Lower Bounds for Randomized Algorithms

Two former students

December 5, 2014

1 Introduction

In this paper, we present a method to find a lower bound on the running time for any Las Vegas randomized algorithm. We first derive Yao's Minimax Principle from the analysis of two-player games. Applying the result of Yao's Minimax Principle then gives us our desired lower bound on Las Vegas randomized algorithms. More specifically, by applying Yao's Minimax Principle, we will show that in order to find a lower bound on running time for a Las Vegas randomized algorithm, it suffices to find a lower bound on the running time of any deterministic algorithm. That is, no Las Vegas randomized algorithm can solve a problem faster than the best possible deterministic algorithm. We will then apply Yao's Minimax Principle to a Las Vegas algorithm used to evaluate game trees and to the set of all algorithms that use comparisons to sort.

2 Yao's Minimax Principle

2.1 Rock! Paper! Scissors!

We begin our discussion of two-player games with the analysis of Rock-Paper-Scissors. Here, a *strategy* is defined as the set of moves that a player selects throughout the game. Our analysis of Rock-Paper-Scissors will be limited to the use of *pure strategies*, where each player can definitively choose a particular strategy to pursue.

For those unfamiliar with the game, Rock-Paper-Scissors has two players, Ryan and Charlotte, who place their hands behind their back and make a sign for rock, paper or scissors. They simultaneously choose and display a sign. The winner is determined by the following rules: rock beats scissors, scissors beats paper and paper beats rock. The loser pays the winner \$1.

The possible payoffs for Rock-Paper-Scissors are listed in Table 1. Table 1 is called a payoff matrix and is denoted by $\mathbf{M}$. The set of possible strategies for Ryan, our row player (R) are described in each row of $\mathbf{M}$. Similarly, the set of possible strategies for Charlotte, our column player (C) are described in each column of $\mathbf{M}$. Each entry $M_{i,j}$ describes the amount that C pays R when C selects strategy j and R selects strategy i. For instance, entry $M_{1,2} = 1$ corresponds to R selecting Scissors and C picking Paper, so C pays R \$1. On the other hand, $M_{2,1} = -1$ corresponds to R picking Paper and C selecting Scissors, so R pays C \$1. If we calculate the expected winnings of C and R, we find that they are both 0. Therefore, Rock-Paper-Scissors is what is known as a $zero - sum$ game.

	Rock	**Paper**	**Scissors**
Rock	0	1	-1
Paper	-1	0	1
Scissors	1	-1	0

Table 1: The payoff matrix for Rock-Paper-Scissors

Clearly, R would like to maximize his payoff and C would like to minimize her payoff. Now, if R picks a particular strategy i, then he is guaranteed a payoff of $min_j M_{i,j}$. However, R wants to maximize his payoff so his optimal strategy is to pick the i that maximizes $min_j M_{i,j}$. Then, the lower bound on a payoff when R uses an optimal strategy is described by $V_r = max_i min_j M_{i,j}$. Similarly, if C picks a particular strategy j, she is guaranteed a maximum payoff of $max_i M_{ij}$. She would like to minimize her payoff, so her optimal strategy would be to select the strategy j that minimizes $max_i M_{ij}$. Then, the best upper bound on a payoff when C uses an optimal strategy is described by $V_c = min_j max_i M_{ij}$.

Returning to our game of Rock-Paper-Scissors, we find that $V_r = -1$ and $V_c = 1$. Notice that $V_r \leq V_c$. In fact, this inequality holds true for all payoff matrices. That is,

$$V_r \leq V_c$$

or

$$max_i min_j M_{ij} \leq min_i max_j M_{ij} \text{ [3]}$$

If $V = V_r = V_c$, then we say that the game has a *solution* and call the corresponding strategies ρ and γ. In our original version of Rock-Paper-Scissors, depicted in Table 1, we saw that $V_r \neq V_c$, which implies that the game has no solution. Consider the modified payoff matrix shown in Table 2. Here, $V_r = V_c = 0$ with $\gamma = 1$ and $\rho = 1$, which implies that our modified Rock-Paper Scissors does in fact have a solution.

