26 How Many Times Must a White Dove Sail ...

From Here to Eternity

According to an old legend the priests in the Temple of Brahma got the task of measuring the time till the end of the world. When they complete the task described below, the world will end. This problem is commonly known as the *Towers of Hanoi* problem.

The priests were given 64 gold discs of descending size that were placed on the first of three towers made of diamond. Their task is to move all disks to the second tower, using the third tower as a helper - making sure never to put a larger disk on top of a smaller one and always move only one disk at a time.

Take a blank sheet of paper, fold it and cut in in half, then do the same with one of the halves, and repeat it two more times. You end up with four pieces of paper, each half the size of the previous one. These are your four towers. Put them on a table, imagine the locations of the other two towers, and experiment in performing the priests' task. Try to use as few moves as possible.

Can we apply the design recipe to solving this problem? We already know the data - three towers each has some disks on it, it could be empty. We also know that on each tower the disks are placed with smaller ones of top of the larger ones. Let us make examples. What would we do if we had only one disk to move? How about, if we were to move two disks? Now that we know how to move two disks, can we move three? Think about it!

Sure, just move the two to the *helper* tower, then move the bottom disk to the final destination, and then move two disks to the destination tower. Here is the description of the process — a program for the priests to follow:

```
// count the steps needed to move n disks from source tower
// to destination tower, using the helper tower
int countMoves(int n, Tower from, Tower to, Tower help){
  return
    // move n − 1 disks from the source to the helper tower
    countMoves(n − 1, from, helper, to)

    + 1   // move the bottom disk: from −−> to

    // move n − 1 disks from the helper tower to the destination
    countMoves(n − 1, helper, to, from);
}
```

We could print the instructions for how to move the disks as follows:

```
// print the steps needed to move n disks from source tower
// to destination tower, using the helper tower
void printMoves(int n, String from, String to, String help){
  // print moves for n − 1 disks from the source to the helper tower
  printMoves(n − 1, from, helper, to)

  System.out.println("Move disk from " + from + " to " + to);

  // print moves for n − 1 disks from the helper tower to the destination
  printMoves(n − 1, helper, to, from);
}
```

Let us count the moves. To move *n* disks requires all the moves to move $n - 1$ disks twice, plus one more move for the bottom disk. The table below shows what we discovered:

Towers	move n - 1 disks	+	bottom	+	move n - 1 disks	=	total
1 disk	0	+	1	+	0	=	1
2 disks	1	+	1	+	1	=	3
3 disks	3	+	1	+	3	=	7
4 disks	7	+	1	+	7	=	15
5 disk	15	+	1	+	15	=	31

The relationship between the number of disks and the number of moves is *moves* = *2ˆdisks* − 1. So, if we need one second to perform each move, how long is it till the world ends? About 584 billion years!!!

This is frightening. We just wrote a small program to print the moves, but if we invoke the method with 64 disks, it will print 584 billion times 31,536,00 (the number of seconds per year) lines. This is why understanding the time complexity of programs is so important. You need to be able to estimate from the structure of the program what is the expected growth in the time needed. The *Towers of Hanoi* problem is on the order of *O*(*2ˆn*), also called the *exponential growth*. No matter what we do, running such programs is not feasible for any inputs other larger than some very small threshold.

Sorting out Sorting

The section title is a name of a short movie from the 70's that illustrated the time complexity of different sorting algorithms. In general, we are interested in understanding complexity of all algorithms used in our programs. The reason for focusing on sorting algorithms is two-fold. The students are familiar with them, and they exhibit the kind of behavior we want to illustrate. In real world programming you will most often use sorting algorithms that are already included in the libraries with the programming language or an application you will work with. At the same time, you may be able to choose among the algorithms available, and our analysis will help you to understand how to make this choice.

Insertion sort

When performing an insertion sort, we insert each item into an already sorted list. At first the sorted list is empty, and with each subsequent insertion its size grows. For each insertion, the time needed to insert the item into the sorted list of size k requires one comparisons for the best case, $k-1$ comparisons for the worst case, and about $k/2$ comparisons on the average.

So, for the best case the number of operations is on the order of $O(n)$ — each of the n items inserted in one step. For the worst case, the number is $1+2+3+\ldots+(n-1) = n\ .\ (n-1)\ /\ 2$. The best case happens when the input list is already sorted. As the result, insertion sort is the preferred sort in situations where the input is almost sorted and we expect only a few items to be out of order.

For the average case, the number of comparisons is about 1/2 of this. However, the measure of complexity, *Big-O* ignores the scale factor for the comlexity function, as well as the polynomial terms of lower degree. So, here, both the worst case and the average case are on the order of $O(n\hat{}2)$.

If the list is represented as an *ArrayList*, then the insertion itself requires the move of the remainder of the list, and so the number of comparisons and move operations for insertion into a list of size k is always k. This does not change our previous results.

