Sorting

Sorting is ordering a list of objects. Here are some sorting algorithms

Bubble sort
Insertion sort
Selection sort
Mergesort

Question: What is the lower bound for all sorting algorithms?

Algorithms that use comparison to determine the relative order of objects is called comparison sorts. Bucket sort and radix sort are based on a different idea, they are non-comparison sort algorithms.

Bubble Sort

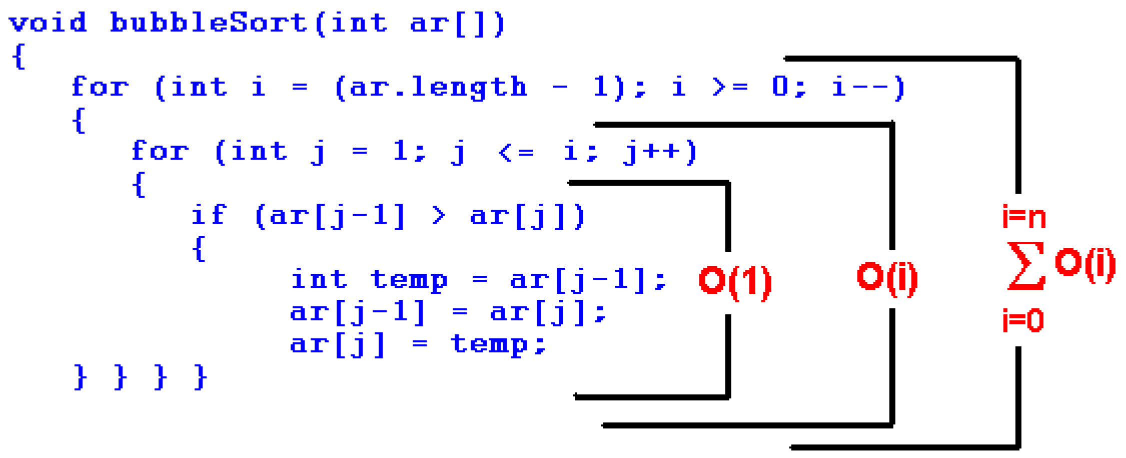
The algorithm works by comparing each item in the list with the item next to it, and swapping them if required. In other words, the largest element has bubbled to the top of the array. The algorithm repeats this process until it makes a pass all the way through the list without swapping any items.

```
void bubbleSort(int ar[]){
      for (int i = (ar.length -1); i >= 0; i--) {
            for (int j = 1; j = i; j++) {
                  if (ar[j-1] > ar[j]) {
                        int temp = ar[j-1];
                        ar[j-1] = ar[j];
                        ar[j] = temp; }
            }
      }
}
```

Example. Here is one step of the algorithm. The largest element - 7 - is bubbled to the top:

```
7, 5, 2, 4, 3, 9
5, 7, 2, 4, 3, 9
5, 2, 7, 4, 3, 9
5, 2, 4, 7, 3, 9
5, 2, 4, 3, 7, 9
5, 2, 4, 3, 7, 9
```

The worst case complexity is O(n2). See explanation below



```
void bubbleSort(int ar[])
{
   for (int i = (ar.length - 1); i >= 0; i--)
   {
      for (int j = 1; j <= i; j++)
      {
         if (ar[j-1] > ar[j])
         {
              int temp = ar[j-1];
              ar[j-1] = ar[j];
              ar[j] = temp;
   } } } }
```

$$\sum_{i=0}^{i=n} O(i) = 1 + 2 + 3 + \ldots + (n-1) = O(n^2)$$

Selection Sort

The algorithm works by selecting the smallest unsorted item and then swapping it with the item in the next position to be filled.

The selection sort works as follows: you look through the entire array for the smallest element, once you find it you swap it (the smallest element) with the first element of the array. Then you look for the smallest element in the remaining array (an array without the first element) and swap it with the second element. Then you look for the smallest element in the remaining array (an array without first and second elements) and swap it with the third element, and so on.

```
void selectionSort(int[] ar){
      for (int i = 0; i < ar.length-1; i++) {
            int min = i;
            for (int j = i+1; j < ar.length; j++)
                  if (ar[j] < ar[min]) min = j;

            int temp = ar[i];
            ar[i] = ar[min];
            ar[min] = temp;
      }
}
```

Example.

```
29, 64, 73, 34, 20,
20, 64, 73, 34, 29,
20, 29, 73, 34, 64
20, 29, 34, 73, 64
20, 29, 34, 64, 73
```

The worst case complexity is O(n2).