	Rock	**Paper**	**Scissors**
Rock	0	1	2
Paper	-1	0	1
Scissors	-2	-1	0

Table 2: The payoff matrix for a modified version of Rock-Paper-Scissors

2.2 Randomized Strategies

In the previous section, we discussed an example of a pure strategy, where each player selects a particular strategy. As it's name suggests, a randomized or mixed strategy introduces some randomness into our model. Now, instead of selecting a particular strategy, each player will pick a probability distribution over all strategies. In turn, the row player will now pick a vector $\mathbf{p} = (p_1, ..., p_n)$, which is a probability distribution over all rows. Similarly, the column player chooses a vector $\mathbf{q} = (q_1, ..., q_n)$, which is a probability distribution over all columns. Notice that both $\mathbf{p}$ and $\mathbf{q}$ are column vectors and p_i denotes the ith element of $\mathbf{p}$ and q_j denotes the jth item in $\mathbf{q}$. If follows then that each p_i and q_j is the probability that the ith and jth strategies will be played respectively. We continue to denote our payoff matrix as $\mathbf{M}$ where M_{ij} corresponds to the payoff that C pays R when the ith row strategy and jth column strategy is played.

Since we have introduced some randomness into model, our payoff is now a random variable. Let X be the payoff for the game. Then, $Pr(X = M_{ij}) = p_i q_j$. Then, our expected payoff is described by

$$\mathbf{E}[X] = \textstyle\sum_{j=1}^{m} \sum_{i=1}^{n} p_i q_j M_{ij} = \mathbf{p}^T \mathbf{M} \mathbf{q}. \text{ [3]}$$

We now define $V_R = max_p min_q \mathbf{p}^T \mathbf{M} \mathbf{q}$ where $\mathbf{q}$ is the probability distribution that minimizes $\mathbf{p}^T \mathbf{M} \mathbf{q}$ for all $\mathbf{p}$. We subsequently select the $\mathbf{p}$ that maximizes $\mathbf{p}^T \mathbf{M} \mathbf{q}$. V_C is similarly defined as $V_C = max_p min_q \mathbf{p}^T \mathbf{M} \mathbf{q}$. V_R and V_C retain their previous meanings. That is, V_R describes the best lower bound on the expected payoff that R can guarantee and V_C is the best upper bound on the expected payoff that C can guarantee. Interestingly, when R and C use mixed strategies, every two-person, zero sum game has a solution. This is stated in Von Neumann's Minimax theorem.

Theorem 2.1 (von Neumann's Minimax Theorem). *For any two person zero-sum game specified by a matrix* $\boldsymbol{M}$*,*

$$max_p min_q \boldsymbol{p}^T \boldsymbol{M} \boldsymbol{q} = min_q max_p \boldsymbol{p}^T \boldsymbol{M} \boldsymbol{q}. \text{ [3]}$$

Theorem 2.2 (Loomis' Theorem). *For any two person zero-sum game specified by a matrix* $\boldsymbol{M}$*,*

$$max_p min_j \boldsymbol{p}^T \boldsymbol{M}\ \boldsymbol{e}_j = min_q max_i \boldsymbol{e}_i^T \boldsymbol{M} \boldsymbol{q}. \text{ [3]}$$

2.3 The Ultimate Showdown: Algorithm Designer vs. Adversary

To see how the previously discussed game theoretic techniques apply to randomized algorithms, consider a problem Π. We assume that there are a finitely many deterministic algorithms that solve Π. Furthermore, the set of distinct inputs for each deterministic algorithm is finite. In this framework, the column player now becomes the algorithm designer and the row player is our adversary who selects the input. The rows of $\mathbf{M}$ correspond to the set of all possible inputs. The columns of $\mathbf{M}$ corresponds to the set of all deterministic algorithms that complete in finite time and always outputs the correct value. The entries of the $\mathbf{M}$ still correspond to the payoff, which in this context, is any real-valued measure of the performance of the algorithm such as running time or computational cost. If our payoff is running time, then M consists of the running times for all algorithms and for all possible inputs. Each entry M_{ij} in $\mathbf{M}$ corresponds to the running time of the jth algorithm when the adversary selects the ith input. Naturally, the algorithm designer would like to minimize the running time of the algorithm and the adversary would like to maximize the running time of the algorithm.

Now, in the context of our game, a mixed strategy for the algorithm designer is a probability distribution over the space of all deterministic algorithms that always output the correct answer. A mixed strategy for the adversary is a probability distribution over the space of all possible inputs. We can then restate Von Neumann's and Loomis' theory in the language of randomized algorithms.