Selection sort

In selection sort, the *ArrayList* is divided into two parts, the lower, sorted part and the upper, unsorted part. In each iteration of the algorithm we first find the index for the smallest item in the *unsorted* part and swap it with the first element of the *unsorted* part. At this point, the boundary between the sorted and unsorted parts moves up by one. The table below illustrates this:

Sorted part	Unsorted part	Minimum and its index	steps
	0 1 2 3 4 5 6 7 D G B E C H A F	value A at index 6	7
0 A	1 2 3 4 5 6 7 G B E C H D F	value B at index 2	6
0 1 A B	2 3 4 5 6 7 G E C H D F	value C at index 4	5
0 1 2 A B C	3 4 5 6 7 E G H D F	value D at index 6	4
0 1 2 3 A B C D	4 5 6 7 G H E F	value E at index 6	3
0 1 2 3 4 A B C D E	5 6 7 H G F	value F at index 7	2
0 1 2 3 4 5 A B C D E F	6 7 G H	value G at index 6	1
0 1 2 3 4 5 6 A B C D E F G	7 H	value H at index 7	0

We see that regardless of how whether the data is sorted or not, the algorithm will always take $1 + 2 + 3 + \ldots + n - 1 = n \,.\, (n - 1) / 2$ steps and so it is on the order of $O(n\hat{}2)$ for the best case, average case, and the worst case. However, the number of moves is n, and so, if the move is an *expensive* operation, this algorithm may be a good choice.

Quicksort

The quicksort algorithm uses the *divide and conquer* strategy. During this discussion, we assume there are no duplicate values in the list. The following description of the algorithm will guide us in the analysis:

1. If the size of the list is 1, STOP.
2. Select one element of the list to be the *pivot*.
3. Produce the *lower-half* list of items with values less that the *pivot* and the *upper-half* list of items with values greater than the *pivot*.

4. Run *Quicksort* on the *lower-half*.

5. Run *Quicksort* on the *upper-half*.

6. Produce the list *sorted lower-half* + *pivot* + *sorted upper-half*.

If at each step the size of each *half* is indeed about 1/2 of the size of the list being divided, the algorithm requires the following number of operations:

```
+-----------------------------------------------+------------------+
| partitions                                    | total operations |
+-----------------------------------------------+------------------+
|                 31 items                      |  1 x 31     = 31 |
+-----------------------------------------------+------------------+
|      15 items          |       15 items       |  2 x 15     = 30 |
+-----------------------------------------------+------------------+
|  7 items     7 items   |  7 items     7 items |  4 x  7     = 28 |
+-----------------------------------------------+------------------+
|  3     3  |  3     3   |  3     3  |  3     3 |  8 x  3     = 24 |
+-----------------------------------------------+------------------+
| 1 1 | 1 1 | 1 1 | 1 1 | 1 1 | 1 1 | 1 1 | 1 1 | 16 x  1     = 16 |
+-----------------------------------------------+------------------+
```

We see that at each iteration there are between *n/2* and *n* operations, and that there are log(n) iterations needed to get the partition sizes down to 1. That means that for this ideal case the *quicksort* algorithm runs on the order of *O*(*n.log*(*n*)) time. This is the ideal situation and the best case.

What would be the worst case? If we select the pivot which has the minimum value of all items in the list, then the *left-half* will be empty and the *right-half* will have $n - 1$ items. If this happens at each iteration, we end up with *n-1* iterations, doing a total of $(n - 1) + (n - 2) + \ldots + 3 + 2 + 1$ comparisons to solve the problem. That means, that in the worst case (for example when the data is sorted and our pivot choice is the leftmost item), the *quicksort* runs in on the order of *O*(*n*ˆ2) time. Formal mathematical studies of this problem show that with a careful choice of the pivot, the probability of this bad behavior is negligible. A practical programmer needs to understand that the strategy for choosing the pivot for a *quicksort* algorithm has to be addressed and understood before using the algorithm in an application.

Binary Search Tree Sort

When given a sequence of *comparable* items, we can always insert them one-by-one into a binary search tree. The *inorder* traversal of this tree then produces the items in a sorted order.

The best case for the insertion part of the algorithm produces a tree with all levels except the last one filled. The height of such binary tree with n nodes is $\log(n)$. Therefore the insertion of all items into the tree runs on the order of $O(n.log(n))$. The three binary tree traversals we have seen (*pre-order*, *in-order*, and *post-order*) each take n steps, i.e. running on the order of $O(n)$. So, while the total number of operations is approximately $n.log(n) + n$, the whole algorithm still is on the order of $O(n.log(n))$, because $n.log(n)$ dominates the growth curve.

In the worst case, the insertion part of the algorithm produces a degenerate tree of height n. In this case the *k-th* item will be inserted as a child of the node at the level $k - 1$, requiring $k - 1$ comparisons. The total number of operations is the same as for the worst case insertion sort, i.e. it runs on the order of $O(n\hat{}2)$ time.

Summary

The table below shows the comparison of our results for the sorting algorithms:

algorithm \ time	worst case	average case	best case
insertion sort	O(n^2)	O(n^2)	O(n)
selection sort	O(n^2)	O(n^2)	O(n^2)
quicksort	O(n^2)	O(nlog(n))	O(nlog(n))
binary tree sort	O(n^2)	O(nlog(n))	O(nlog(n))
merge sort	O(nlog(n))	O(nlog(n))	O(nlog(n))

Exercise

Verify the estimated running times of the *merge sort* algorithm, shown in the table above.