Insertion Sort

To sort unordered list of elements, we remove its entries one at a time and then insert each of them into a sorted part (initially empty):

```
void insertionSort(int[] ar) {
      for (int i=1; i < ar.length; i++){
            int index = ar[i]; int j = i;
            while (j > 0 && ar[j-1] > index) {
                  ar[j] = ar[j-1];
                  j--;
            }
            ar[j] = index;
      }
}
```

Example. We take an element from unsorted part and compare it with elements in sorted part, moving from right to left.

```
29, 20, 73, 34, 64
29, 20, 73, 34, 64
20, 29, 73, 34, 64
20, 29, 73, 34, 64
20, 29, 34, 73, 64
20, 29, 34, 64, 73
```

Let us compute the worst-time complexity of the insertion sort. In sorting the most expensive part is a comparison of two elements. Surely that is a dominant factor in the running time. We will calculate the number of comparisons of an array of N elements:

we need 0 comparisons to insert the first element
we need 1 comparison to insert the second element
we need 2 comparisons to insert the third element
...
we need (N-1) comparisons (at most) to insert the last element

This takes us to

```
1 + 2 + 3 + ... + (N-1) = O(n2)
```

The worst case complexity is O(n2).

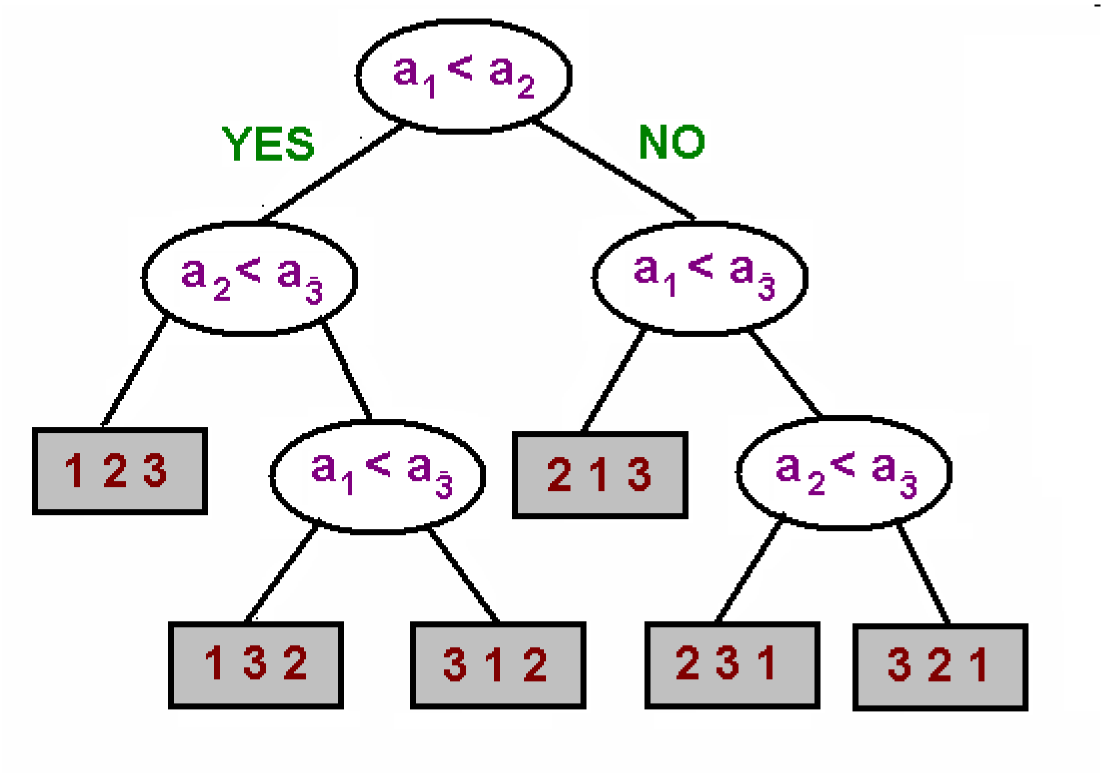
Question: What is the best-case complexity?

The advantage of insertion sort compare it to the previous two sorting algorithm is that insertion sort runs in linear time on nearly sorted data.

Lower bound

What is the lower bound (the least running time in the worst-case) for all sorting comparison algorithms? Suppose we have N elements. How many different arrangements can you make? There are N possible choices for the first element, (N-1) possible choices for the second element, .. and so on. Multiplying them, we get N! (N factorial.)

Next, we observe that each comparison cut down the number of all possible comparisons by a factor 2. Any comparison sorting algorithm can always be put in the form of the decision tree. And conversely, a tree like this can be used as a sorting algorithm



Observe, that the worst case number of comparisons made by an algorithm is just the longest path in the tree. At each leaf in the tree, no more comparisons to be made. Therefore, the number of leaves cannot be more than 2x, where x is the maximum number of comparisons (or the longest path in the tree). On the other hand, as we counted in the previous paragraph, the number of all possible permutations is n! Combining these two facts, gives us the following equality:

```
2x = N!
```

where x is the number of comparisons.