Theorem 2.3. *Let Π be a problem with a finite set $\mathcal{I}$ of input instances (of a fixed size), and a finite set of deterministic algorithms $\mathcal{A}$. For input $I \in \mathcal{I}$ and algorithm $A \in \mathcal{A}$, let $C(I, A)$ denote the running time of algorithm A on input I. For probability distributions $\boldsymbol{p}$ over $\mathcal{I}$ and $\boldsymbol{q}$ over A, let I_p denote the random input chosen according to p and A_q denote a random algorithm chosen according to $\boldsymbol{q}$. Then,*

$$max_p min_q E[C(I_p, A_q)] = min_q max_p E[C(I_p, A_q)]$$

and

$$max_p min_{A \in \mathcal{A}} E[C(I_p, A)] = min_q max_{I \in \mathcal{I}} E[C(I, A_q)]. \text{ [3]}$$

Yao's Minimax Principle follows directly from Theorem 2.3. It is stated as follows:

Proposition 2.4 (Yao's Minimax Principle)**.** *For all distributions $\boldsymbol{p}$ over $\mathcal{I}$ and $\boldsymbol{q}$ over $\mathcal{A}$,*

$$\min_{A \in \mathcal{A}} \boldsymbol{E}[C(I_{\boldsymbol{p}}, A] \leq \max_{I \in \mathcal{I}} \boldsymbol{E}[C(I, A_{\boldsymbol{q}}]. \text{ [3]}$$

In other words, Yao's Minimax Principle states that the expected cost for a las vegas randomized algorithm is greater than the expected cost for the best deterministic algorithm for any distribution on the inputs. Therefore, the expected running time for the best deterministic algorithm for an arbitrary distribution on the inputs is a lower bound for our randomized algorithm. While we are only applying Yao's Minimax Principle to the running time of Las Vegas randomized algorithms, it is interesting to note that $C(I, A)$ could also denote, for example, the memory used by algorithm A on input I. Thus Yao's Minimax Principle could, in fact, be used to find lower bounds other than just running time for Las Vegas randomized algorithms.

3 Applications of Yao's Minimax Principle

3.1 Game Trees

Our first application of Yao's Minimax Principle is to the evaluation of game trees. Game trees represent a game with two players, one of which is trying to maximize a certain score and the other is trying to minimize this score. Then, we define a game tree as a rooted tree with MIN and MAX nodes and each leaf of the tree is assigned an numeric value. For simplicity, each leaf will be assigned either a 0 or a 1 as its numeric value. An example of a game tree is depicted in Figure 1. To evaluate a game tree, we note that each leaf *returns* its own numeric value, each MAX node returns the maximum of its childrens' values, and each MIN node returns the minimum of its childrens' values. Thus to evaluate a game tree, we must find what the root returns.

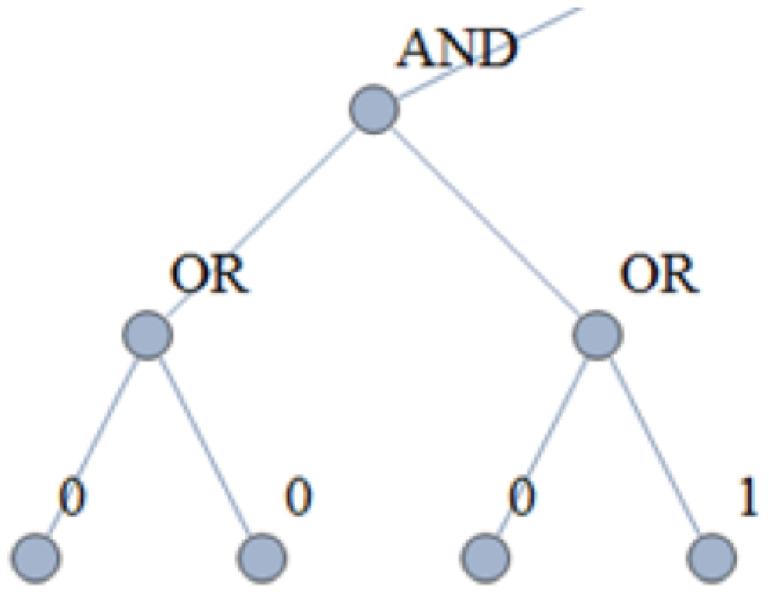

Figure 1: Example of a $T_{2,2}$ Game Tree

Let $T_{d,k}$ [3] be a game tree in which each node has d children and every leaf is distance $2k$ from the root. Thus any root to leaf path must pass through k AND nodes and K OR nodes. Every $T_{d,k}$ has d^{2k} leaves. We would like to consider the evaluation of $T_{2,k}$, where we let $d = 2$ for simplicity. We will use Yao's Minimax Theorem to find a lower bound on any Las Vegas randomized algorithm that could be used to evaluate $T_{2,k}$.

