CIS 575. Introduction to Algorithm Analysis
Remarks on Assignment #5, Spring 2014

In Java, one may write SlowSort as follows:

```
public class SlowSort {

public static final int Size = 100;

public static void slowSort(int[] A, int low, int high) {
int n = high - low + 1;

// Map [low high] to [1 n]
int offset = low - 1;

int ceiling = (int) Math.ceil((double) 2 * n / 3);
int floor = (int) Math.floor((double) n / 3);

if (n == 2) {
if (A[low] > A[high]) {
// switch
A[high] = A[high] + A[low];
A[low] = A[high] - A[low];
A[high] = A[high] - A[low];

}

} else if (n > 2) {
// Map [1 n] back to [low high]
slowSort(A, low, ceiling + offset);
slowSort(A, (floor + offset + 1), high);
slowSort(A, low, ceiling + offset);

}
}

public static void main(String[] args) {
//random input will be OK, since T(n) is unrelated to initial status of A
int A[] = new int[Size];
for (int i = 0; i < Size; i++) {
A[i] = (int) (Math.random() * 10000);
}

long startTime = System.currentTimeMillis();
slowSort(A, 0, Size - 1);
long timeCost = System.currentTimeMillis() - startTime;
System.out.println("time: " + timeCost);

// for (int i = 0; i < Size; i++) {
// System.out.println(A[i]);
// }
}
}
```

For another implementation, written in Standard ML and run on a MacBook (2.4GHz Intel Core 2 Duo), we used a watch to loosely measure running time, with the following results:

input length n	iterations	total time (secs)	time $T(n)$ (ms)	$\lg(T(n))$	$\lg(n)$
10	500,000	32	0.064	-4.0	3.3
20	50,000	28	0.56	-0.8	4.3
40	10,000	17	1.7	0.8	5.3
100	500	23	46	5.5	6.65
200	100	14	140	7.1	7.65
400	20	26	1,300	10.35	8.65
1000	5	62	12,000	13.55	10
2000	1	110	110,000	16.75	11

Now let us explore if there exists some c and k such that $T(n)$ is approximately cn^k. Then we would have $\lg(T(n)) \approx k\lg(n) + \lg(c)$ and hence k can be estimated as the "typical value of

$$\frac{\Delta(\lg(T(n)))}{\Delta(\lg(n))}$$

This suggests (though some cases are outliers) a k slightly less than 3 which does not contradict $k = \log_{1.5}(3) \approx 2.7$.

In fact, the running time can be predicted pretty accurately! For the depth of the call tree is given as

depth	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
from	2	3	4	5	7	10	14	20	29	43	64	95	142	212	317	475	712	1067	1600
to	2	3	4	6	9	13	19	28	42	63	94	141	211	316	474	711	1066	1599	2398

Thus, e.g., the call tree for $n = 28$ has the *same* depth, 8, as the call tree for $n = 20$, and hence also the *same* size. In general, this suggests that as n grows, the running time stays almost the same for a long while, and then jumps by a factor 3 as the recursion depth increases by 1. This is confirmed by our experiments: increasing n from 1599 to 1600 increases the running time from 34 seconds to 99 seconds; the former is only 1 second more than when $n = 1100$. Further observe, e.g, that increasing n from 200 to 400 increases the call depth by 2 and accordingly we would expect the running time to multiply by $3^2 = 9$, as is consistent with the above table.

Finally, we must consider what happens when we replace ceilings by floor, and vice versa. Then, for $n = 4$, we would have the recursive calls on $A[1..2]$, $A[3..4]$, and $A[1..2]$. Thus the left half of A never interacts with the right half, and a call of SLOWSORT on say $[8, 4, 6, 2]$ will return $[4, 8, 2, 6]$.