Chapter 34 (I)
Computational Complexity

The following table summarizes all the problems that we have to deal with.

<table>
<thead>
<tr>
<th>Category</th>
<th>Time</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>Solvable</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Easy</td>
<td>$O(p(n))$</td>
<td>Sorting, DFS, MST, SRT...</td>
</tr>
<tr>
<td>Uncertain</td>
<td>?</td>
<td>TSP, NP Complete ...</td>
</tr>
<tr>
<td>Hard</td>
<td>$\Omega(2^n)$</td>
<td>Hanoi Tower problem</td>
</tr>
<tr>
<td>Unsolvable</td>
<td>N/A</td>
<td>Halting problem (CS 3780)</td>
</tr>
</tbody>
</table>

We have looked at many “easy” problems in this course, for which we have found out an algorithm with a polynomial, $p(n)$, price tag.

In CS 3780 Introduction to Computational Theory, we will study all these unsolvable problems.

Now something in between... The Hanoi Tower problem is to move $n$ disks, one at a time, from Stick A to C, with the help of B. When moving, no larger disk can be placed on a smaller one. Check out the course page for more stuff...
Play with Hanoi(3) and beyond...

The key steps are $0 \xrightarrow{\quad} 3 \xrightarrow{\quad} 4 \xrightarrow{\quad} 7$:

Since $H(1) = 1$, and $H(n) = 2H(n - 1) + 1$, it is easy to get that $H(n) = 2^n - 1$ (Cf. Next page), which turns out to be the best deal. 😊

Hence, $H(n) = \Omega(2^n)$. It is thus a hard one. 😞
How much is $H(n)$?

We have that, for $n \geq 2$,

$$H(n) = 2H(n - 1) + 1$$
$$= 2[2H(n - 2) + 1] + 1$$
$$= 2^2H(n - 2) + 1 + 2^1$$
$$= \ldots$$
$$= 2^kH(n - k) + \sum_{i=1}^{k-1} 2^i$$
$$= \ldots$$
$$= 2^{n-1}H(1) + \sum_{i=1}^{n-2} 2^i.$$

Since $H(1) = 1$, and, if you look at Page 8 of the Math preview notes,  
$$\sum_{i=1}^{n-2} 2^i = 2^{n-1} - 1.$$ 

Therefore, for all $n \geq 1$,

$$H(n) = 2^n - 1.$$
Algorithmics vs. complexity

With *algorithmics (CS)*, we want to show that a problem can be solved in $O(f(n))$ by digging out an algorithm with a price tag of *at most* $f(n)$. It should be reduced as much as possible. That is what we have been doing with sorting.

With *complexity theory (MA)*, we try to find a function $g(n)$ as large as possible for which we have to prove that any algorithm that can correctly solve *all* the instances of that problem must take $\Omega(g(n))$, *at least* $\Omega(g(n))$.

We call $g(n)$ the complexity of that problem. It provides a lower bound for any algorithm. If we can find out such a $g(n)$ and show $f(n) \in \Theta(g(n))$, we are done with this problem. 😊

The complexity of a problem is challenging: Given a problem, we have to find out the minimum time needed to solve it, taking into account *all* the possible algorithms that someone might come up with in the future... 😞
The game of 20 questions

**Question:** Assume that someone has picked up a positive number between 1 and one million. How many questions do we need to ask to figure out the number?

**A solution:** A million. 1?, 2?, 3?, ....

**A better one:** Ask first “Is the number between 1 and 500,000?” If your friend answers “yes”, you ask “Is the number between 1 and 250,000?” Otherwise, you ask, instead, “Is the number between 500,001 and 1,000,000?” etc.

As each question cuts the number “by half”, and 1,000,000 < 2^{20}, this approach will find the number in at most 20 steps.

**Question:** Is there an even better solution?

**Answer:** No!

**Question:** Why not?
The complexity of this game

Let $S_i$ be the set of candidates for the number after the $i$th question has been asked and let $k_i$ be its size. For example, $S_0 = [1, 10^6] = \{x | 1 \leq x \leq 10^6\}$ and $k_0 = 10^6$.

Let $Q_i$ be the $i$th question, and let $Q_i(n)$ be the answer to that question for the number $n$. For example, if $Q_1$ is “Is the number larger than 20”, then $Q_1(15)$ is “no”, while $Q_1(25)$ is “yes”.

Let $Y_i$ be $\{n \in S_{i-1} | Q_i(n) = \text{“yes”}\}$ and let $N_i$ be $\{n \in S_{i-1} | Q_i(n) = \text{“no”}\}$. For example, we have that

$Y_1 = \{n | n \in [21, 1,000,000]\}, |Y_1| = 999,980.$

and

$N_1 = \{n | n \in [1,20]\}, |N_1| = 20.$

Clearly, $k_0 = |S_0| = 1,000,000 = |Y_1| + |N_1|$. 
The punch line

As for every number, the answer is either “yes” or ”no”, but not both or neither, we have that, for all $i \geq 1$, $Y_i \cup N_i = S_{i-1}$ and $Y_i \cap N_i = \emptyset$, i.e., $Y_i$ and $N_i$ are disjoint. Therefore, for all $i \geq 1$,

$$k_{i-1} = |S_{i-1}| = |Y_i| + |N_i|.$$ 

Now, either $Y_i$ or $N_i$ has to contain at least $\lceil k_{i-1}/2 \rceil$ numbers. For example, when 7 is cut to 3 and 4, it might always be the case that the smaller one is cut, e.g., we might keep 4, i.e., $k_i \geq \lceil k_{i-1}/2 \rceil$ for each $i \geq 1$. (Notice that for all $n$, $n = \lfloor n/2 \rfloor + \lceil n/2 \rceil$.)