3.2 Expected Running Time of a Randomized Evaluation

We now present a game tree evaluation Las Vegas randomized algorithm with an expected running time that is faster than the worst case of any deterministic algorithm. Then, by applying Yao's Minimax theorem, we can find a lower bound for the expcted running time of our algorithm.

Before we present our algorithm, consider the evaluation of a node v in a $T_{2,k}$. Assume that v is a MAX node. Since v is a MAX node, if v returns 1, then at least one of its children returns 1. If v returns 0, then both of its children must return 0. Now, if we wish to determine the value of v, we can begin by randomly picking one of its children. If the first child that we chose to evaluate returns 1 then v returns 1 regardless of the value of the other child. However, if the first child that we select returns 0, we must inspect the other child. Our algorithm then follows naturally.

The key difference between **RandomEval** and a deterministic algorithm is the evaluation of each node's children. Here, for a particular node, we randomly select one of its children to be evaluated first. In constrast, a deterministic algorithm visits each child in a particular order.

Theorem 3.1. *The expected number of leaves that* ***RandomEval*** *must inspect*

```
**RandomEval**
**Input**: A $T_{2,k}$ and a node $v$ to evaluate.
**Output**: The evaluation of node $v$ (either a 0 or 1).
1. Randomly pick a child $v_1$ of $v$.
2. If $v$ is an MAX node
    2a: If $v_1$ returns 1, return 1.
    2b: If $v_1$ returns 0, call **RandomEval** on $v_1$.
3. If $v$ is an MIN node
    3a: If $v_1$ returns 1, return 1.
    3b: If $v_1$ returns 0, call **RandomEval** on $v_1$. [3]
```

on a $T_{2,k}$ is less than or equal to 3^k [2].

Proof. Let X be the number of children we need to evaluate in order to evaluate v. With an inductive proof, we show that the expected number of leaves that are inspected in **RandonEval** less than or equal to 3^k.We split this problem into two cases:

Case 1 - v returns 1: Since v is a MAX node, at least one child must return 1. In the best case, both children are 1, and thus $\mathbf{E}[X] = 1$ no matter what child we randomly pick first. In the worst case, v has one child v_1 that returns 1 and one child v_2 that returns 0. Then we have a $\frac{1}{2}$ chance of randomly picking v_1 first, which would require us to only have to evaluate one child. We also have a $\frac{1}{2}$ chance of picking v_2 first, which requires evaluating both children. Therefore, no matter what the children evaluate to,

$$\mathbf{E}[X] \leq \tfrac{1}{2}(1) + \tfrac{1}{2}(2) = \tfrac{3}{2}$$

Case 2 - v returns 0: Both children must return 0. Thus, no matter which child is picked first, both children must be evaluated. Thus

$$\mathbf{E}[X] = 2$$

We now begin our induction. We assume that the root node of our game tree is a MIN node, as the proof for a MAX node easily follows. Let G be the number of grandchildren evaluated.

Base case: Consider a $T_{2,1}$. In this case, G also gives us the running time of the algorithm, as the grandchildren of the root node are leaves.

If the root returns 1, then both of its children must both return 1. Since the children must both be MAX nodes, we just proved that $\mathbf{E}[X]$ for both children is less than or equal to $\frac{3}{2}$. Therefore $\mathbf{E}[G] \leq 2 \cdot \frac{3}{2} = 3$.

If the root returns 0, then its children either both return 0 or one child returns a 1 and the other returns a 0. If both return a 0, then no matter which child we pick, we only need to evaluate one child. Since the children are MAX nodes, we just proved that $\mathbf{E}[X]$ for both children is 2. Therefore $\mathbf{E}[G] = 2$. If one returns a 0 and the other returns a 1, then, in the worst case, we pick the one that returns 1. Then we have to evaluate that node with $\mathbf{E}[X] \leq \frac{3}{2}$ and evaluate the other node that returns 0 with $\mathbf{E}[X] = \frac{1}{2}$. Therefore $\mathbf{E}[G] \leq 2 + \frac{3}{2} = \frac{7}{2}$.

