1 The Yao Principle (from Minimax)

In classical computing, the *Yao principle* (Yao, 77) gives an equivalence between two kinds of randomness in algorithms: randomness inside the algorithm itself, and randomness on the inputs. Let us fix some model of computation for computing a Boolean function F, like query complexity, communication complexity, etc. Let $R_\epsilon(F)$ be the minimal complexity among all *randomized* algorithms that compute $F(x)$ with success probability at least $1-\epsilon$, for all inputs x. Let $D_\epsilon^\mu(F)$ be the minimal complexity among all *deterministic* algorithms that compute F correctly on a fraction of at least $1-\epsilon$ of all inputs, weighed according to a distribution μ on the inputs. The Yao principle now states that these complexities are equal if we look at the "hardest" input distribution μ:

$$R_\epsilon(F) = \max_\mu D_\epsilon^\mu(F).$$

Since its introduction, the Yao principle has been an extremely useful tool in computational complexity analysis. In particular, it allows us to derive lower bounds on randomized algorithms from lower bounds on deterministic algorithms: choose some "hard" input distribution μ, prove a lower bound on deterministic algorithms that compute f correctly for "most" inputs, weighted according to μ, and then use $R_\epsilon(F) \geq D_\epsilon^\mu(F)$ to get a lower bound on $R_\epsilon(F)$. This method is used very often, because it is usually much easier to analyze deterministic algorithms than to analyze randomized ones. Below we derive this principle from the theory of 2-player games.

Consider the following setting: player 1 has a choice between some m "pure" strategies and player 2 has a choice between n "pure" strategies. If player 1 plays i and player 2 plays j, then player 1 receives "payoff" P_{ij}. Player 1 wants to maximize the payoff, player 2 wants to minimize. Viewing P as an $m \times n$ matrix, and using e_i and e_j to denote the appropriate unit column vectors with a 1 in place i, respectively j, the payoff corresponds to the matrix product $e_i^T P e_j$. However, the players may also use "mixed" strategies (probability distributions over "pure" strategies) to further their goals. Mixed strategies of players 1 and 2 correspond to m- and n-dimensional column vectors ρ and μ, respectively, of non-negative reals that sum to 1. Now the *expected* payoff is $\rho^T P\mu$. Note that if player 1 can choose his strategy ρ knowing player 2's strategy μ, then he would choose ρ to maximize the payoff $\rho^T P\mu$; in this situation player 2 would do best to choose μ to minimize $\max_\rho \rho^T P\mu$, giving expected payoff $\min_\mu \max_\rho \rho^T P\mu$. Conversely, if player 2 could choose his strategy knowing player 1's strategy, then the expected payoff would be $\max_\rho \min_\mu \rho^T P\mu$. The minimax theorem tells us that these two quantities are in fact equal:

$$\min_\mu \max_\rho \rho^T P\mu = \max_\rho \min_\mu \rho^T P\mu.$$

It is not hard to see that without loss of generality the "inner" choices can be assumed to be pure strategies, so as an easy consequence we also have

$$\min_\mu \max_i e_i^T P\mu = \max_\rho \min_j \rho^T P e_j.$$

Yao was the first to interpret this result in computational terms. Player 1 chooses an algorithm to compute function F and player 2 chooses an input x that is hard for player 1. The pure strategies for player 1 are all *deterministic* classical algorithms of query complexity $\leq c$, so his mixed strategies are all *randomized* classical algorithms of complexity $\leq c$. The pure strategies for player 2 are the possible inputs, so his mixed strategies are all possible input distributions μ. We define the payoff matrix such that $P_{ax} = 1$ if deterministic algorithm a computes the function correctly on input x; and $P_{ax} = 0$ otherwise. In this setting, the minimax theorem states

$$\min_\mu \max_a e_a^T P\mu = \max_\rho \min_x \rho^T P e_x.$$

Let us interpret both sides of this equation. On the left, the quantity $e_a^T P\mu$ is the fraction of inputs on which deterministic algorithm a is correct, weighed according to μ, and $\max_a e_a^T P\mu$ denotes this fraction for the optimal deterministic algorithm of complexity $\leq c$. Thus the left-hand-side of the equation gives this optimal correct fraction for the hardest distribution μ achievable by deterministic complexity-c algorithms.