Check out next page for an example...

Thus, it is possible that $k_{19} \geq 2$ (See why on the Page 9), meaning that we need to ask at least one more question, after asking nineteen of them, i.e., at least 20 questions in total in the worst case. This gives its complexity.
Below shows an example of the structure of this process, and an example for the fact that, when \( i = 3 \), either \( Y_3 \) or \( N_3 \) has to contain at least \( \lceil k_2/2 \rceil \) (\( = \lceil |S_2|/2 \rceil \)) numbers.

More details for the general cases are in the next page.
Here are the details...

1. Assume that $|Y_i| < \lceil k_{i-1}/2 \rceil$ (A) and $|N_i| < \lceil k_{i-1}/2 \rceil$ (B), i.e., $|Y_i| \leq \lceil k_{i-1}/2 \rceil - 1$ and $|N_i| \leq \lceil k_{i-1}/2 \rceil - 1$. Thus,

$$k_{i-1} = |S_{i-1}| = |Y_i| + |N_i| \leq 2 \left( \lceil k_{i-1}/2 \rceil - 1 \right).$$

If for some $n \geq 0$, $k_{i-1} = 2n$, we would get $2n \leq 2(n - 1)$, which could not be true. Otherwise, $k_{i-1} = 2n + 1$, then we would have $2n + 1 \leq 2n$, which could not be true, either.

By de Morgan’s law, $\neg (A \wedge B) \equiv \neg A \lor \neg B$, either $|Y_i| \geq \lceil k_{i-1}/2 \rceil$ or $|N_i| \geq \lceil k_{i-1}/2 \rceil$.

For example, if $k_{i-1} = 7$ or $k_{i-1} = 8$, then either $|Y_i| \geq 4$, or $|N_i| \geq 4$.

2. When for $i \geq 1, k_i \geq \lceil k_{i-1}/2 \rceil$, we have the lower bounds for $k_i, i \in [1, 19]$, as follows:

500,000; 250,000; 125,000; 62,500; 31,250; 15,625; 7,813; 3,907; 1,954; 977; 489; 245; 123; 62; 31; 16; 8; 4; 2. Thus, $k_{19} \geq 2$.

We can trust Dr. Shen at least this time. 😊
The complexity of sorting

We have learned several sorting algorithms: \textit{InsertionSort}, \textit{BubbleSort} and \textit{SelectionSort} take $O(n^2)$; \textit{QuickSort} and \textit{TreeSort} take $O(n \log(n))$ on average; finally, \textit{HeapSort} and \textit{MergeSort} take at worst $O(n \log(n))$. All of these algorithms are based on key comparison.

Since any such algorithm corresponds to a decision tree, which must contain at least $n!$ leaves (Page 11 of \textit{Linear-Time Sort} notes). Such a tree must have height at least $\lceil \log(n!) \rceil$ and an average height of at least that much as well.

As $\log(n!) \in \Theta(n \log(n))$, the complexity of key comparison based sorting is $\Omega(n \log(n))$.

Hence, such sorting algorithms as \textit{HeapSort}, \textit{MergeSort}, and \textit{QuickSort} (for the average case) are the best we could get, if we sort things out based on key comparison.
Adversary arguments

Decision tree is a tool of *information-theoretic arguments*, in the form that “to get this much information, you have to make at least this much effort.” For example, to find out the relative order of all the $n$ elements, you have to make at least $n \log n$ comparisons.

Sometimes, *adversary arguments* are more useful, applicable, and effective.

For a given problem on an input, i.e., a problem instance, whenever the algorithm checks the input and asks a question, an intensional *demon*, an adversary, answers in a way that will force the algorithm to work harder so *it will address all the cases as it should*.

Notice that *Algorithmics* tries to solve the current problem instance as fast as possible, while *complexity* wants to make sure that a *correct solution has to work for all the instances*, thus taking enough time to complete.
What’s happening?

Essentially, the demon tries to keep the algorithm uncertain of the correct answer as long as possible.

If the algorithm claims to know the answer prematurely, the demon must be able to construct an input that is consistent with all of its answers to the questions, and show that the correct solution to this input just constructed is different from the output of the algorithm.

Since this just constructed input could be the actual input, it shows that the algorithm has not done enough, yet. 😞

In other words, the problem at hand is more complicated, thus deserving more time to derive its complete, and correct, solution.

Always keep in your mind that correctness precedes efficiency.
The 20 question problem revisited

In the game of 20 questions, the demon doesn’t need to pick a number first 😊: it can wait until a question has been asked and then pick up a number in the bigger range of the two, thus forcing the algorithm to cut off the smaller range. For example, if $Q_1$ is “If the number falls in between 1 and 20?” Its answer must be “No” to serve this purpose. 😞

After asking 19 questions, as we saw earlier, at least two numbers remain, which are consistent with all the previous answers.

If the algorithm claims to know the number at that moment or earlier, the demon can always pick up a number from the leftover and argue that this number is what was in its mind. 😊

Therefore, we have to ask at least 20 questions, but not 19 or less. 😞