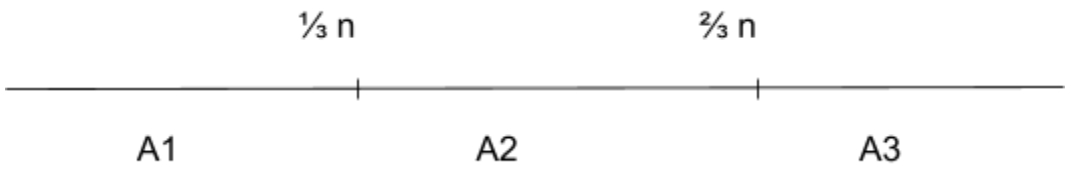
CS 515
Tenghuan Li; 933638707; liten@oregonstate.edu
Liangjie Zheng 934056203; zhengli@oregonstate.edu
Xingjian Su; 933647032; sux@oregonstate.edu

HW1

Problem 1.

a. This algorithm actually sorts its input.
Because as the picture shows, when we take the ceiling (⅔ n), it will make the data A[0~n-1] be 3 parts, A1, A2 and A3. For this recursive algorithm (A[0...m-1]), the first time is to swap for the A1 and A2, when it finishes the biggest data all in A2. The second time(a[n-m,...n-1]), it is swap the A2 and A3, then the biggest data all in A3, but we can not say A1's data is smaller than A2's data. So in the third recursive(A[0...m-1]) we do the comparisons A1 and A2, when all is finished, A1 < A2 < A3. So this algorithm is actually sorting its input. Because we take the ceiling (⅔ n), making m>=2/3n, that is, part A2>= 1/3n, it can be guaranteed that the number of A1 can be moved to A3, and vice versa.



b. When we replaced the ceiling (⅔ n) with floor (⅔ n), this algorithm is not correct.
For example, when n = 4, so m= floor (⅔ n)=2.
So in this STOOGESORT(A[0...3]):

```
If n=2 and A[0]>A[1]:
        Swap A[0]and A[1]
Else if n>2
        m=2;
        STOOGESORT(A[0...1]);
        STOOGESORT(A[2...3]);
        STOOGESORT(A[0...1]);
```

If inputs data is {10,9,8,7}.
First time, swap A[0] and A[1]. So result is {9, 10, 8, 7}
Second time, Swap A[2] and A[3]. So result is {9, 10, 7, 8}
Third time, because 9< 10 so do not swap, and Final result is {9, 10, 7, 8}. It is not correct when m = floor(⅔ n).

c. Because we need to get the number of comparisons executed by STOOGESORT.
So as the code
```
If n=2 and A[0]>A[1]:
    Swap A[0]and A[1].....1
Else if n>2
    m=2;
    STOOGESORT(A[0...m-1]);.....T(m)
    STOOGESORT(A[n-m...n-1]);.....T(m)
    STOOGESORT(A[0…m-1]);.....T(m)
```
So T(n) = 3T(m)+c
= 3T(ceiling(⅔*n))+1

d. Because of T(n) = 3T(⅔*n)+1.
So according to master's theorem, T(n) = a T(n/b) + f(n)

a = 3, b = 3/2 and f(n) = 1, calculate $n^{\log_b a}$ =1,

$$\log_{1.5} 3 - \varepsilon = 0, \varepsilon = 1.5 > 0$$

So the Time complexity is θ($n^{\log_{1.5} 3}$)

e. Because as the proof of the above problem, we know that it can be correct to order the A[n]. So now we guess in the n array data, we need to order the worst situation.
For example: when n =4, {10,9,8,7} m =3, when we in first recursion(A[0...m-1]), and change the 10 and 9, it will like {9,10,8,7} so 9 it must be before the 10, it same as all data area. The worst case is that when n=4, any two numbers need to be exchanged. Then the most swaps number it be $\binom{4}{2}$. When n=n, so the most case it will be $\binom{n}{2}$.

