1. Bubble Sort
2. Selection Sort
3. Insertion Sort
4. Merge Sort
5. Quick Sort

A comprehensive list and animations of various sorts can be found at
http://www.cs.ubc.ca/spider/harrison/Java/sorting-demo.html

1 Towers of Hanoi

1.1 The Legend

In an ancient city in India, so the legend goes, monks in a temple have to move a pile of 64 sacred disks from one location to another. The disks are fragile; only one can be carried at a time. A disk may not be placed on top of a smaller, less valuable disk. And, there is only one other location in the temple (besides the original and destination locations) sacred enough that a pile of disks can be placed there. So, the monks start moving disks back and forth, between the original pile, the pile at the new location, and the intermediate location, always keeping the piles in order (largest on the bottom, smallest on the top). The legend is that, before the monks make the final move to complete the new pile in the new location, the temple will turn to dust and the world will end. Is there any truth to this legend?
(Since it would take at least $2^{64} - 1$ moves to complete the task, we're safe for now. Assuming one move per second, and no wrong moves, it would take almost 585,000,000,000 years to complete.)

1.2 The Game

There's a game based on this legend. You have a small collection of disks and three piles into which you can put them (in the physical version of this game, you have three posts onto which you can put the disks, which have holes in the centre). The disks all start on the leftmost pile, and you want to move them to the rightmost pile, never putting a disk on top of a smaller one. The middle pile for intermediate storage. You can only move one disk at a time.
Here is what the initial configuration looks like Fig. 1
For an animation of this game you can visit:
http://cs.colgate.edu/ parks/cosc/101/examples/towers/towers.html–animation

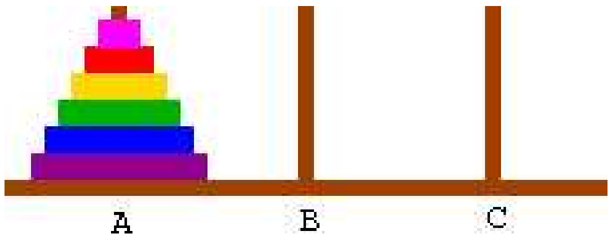

Figure 1: Tower of Hanoi–initial configuration

1.3 Solutions Overview

http://obelix.ee.duth.gr/ apostolo/TowersOfHanoi/
Recursive Solution:
Why a recursive solution? The ultimate goal is to move the bottom (largest) disk to the target pole. But we cannot do this until the one above it (second largest) is moved. Applying that logic again and again, we would get to moving the top disk (the again and again part should suggest to you all that recursion is required).
The invariant is as follows: on recursive call k, all disk smaller than k should be on the spare pile, all disks larger than k should be on the target pile. We move disk k to the destination pile.

1. **In order to move all of the disks to pole B we must certainly move the biggest disk there, and pole B must be empty just prior to moving the biggest disk to it**
2. **Now looking at the initial configuration, we can't move the biggest disk anywhere until all other disks are first removed. Furthermore, the other disks had better not be moved to pole B since then we would not be able to move the biggest disk there. Therefore we should first move all other disks to pole C**
3. **Then we can complete the key step of moving the biggest disk from pole A to pole B and go on to solve the problem**

```
int moves=0;
void hanoi(int   height,
           char  fromPole,
           char  toPole,
           char  withPole){
       if (height >= 1)
       {
          hanoi(height-1, fromPole, withPole, toPole); //move the smaller disks to the spa
          moves++; //move this disk to the destination pole
          hanoi(height-1, withPole, toPole, fromPole); //move the smaller disks to the des
       }
```

```
    }
```

2 How long does this take

This section is optional: it wouldn't hurt you to read it, especially if you plan to stay in computer science.
To estimate the complexity of recursive functions, we use *recurrence.* A recurrence is a well-defined mathematical function written in terms of itself; it's a mathematical function defined recursively. In the case of Towers of Hanoi, our functions takes the following number of steps: $t(n) = t(n-1) + k + t(n-1)$: it's pretty straightforward–two recursive calls with parameter n-1 (since we reduced the data size, ie. the height by one) . So we have $t(n) = t(n-1) + k$. Additionally, we have a starting point: t(0)=0, since for height 0 we do nothing, i.e. just return.
Moving from here is a more complicated issue. To get an O bound on this function, we need to *solve* the recurrence. This is done inductively (i.e. by induction): start with the values given and substitute the values until you see a pattern in terms on n. For simplicity, let's replace k with 1:

```
t(0)=0
t(1)=2*t(0)+1=1
t(2)=2*t(1)+1=2+1=3
t(3)=2*t(2)+1=6+1=7
t(4)=2*t(3)+1=14+1=15
```

Looking carefully, you will notice that the general pattern is of the form $t(n) = 2^n - 1$ (for the curious, turns out this sequence is called the Mersenne numbers–they have a whole bunch of interesting properties). In other words, you just wrote your first exponential function

3 Merge Sort

This algorithm is based on a technique called *divide and conquer*
Definition:An algorithmic technique. To solve a problem on an instance of size n, a solution is found either directly because solving that instance is easy (typically, because the instance is small) or the instance is divided into two or more smaller instances. Each of these smaller instances is recursively solved, and the solutions are combined to produce a solution for the original instance

3.1 Idea

Split the original array(sequence) in half, sort each half separately. Merge the two halves while imposing a total order in the new list. The *sort* word above is a hint to the fact that this is,again, a recursive algorithm. I will use ints for simplicity, please not that this works for any type, as long as it implements Comparable, so you can compare two objects using $< or >$.
Note: The website given at the beginning of the handout has an “in-place” version of this algorithm, which does not waste space, but takes longer to execute

```
void MergeSort(int[] toSort){
    //base case
    if (toSort.length==1)return;
    else{
      int mid=toSort.length/2;
      int[] left=new int[mid];
      int[] right=new int[toSort.length-mid];
      int i=0;
      //copy lower 1/2 into left
      for (;i<mid;i++)
        left[i]=toSort[i];
      //copy upper 1/2 into left
      for (int j=0;j<toSort.length-mid;j++)
        right[j]=toSort[i++];
      MergeSort(left); //left recursion
      MergeSort(right);//right recursion
      //overwrite toSort with the sorted data from left and right
      merge(toSort,left,right);
    }
}

void merge(int[] toSort,int[] left,int[] right){
  int curr=0;
  int left_pos=0;
  int right_pos=0;
  while (left_pos<left.length && right_pos<right.length)
     if (left[left_pos]<right[right_pos])
          toSort[curr++]=left[left_pos++];
      else
          toSort[curr++]=right[right_pos++];

      if(left_pos==left.length)
         while (right_pos<right.length)
    toSort[curr++]=right[right_pos++];
      if (right_pos==right.length)
```

```
          while(left_pos<left.length)
      toSort[curr++]=left[left_pos++];
      }

}
```

3.2 How long does this take

If you read section 2, you should be able to solve this "scientifically":$T(n) = T(n/2) + T(n/2) + n, if n > 1; T(1) = 0$

The indirect way goes like this:
At each invocation, the algorithm does a linear amount of work at each level: How many invocations do we have at each level?
Pick a level in the recursion tree, say k. At level k, there are 2^k **nodes.**
At each node, merge does work linear in the number of items in the input array for that node.The size of this array per node, in terms of n is $n/2^k$**.**
The total work ends up being $2^k * n/2^k = n$
So the total work at each level is O(n).
The tree has a depth of O(lg n) , so the overall complexity is O(nlgn)
How about space? Look at how much additional space are you allocating at each call? We copy the whole array out in two parts, so the total space is O(n).