Using Bayes' Theorem and conditional probability, we find the expected running time of evaluating the root in either case. First, we find the expected value given that v returns 0.

$$\mathbf{E}[G|\text{return0}] \leq \left(\frac{1}{3}\right)\frac{3}{2} + \left(\frac{2}{3}\right)\frac{7}{2} = 3.$$

We already know that

$$\mathbf{E}[G|\text{return1}] \leq 3.$$

Therefore, $\mathbf{E}[G] \leq 3$ no matter what v returns. Thus for $T_{2,1}$, the running time is definitely bounded by $3^1 = 3^k$.

Inductive Hypothesis: Let X_k be the running time of $T_{2,k}$. Assume that $\mathbf{E}[X_k] \leq 3^{k-1}$. The grandchildren of the root of a $T_{2,k}$ are each a $T_{2,k-1}$. Since these grandchildren are evaluated independently of one another, we can use the property of linearity of expectations. Since we just showed that the number of grandchildren of the root we must evaluate is at most 3, we find that

$$\mathbf{E}[X_k] \leq 3 \cdot \mathbf{E}[X_{k-1}] \leq 3 * 3^{k-1} = 3^k.$$

[2] □

Since there are 4^k leaves, we can solve for the expected running time of our algorithm in terms of the number of leaves it needs to evaluate. Let n be the number of leaves.

$$\begin{aligned} 3^k &= n^x \\ 3^k &= 4^{kx} \\ k\log_4 3 &= kx \\ x &= log_4 3 \end{aligned}$$

Therefore, the expected running time of our algorithm is $n^{\log_4 3} \approx n^{0.793}$ [3]. Interestingly, we note that an adversary can make any deterministic algorithm look at all n leaves, since a deterministic algorithm must look at leaves in a fixed order. Since $n^{0.793} < n$, we see that our randomized performs better than the worse case of any deterministic algorithm.

3.3 A Lower Bound for the Evaluation of Game Trees

In order to simplify our analysis, we will slightly modify our representation of game trees. First, as depicted in Figure 2 since we each leaf is assigned a binary (0 or 1) value, each MAX node is equivalent to a Boolean OR operation and each MIN node is equivalent to a boolean AND operation.

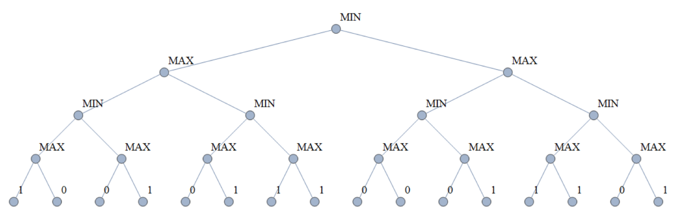

Figure 2: Truth Table: MAX/MIN to OR/AND

Second, we realize that we can replace all of the internal nodes of our game tree with nodes that compute the boolean NOR function. The NOR function only returns 1 if both inputs are 0 and returns 0 otherwise. This replacement will yield equivalent values to our previous tree with AND and OR nodes.

Using Yao's Minimax Theorem, we know that if we set a probability distribution p of inputs and use p to find a lower bound on any deterministic algorithm to evaluate $T_{2,k}$, we will have found a lower bound on the expected running time of any Las Vegas randomized algorithm to evaluate $T_{2,k}$. Therefore, our first step is to set a probability distribution. Let $p = \frac{3-\sqrt{5}}{2}$ [3], and let each leaf of $T_{2,k}$ be set to 1 with probability p. Notice that if a NOR node v has two inputs that are 1 with probability p, then we can compute the probability that v outputs 1 by realizing that both children must return 0 in order for this to occur. Thus

$$\begin{aligned} Pr(v \text{ returns } 1) &= (Pr(\text{both children return } 0))^2 \\ &= (1-p)^2 \\ &= \left(\frac{\sqrt{5}-1}{2}\right)^2 \\ &= \frac{3-\sqrt{5}}{2} = p \end{aligned}$$

Therefore any node in our game tree returns 1 with probability p. Now, instead of having to consider every possibly deterministic algorithm that could evaluate a $T_{2,k}$, we will only consider a certain type of deterministic algorithm. This algorithm will evaluate the root of a game tree through a depth-first search of the tree that stops searching once it has determined the value of the root. This algorithm can perform as well as any deterministic algorithm, so a lower bound on it will be a lower bound for any deterministic algorithm.

