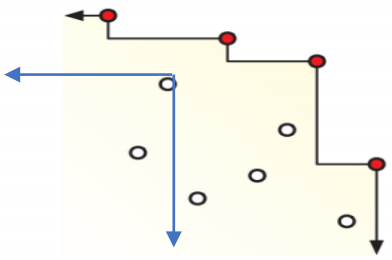
Practice Assignment 2

Daniel Schroeder

Schrodan@oregonstate.edu

1. Finding the maximums of a set of points P. I understood the idea of what I had to do for this algorithm but had difficulty finding out how to implement a recursive way of doing so. The first time I looked at the problem, I saw that you could do something like:



When you have a specific point, you can remove all points to that have X less than and Y less than, by the definition of maximums. Although I kept getting $O(n^2)$ algorithms because I could not definitely split the problem in half every recursive loop through. I turned to some research and found the Kirkpatrick-Seidel algorithm, which performs a similar technique, but in turn find the median, max, reduces, then splits the algorithms into two parts.
The Kirkpatrick-Seidel algorithm has six linear processes, followed by a recursive call on the upper and lower half of the problem set, giving us log(n) recursive calls at the $i^{th}$ call effectively performing in O(n log(n) ) timing. The algorithm is as follows:

```
findMaximums(P, Maximums):
        median = findMedian()  //finds the median x value'
        P_less = getAllPointsLess(median)        //a list all point with x <= median
        P_more = getAllPointsMore(median)        //a list all point with x > median

        Max = findMax(P_more)                    // get the point with the max y-value in P_more
        Maximums.add(Max)
        P_more.pop(Max)                  //remove max from P_more

        deleteDominating(P_more, Max)    //remove all the points like the diagram above
        deleteDominating(P_less, Max)            //that are x <= Max and y <= Max


        Maximums.add( findMaximums(P_more, Maximums) )  //recurse on P_more
        Maximums.add( findMaximums(P_less, Maximums) )          //recurse on P_less
```

Finding the median, finding all points less and greater, finding max, and removing dominated points are all linear time processes. By cutting the problem in half each time we can get a run time analysis of:

$$T(n) = 2 * T\left(\frac{n}{2}\right) + O(n) + O(n) + O(n) + O(n) + O(n) + O(n)$$

$$= 2 * T\left(\frac{n}{2}\right) + 5\,O(n)$$

$$= O(n \lg(n))$$

2. With a bitonic array that is first increasing and then decreasing after some index i… in order to find some number k, we can do three binary searches to receive a O(log (n)) overall runtime.

    2.1. The first binary search can be a modified binary search to find the max element. with a check for increasing and decreasing tendencies in the arrays that can be used to find the max number in the array X.
    2.2. Once we know the index of the Max number, we can binary search the array that is increasing X[0..max] for some number k
    2.3. And binary search the decreasing array X[max+1..n] for some number k

After three binary searches, we can effectively do three different O(log(n)) procedures to this bitonic array to search it for a value k in 3 * O(log(n)) or O(log(n))

3. This problem can be solved using dynamic programming. An oscillating array can have many subsequences that may or may not include every consecutive even/odd integer. For example:

The array: [1, 3, 4, 7, 8, 5, 2] has a subsequence [1,4,2] or [1,3,8,5], so there is a lot of room for variability with this problem.

To begin, we can find the LIS or longest increasing arrays. Using an algorithm like the one we learned in class for longest increasing subsequence or one I found online:

```
def _lis(arr , n ):

    # to allow the access of global variable
    global maximum

    # Base Case
    if n == 1 :
        return 1

    # maxEndingHere is the length of LIS ending with arr[n-1]
    maxEndingHere = 1

    """Recursively get all LIS ending with arr[0], arr[1]..arr[n-2]
       IF arr[n-1] is maller than arr[n-1], and max ending with
       arr[n-1] needs to be updated, then update it"""
    for i in xrange(1, n):
        res = _lis(arr , i)
        if arr[i-1] < arr[n-1] and res+1 > maxEndingHere:
            maxEndingHere = res +1

    # Compare maxEndingHere with overall maximum. And
    # update the overall maximum if needed
    maximum = max(maximum , maxEndingHere)

    return maxEndingHere
```

We would have to modify the algorithm to look at every other index so it only checked even indices.
We can then modify the algorithm to look in reverse order for decreasing subsequences of odd indexes.
Ideally after both algorithms run, we have the longest increasing subsequence, and the longest increasing subsequence, that we could then merge together based on their original indexes in A and get the largest oscillating subsequence.