General Algorithmic Technique: Sweeps

General Idea
For some problems, it makes sense to sort the data in some fashion, and then walk through the data with one or more pointers, moving either left to right or right to left. In a list of n items, if we do a constant amount of work for each position of one of the pointers and each pointer never moves backwards, we will do all of our work in O(n) time.

It turns out that for a wide range of problems, this general technique of sorting the data and then sweeping through it in some fashion can yield very efficient algorithms. In this lecture we'll look at six examples of the use of a sweep. Pay attention to the commonalities between the problems as well as the flexibility of the technique.

Since this technique is used in such different ways for specific problems, no "generic" code will be presented for it.

Problem #1: Sorted List Matching Problem
The problem is as follows: given two sorted lists of items, output a list with all items that are common to both lists. Here is a specific problem instance with two lists of names:

List #1	List #2
Adams	Boston
Bell	Davis
Davis	Duncan
Harding	Francis
Jenkins	Gamble
Lincoln	Harding
Simpson	Mason
Zoeller	Simpson

The most nieve way to solve this problem is to loop through each name in list one and do a linear search through list two, looking for the name. Unfortunately, if the lengths of the lists are *n* and *m*, respectively, then the run-time of this algorithm is *O(nm).*

An improvement over this idea would be to run a binary search in our search for an individual item in list two. This would improve our run-time to *O(nlogm).*

However, if we take our very first nieve algorithm, we might make one simple observation: as we are linearly searching through names in the second list, there is no need to start at the beginning of the list - we could just start searching where we left off, knowing that the previous names came too early in the alphabet. For example, if we're searching for Jenkins in list two, we can see that when we reach Mason in list two, we've gone to far, and we can conclude that Jenkins is not on the list. Now, our next search is for Lincoln. We can simply start search at Mason, since all of the previous items on list two preceded Jenkins!

Thus, we get the idea for our algorithm. Start two pointers at the beginning of both lists.

List #1	List #2
Adams *	Boston**
Bell	Davis
Davis	Duncan
Harding	Francis
Jenkins	Gamble
Lincoln	Harding
Simpson	Mason
Zoeller	Simpson

In code these would just be two integer variables storing indexes into the two arrays, respectively. At any point in time, just compare what's stored in the two respective indexes of the two arrays. If they are equal, output the value and increment BOTH pointers. If they are not equal, increment the pointer belonging to the item that comes first according to the method of comparison. In this example, we'd move the * pointer to Bell and again to Davis before moving the ** pointer to Davis, when we'd output Davis. Next we'd increment the ** pointer 4 times before we discovered that Harding was a match as well, outputting that name. After both pointers increment to Jenkins and Mason, respectively, the * pointer will move twice to Simpson, at which point the ** pointer will move to the Simpson on the second list. Once Simpson is outputted, the algorithm terminates, since one of the two pointers has finished sweeping through its list.

Problem #2: Merge Algorithm
This problem is nearly identical to the previous one, but is part of one of the most historic algorithms in Computer Science, which is why it's included in this lecture. The input to the problem is the same: 2 sorted lists. The output is a single list sorted containing all the items from both lists. As you might imagine, the only difference in the algorithm is that when one item in a list comes before an item in the other list, the earlier item gets outputted. Technically, for the merge problem, if two items are equal, we keep both copies of them in the output array. Once we have a solution to the merge algorithm that runs in $O(n+m)$ time, for two lists of size n and m, respectively, then we can write a recursive sort (named Merge Sort) as follows:

```
public static void mergeSort(int[] array, int low, int high) {
    if (low < high) {
        int mid = (low+high)/2;
        mergeSort(array, low, mid);
        mergeSort(array, mid+1, high);
        merge(array, low, mid, high);
    }
}
```

This code will sort the section `array[low...high]`, inclusive. The merge method will merge the subsections `array[low...mid]` and `array[mid+1...high]` storing the answer back into array.