By applying logarithm, this implies

```
                x = log N!
```

Using the Stirling formula for N! we finally arrive at

```
                x = N log N
                or
                x = O(N Log N)
```

Question: How many comparisons do we need to sort five items?

Mergesort

Mergesort involves the following steps:

- Divide the array into two subarrays
- Sort each subarray (Conquer)
- Merge them into one (in a smart way!)

Example. Consider the following array of numbers

```
          27 10 12 25 34 16 15 31
```

divide it into two parts

```
          27 10 12 25           34 16 15 31
```

divide each part into two parts

```
     27 10       12 25           34 16           15 31
```

sort each part independently

```
     10 27       12 25           16 34      15 31
```

merge parts into one (by comparing, one by one, the paired elements from the two arrays)

```
     10 12 25 27           15 16 31 34
```

Merge the last two subarrays into one array

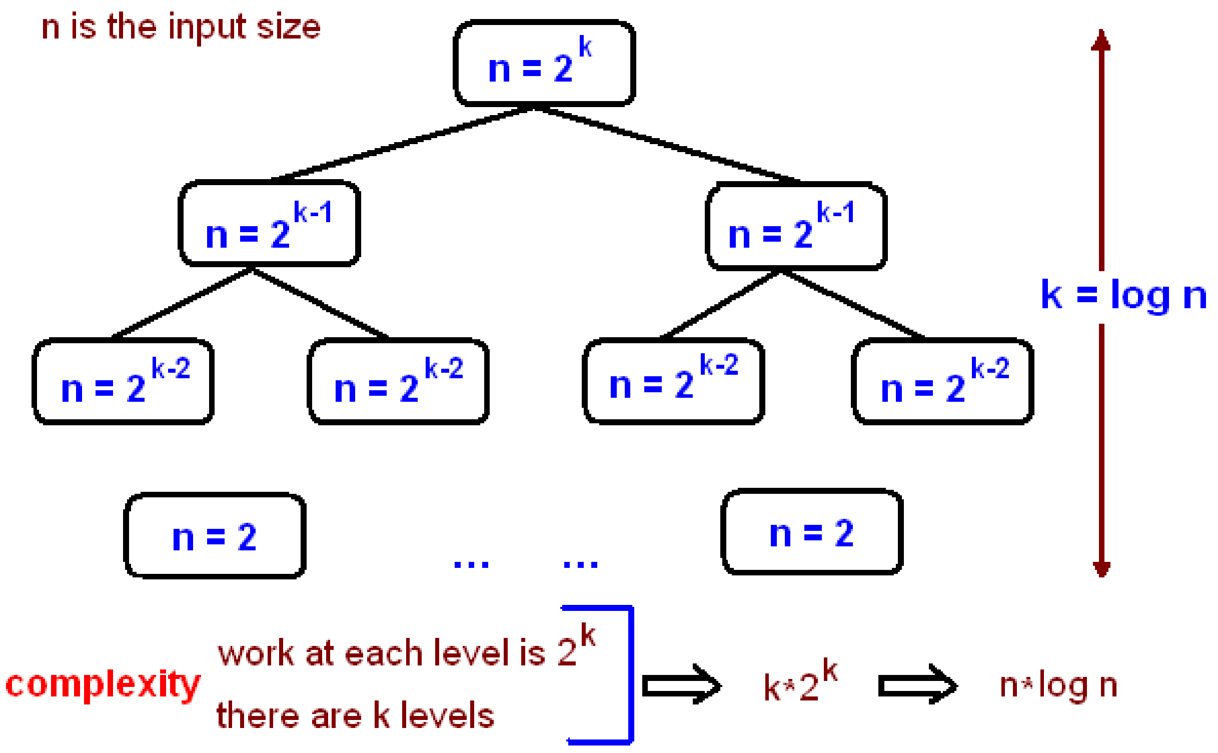
```
          10 12 15 16 25 27 31 34
```

Complexity Analysis

Suppose T(n) is the number of comparisons needed to sort an array of n elements by the MergeSort algorithm. By splitting an array in two parts we reduced a problem to sorting two parts but smaller sizes, namely n/2. Each part can be sort in T(n/2). Finally, on the last step we perform n-1 comparisons to merge these two parts in one. All together, we have the following equation

```
          T(n) = 2*T(n/2) + n -1
```

The solution to this equation is beyond the scope of this course. However I will give you a reasoning using a binary tree. We visualize the mergesort dividing process as a tree


n is the input size
$n = 2^k$
$n = 2^{k-1}$
$n = 2^{k-1}$
$n = 2^{k-2}$
$n = 2^{k-2}$
$n = 2^{k-2}$
$n = 2^{k-2}$
n = 2
... ...
n = 2
k = log n
complexity
work at each level is 2^k
there are k levels
$k*2^k$
n*log n