On the other hand, $\rho^T P e_x$ is the success probability on input x achieved by the randomized algorithm given by probability distribution ρ over deterministic algorithms, and $\min_x \rho^T P e_x$ is its success probability on the hardest input. Thus the right-hand-side gives the highest worst-case success probability achievable by randomized complexity-c algorithms. Since these two quantities are equal for all c, we obtain:

$$R_\epsilon(F) = \max_\mu D^\mu_\epsilon(F).$$

2 Classical Lower Bound for Simon's Problem

We will use the Yao principle to prove a classical lower bound for a decision version of Simon's problem:

Given: input $x = (x_0, \ldots, x_{N-1})$ which we can query, where $N = 2^n$
Promise: there is an $s \in \{0,1\}^n$ such that: $x_i = x_j$ iff ($i = j$ or $i = j \oplus s$)
Task: decide whether $s = 0^n$

Consider the input distribution μ defined as follows. With probability 1/2, f is a random permutation of $\{0,1\}^n$; this corresponds to the case $s = 0^n$. With probability 1/2, we pick a non-zero string s at random, and for each $i, i \oplus s$ pair, we pick a unique value for $x_i = x_{i \oplus s}$ at random. Now consider a deterministic algorithm with error $\leq 1/3$ under μ, that makes T queries to x. We want to show that $T = \Omega(\sqrt{2^n})$.

First consider the case $s = 0^n$. We can assume the algorithm never queries the same point twice. Then the T outcomes of the queries are T distinct n-bit strings, and each sequence of T strings is equally likely.

Now consider the case $s \neq 0^n$. Suppose the algorithm queries the indices $i_1, \ldots, i_T$ (this sequence depends on x) and gets outputs $x_{i_1}, \ldots, x_{i_T}$. Call a sequence of queries $i_1, \ldots, i_T$ *good* if it shows a collision (i.e., $x_{i_k} = x_{i_\ell}$ for some $k \neq \ell$), and *bad* otherwise. If the sequence of queries of the algorithm is good, then we can find s, since $i_k \oplus i_\ell = s$. On the other hand, if the sequence is bad, then each sequence of T distinct outcomes is equally likely—just as in the $s = 0^n$ case! We will now show that the probability of the bad case is very close to 1 for small T.

If $i_1, \ldots, i_{k-1}$ is bad, then we have excluded $\binom{k-1}{2}$ values of s, and all other values of s are equally likely. The probability that the next query i_k makes the sequence good, is the probability that $x_{i_k} = x_{i_j}$ for some $j < k$, equivalently, that the set $S = \{i_k \oplus i_j \mid j < k\}$ happens to contain the string s. But S has only $k-1$ members, while there are $2^n - 1 - \binom{k-1}{2}$ equally likely remaining possibilities for s. This means that the probability that the sequence is still bad after query i_k is made, is very close to 1. In formulas:

$$\begin{aligned}
\Pr[x_1, \ldots, x_T \text{ is bad}] &= \prod_{k=2}^{T} \Pr[i_1, \ldots, i_k \text{ is bad} \mid i_1, \ldots, i_{k-1} \text{ is bad}] \\
&= \prod_{k=2}^{T} \left(1 - \frac{k-1}{2^n - 1 - \binom{k-1}{2}}\right) \\
&\geq 1 - \sum_{k=2}^{T} \frac{k-1}{2^n - 1 - \binom{k-1}{2}}.
\end{aligned}$$

Here we used the fact that $(1-a)(1-b) \geq 1 - (a+b)$ if $a, b \geq 0$.

We can approximate the last formula by $1 - T^2/2^n$. Hence T has to be $\Omega(\sqrt{2^n})$ in order to enable the algorithm to get a good sequence of queries with decent probability (if it gets a bad sequence, it cannot "see" the difference between the $s = 0^n$ case and the $s \neq 0^n$ case).