Problem #3: Bubble Wrap (SI@UCF 2015 Contest #1)
The problem is as follows:

Given a set of positive integers, determine whether or not either one or two of them add up to a specified target value. The bounds of this problem were relatively small, allowing for a brute force solution for each query. However, a more efficient solution does exist, using a sweep.

First, add 0, to represent not taking a second item, and then sort the array. Next, start with one pointer at the beginning of the array and another at the end. Consider the following example with a target of 25:

index	0	1	2	3	4	5	6	7	8	9
value	0	2	3	7	9	11	19	21	27	34
	low									high

Just by adding array[low] and array[high], we get a value too high. This means if there is to be a solution, the high pointer must come down. Notice that it'll come down a second time to index 7. Since 0 + 21 < 25, this means the low pointer must come up to index 1 then index 2, ending at index 3. At this point, the high pointer decrements twice to equal 5 (since 7 + 21 > 25 and 7 + 19 > 25). Finally, when the low pointer crosses over the high pointer, we know that no solution can exist. Basically, we don't move off of a number with either pointer unless we have proof that no solution exists with that number. Essentially, if a solution were to exist with 0, the number 25 (the number itself) would have to be in the array. Since the array is sorted, we know to continue searching for lower numbers until we hit 21. At this point, we know if a solution with 21 existed, the array would have to have a 4 in it. Thus, at this point, the low pointer must slide up twice. Once a pointer has moved off each index without finding its match, we can ascertain that no solution exists.

Here is the pertinent section of the sweep from one of the posted solutions, which uses an array of size n+1:

```
int low = 0, high = n;
while (low < high) {

     int cur = vals[low]+vals[high];

     if (cur > target) high--;
     else if (cur < target) low++;

     else {
          res = true;
          break;
     }
}
```

Problem #4: Goldilocks and the N Cows (USACO Bronze Nov 2013)
The problem is as follows:

You are given up to 20000 cows. Each cow as a preferred temperature range [a_i, b_i], for the ith cow. If the temperature is below an individual cow's preferred range, she produces X units of milk. If the temperature is within an individual cow's preferred range, she produces Y units of milk. Finally, if the temperature is above an individual cow's preferred range, she produces Z units of milk. It's guaranteed that Y > X and Y > Z. The bounds are such that integers are valid to use for all calculations. The goal of the problem is to find the maximum amount of milk the cows can produce, assuming that we're only allowed to set one temperature for all of the cows. (They are all in the same barn, controlled by a single thermostat, apparently.)

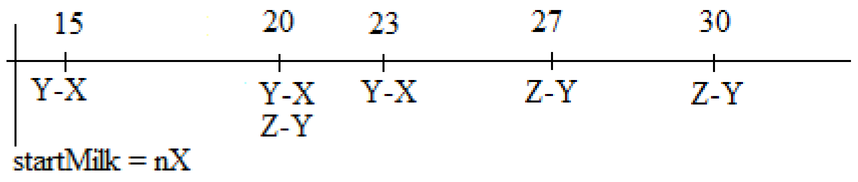
The bounds are also such that trying each possible temperature and calculating the corresponding milk production each time would be too slow.

Now, we can consider the idea of sorting ALL of the temperatures given in the ranges. Imagine, for example, for three cows the comfortable temperature ranges are [20, 30], [23, 27] and [15, 19]. Our sorted list would be 15, 19, 20, 23, 27, 30. Imagine starting the thermostat below the first temperature and gradually increasing it until it's hotter than the last temperature. If there are n cows, at first the milk production would be nX and at the very end it would be nZ. In fact, the milk production would only change any time we hit one of these critical boundaries. Changing the thermostat from 17 to 18 does nothing for us because any cow comfortable at 17 degrees is also comfortable at 18 degrees, and vice versa.