Problem 2:

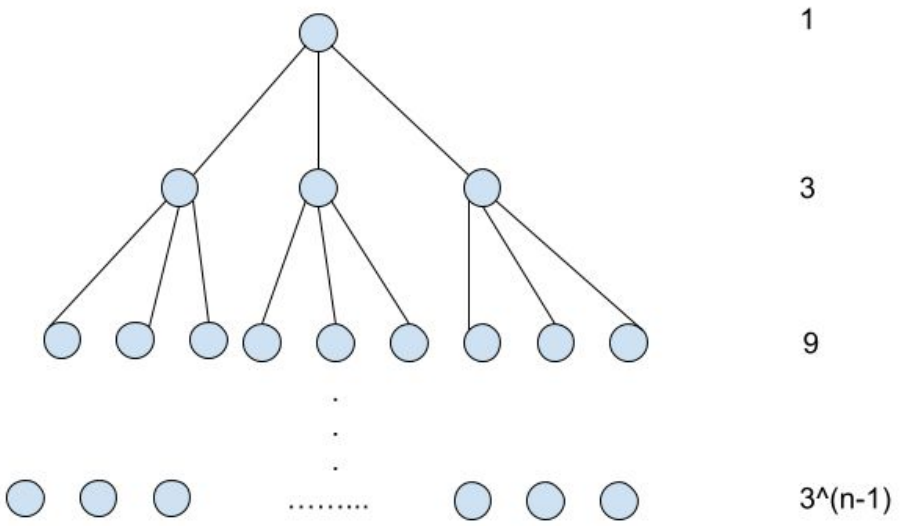
For this problem, we can't move between peg1 and peg2, every move must involve peg0.
I counted on the draft paper to get the result:

When n=1, the result is 1.
When n=2, the result is 4.
When n=3, the result is 13.
When n=4, the result is 40.

T(n) = T(n-1) + 1 + 2T(n-1).

Explain: When n=3 (a b c), for T(n-1), it means that move a and b to the mid need move T(n-1) times, and then move C for peg0 to peg2 it just 1 times. Then we need to move a, b from peg1 to peg0 then to peg2, so need to move 2T(n-1) times.

So, as the picture shows.



Sn = $(1- 3^n)/(1-3) = \frac{(3^n-1)}{2}$

Problem 3:

Record all delegates as array S(0,n-1), for each S[i] = (name, count 1)

Because in S, the number of people in party A is more than half; when we divide S into two equal arrays, S_left and S_right, A must be the majority in one (at least one) array. In the same way, we can treat S_left as S and divide equally, just like merge sort.

First we have a function max(left, right)
if count left >= count right, return left;
else return right

For every handshake,
If the two smile, the counts of the two are added, and either one is shifted to a higher level of handshake, and this person inherits the added count.
If the two are angry, the two counts are subtracted to take the absolute value; the larger one is shifted to a higher level of handshake, and this person inherits the calculated count.
for base case n = 2, there is a function handshake(left, right):

if smile, return (left_name, count left+right)
else, return (max(left, right), count |left-right|)

For the array S(n), divide the array into two equal parts and call the handshake function recursively, like merge sort. In the end, the person X who returns must belong to the majority party A. In the last step, we only need to make this person shake hands with everyone else to get the total number of majority party A.

The total number of people is n
T(1) = 0
T(n1) = 2T((n1)/2) + 2n1 = O(n1logn1) // Find someone who belongs to majority party A
T(n2) = n*O(1) = O(n) //Last step
T(n) = T(n1) + T(n2) = O(nlogn)

Problem 4.

For a given interval right endpoint i, take the interval maximum sum of the first i term as F[i]; obviously for any F[i], if F[i-1] is positive, then F[i]=F [i-1] + A[i]; If F[i] is negative, then F[i] = A[i].

base case: F[0] = A[0]

Obviously, the maximum value in the array F[n-1] is equal to $\max_{1 \le i \le j \le n} \sum_{k=i}^{j} A[k]$
So we can get the following algorithm:

```
max_SubArray(A[n])
{
F[0] = A[0];
max_value = F[0];
for (i = 1; i < n; i ++)
{
        F[i] = max(F[i-1]+A[i], A[i]);
        if (F[i] > max_value) max_value = F[i];
}
return max_value;
}
```

T(n) = n*O(1) = O(n)