Theorem 3.2. *The lower bound on the expected running time of any Las Vegas randomized algorithm that evaluates game trees with n leaves is $n^{0.694}$. [3]*

Proof. Consider a $T_{2,k}$ with internal NOR nodes and n leaves. Let $W(h)$ be the expected amount of work for evaluating a node distance h from the leaves. That is, $W(h)$ is the expected number of leaves our algorithm will have to examine when evaluating a node distance h from the leaves. Since the root is distance $\log_2 n$, we want to find $W(\log_2 n)$. Using a recursive algorithm, we find that

$$W(h) = W(h-1) + (1-p)W(h-1). \text{ [3]}$$

This is true because in order to evaluate a node at distance h from the leaves, we must evaluate at least one of its children, and we must evaluate the other child if the first child returns 0. If we plug in $h = \log_2 n$, we can solve this recursive formula to find that $W(\log_2 n) \geq n^{0.694}$. Using Yao's Minimax Principle, this bound also applies to any Las Vegas randomized algorithm. □

This lower bound is less than the expected running time of our randomized algorithm, which was $n^{0.793}$. One possibility for this discrepancy is that our probability distribution was not the best we could have chosen. However, this lower bound does apply to all Las Vegas randomized algorithms that evaluate game trees.

3.4 Sorting Algorithms

We would like to find a lower bound for sorting algorithms that use comparisons to sort lists - for example, Quicksort and Bubblesort. In order to bound these algorithms, we will create a model called a *decision tree* that will represent every sorting algorithm that uses comparisons [4]. This model will be a binary tree, in which the *execution* of the algorithm is a root-to-leaf path. Each internal node corresponds to a comparison, and the leaves are the results of the execution (the input list in a certain order). In order to find a lower bound on the running time of these sorting algorithms, it will suffice to find a lower bound on the possible height of our decision tree.

Theorem 3.3. *The lower bound on the running time of any Las Vegas sorting algorithm is $\Omega(n \log n)$. [4]*

Proof. We will assume that we are sorting a list of n distinct numbers. Since the leaves correspond to the possible orders of this list, there must be $n!$ leaves. We also know that any height h tree has no more than 2^h leaves. Thus, using the logarithm laws,

$$\begin{aligned}
2^h &\geq n! \\
h &\geq \log_2(n!) \\
&= \log_2((n)(n-1)(n-2)(n-3)...(2)) \\
&= \log_2(n) + \log_2(n-1) + \log_2(n-2) + ... + \log_2(2) \\
&= \sum_{i=2}^{n} \log_2(i) \\
&= \sum_{i=2}^{\frac{n}{2}-1} \log_2(i) + \sum_{i=\frac{n}{2}}^{n} \log_2(i) \\
&\geq \sum_{i=\frac{n}{2}}^{n} \log_2(\frac{n}{2}) \\
&= \frac{n}{2} \cdot \log_2\left(\frac{n}{2}\right) \\
&= \Omega\left(n \log_2(n)\right)
\end{aligned}$$

[4]

Using Yao's Minimax Principle, since $\Omega\left(n \log_2(n)\right)$ is a lower bound on the running time of any deterministic sorting algorithm, $\Omega\left(n \log_2(n)\right)$ is a lower bound on the running time of any Las Vegas randomized sorting algorithm.

□

References

[1] Habib, Michael. "The Minimax Principle and Lower Bounds." *Probabilistic Methods for Algorithmic Discrete Mathematics.* Ed. Colin McDiarmid. 1998 ed. Vol. 16. N.p.: Springer, n.d. 28-37. Print. Algorithms and Combinatorics.

[2] Jop, Sibeyn. "Game Theoretic Techniques" *Users.informatik.uni-halle.de,* 2014. [Online].[Accessed: 14- Dec- 2014].

[3] Motwani, Rajeev, and Prabhakar Raghavan. "Game-Theoretic Techniques." *Randomized Algorithms.* 1st ed. N.p.: Cambridge UP, n.d. N. pag. Print.

[4] Toma, Laura. *Sorting Lower Bound.* 1st ed. Bowdoin College, 2007.