But, a full recalculation of the milk production would yield is a run time of *$O(n^2)$* still, even if we just try each "important" temperature. The trick is realizing that when I change my theromstat from 14 to 15, not a whole lot changes in terms of milk production. Namely, one cow transitions from producing X units to producing Y units. Thus, if we have a variable storing the current milk production, all we have to do is add (Y - X) to it to indicate the new total milk production when our thermostat hits 15 degrees. When we hit 20, we lost some production from the cow who's range is [15, 19] and gain some production from the cow who's range is [20, 30]. Thus, our "critical points" are every cow's low threshold and one plus every cow's high threshold, since they cross over to producing Z units of milk when the temperature ***exceeds*** the given high threshold. At each critical point, we have some number of cows moving from too cold to just right and some number of cows moving from just right to too hot. In this example, when we hit 20, we add (Z - Y) for the cow that becomes too hot and (Y - X) for the cow who goes from too cold to just right. We can just count how many cows make these two moves at each critical point to adjust our temperature like so:

```
temp = temp + numMoveToJustRight*(Y-X) + numMoveTooHot*(Z-Y);
```

Graphically, we have something like this:



Thus, as we sweep through the possible temperatures to which to set the thermostat, we only do an O(1) update to calculate the total milk production for each new temperature, since we can just use the frequencies of the two types of cow changes. Thus, the dominating portion of our code is sorting the up to 40000 values.

Note: The full description of the problem as well as the data for it can be found on the USACO website for the November 2013 contest. A sample solution is included in the corresponding section of the camp webpage, alongside these nodes in the file milktemp.java.

Problem #5: Ski Course Design (USACO Bronze Jan 2014)
The problem is as follows:

You are given up to 1000 hills, each of with a height in between 0 and 100, inclusive. You are allowed to increase the height of a short hill or decrease the height of a tall hill in the hopes of making the difference in height between the shortest and tallest hill 17 or less. The cost of either increasing or decreasing the height of an individual hill x units is x^2. The goal is to satisfy the goal at a minimum cost.

We first notice that there are only 101 possible different heights, but up to 1000 hills and that whatever we do to one hill of a particular height, we must do to all of the other hills of that same height. Thus, we only care about how many hills have some fixed height. Thus, we should store the input data in a frequency array, where freq[i] stores how many hills are of height i, originally.

Once we realize this efficiency in storage (note that with this storage, we could take an input of many more hills, say up to 1000000), we can just do a basic brute force solution that tries each interval [0, 17], [1, 18], [2, 19], ..., [83, 100]. Checking each of these intervals is essentially a sweep. Due to the complication of the frequency array and the square scoring function however, it's much easier to do a full recomputation for each interval, rather than trying an O(1) calculation to account for the shift over the desired range.

Problem #6: Lazy Cow (USACO Bronze Mar 2014)

The problem is as follows:

You are given upto 100000 patches of grass located on a number line from x = 0 to x = 1000000. Each patch of grass has a value. Your goal is to position yourself on the number line such that the amount of grass within k units of you is maximal.

Once again, a sweep is idea. Sort the data by x coordinates and set up a low and high pointer, both to start at the left end of the array. If the range of x values covered from low to high is less than 2k, our total range (k to the left, k to the right), then increment the high pointer. Otherwise, increment the low pointer. Consider the following example with k = 5:

index	0	1	2	3	4	5	6	7	8
x-coor	3	12	13	17	22	41	47	62	65
value	53	22	21	5	22	31	10	47	53

Both low and high start at index 0. high will increment through index 2, where we get a sum of 96. When we increment high again, we see that the difference in x values is too high, so we don't count this interval. On the next step we increment low, adjusting our sum (just by subtracting 53) to 48. We increment high again, adjusting our running sum to 70. Once we increment high to index 5, index low will repeatedly increment until it equals 5 and we have a running tally of 31. The two indexes will meet again at index 7 and then high will increment to index 8. Here at our last iteration, we get a sum of 100 which beats our previous high of 96. Notice that each time we move an index it takes O(1) time since we are simply adding or subtracting one value from our running tally. Thus, our overall run time is O(*nlgn*), for sorting the *n* patches of grass by x.