Lower Bounds for Geometric Problems

Subhash Suri

November 21, 2019

1 Lower Bounds and Models of Computation

- Proving computational complexity lower bounds is generally much harder than designing efficient algorithms. Even when it seems intuitively obvious that a certain way of solving a problem is probably the most efficient one, converting that intuition into a rigorous proof has often been either elusive or *wrong*.
- The most famous case of elusiveness is the P vs. NP. In spite of general believe that a problem such as 3-SAT cannot be solved in sub-exponential time, there is no proof.
- Another famous example of our intuition being flawed is the complexity of matrix multiplication. For long time, it seemed "obvious" that the classical method of multiplying two $n \times n$ matrices in $O(n^3)$ was the best, until Strassen's shocking discovery.
- In this lecture, we will explore some techniques for arguing about lower bounds.
- To analyze an algorithm's performance, or to reason about the intrinsic complexity of a computational problem, one needs a *formal model* of computation, which specifies the *primitive operations* that may be executed and their *costs*.
- Examples include Turing Machine, or Random Access Model.
    1. The primary difference for our purposes between these models is how the manipulation of individual numbers is treated.
    2. TM uses bits, and so to add two k bits numbers has $O(k)$ cost—namely, the cost grows in proportion to the operand length.
    3. RAM allows two numbers to be manipulated in constant time, in line with the hardware of digital computers, with the implicit assumption that each number fits in a hardware *word*.
- Geometric computation introduces another level of complexity: even when the input numbers are small integers, their geometric calculations may entail more complex numbers, including irrationals. Length of the diagonal of a unit square, for instance.

- Therefore, a more appropriate model for analyzing geometric algorithm is the so-called *Real RAM*. (It allows us to dispense with round-off errors in the approximate representation of real numbers, but we should make sure our software libraries provide mechanisms to deal with these overflows and roundoffs, when needed.
- In simple terms, the Real RAM allows each memory location to *hold a single real number, and allows the following primitives at O(1) cost*:
  1. Arithmetic $(+, -, \star, /)$
  2. Comparison $(<, \leq, =, \neq, \geq, >)$
  3. Indirect addressing with integer addresses
  4. kth root, trig functions, analytic functions (exp, log etc).
- This model fairly closely captures the primitives of all modern programming languages.

2 ADT—Algebraic Decision (Computation) Tree

- While Real RAM is the right model for designing algorithms, it is not terribly well-suited for proving lower bounds.
- Instead, a slightly different but *computationally equivalent* model, called the Algebraic Decision (Computation) Tree (ADT) is more convenient.
- ADT mimics the way we think about "programs" or algorithms—as an interleaving of *compute* and *branch* instructions.
- Specifically, assume the input involves a set of real variables $x_1, \ldots, x_n$. Then, a ADT is a program with statements $L_1, L_2, \ldots, L_p$ of the form:
  1. Compute a function $f(x_1, \ldots, x_n)$.
  2. If $f > 0$, go to statement L_i; else go to L_j.
  3. Halt and output YES, or Halt and output NO.
- The function f is an *algebraic function* (a polynomial of some degree).
- We assume that the program has been "unrolled" so it has no loops. Therefore, it has the structure of a *tree* T, where each internal node v is associated with a polynomial function evaluation and comparison:

$$f_v(x_1, \ldots, x_n) > 0 \ ?$$

  1. We can always collapse all the computation that occurs between two comparison nodes into the next comparison node, without loss of generality.

2. The ADT is a dth order tree if d is the largest degree used. 1st order tree is also called Linear Decision Trees—only linear functions are evaluated.
3. If you recall the comparison-based sorting lower bound, it only used a linear decision tree.
4. We also assume, wlog, that the tree is a binary tree—comparisons are binary. Any k-way tree can be simulated by a sequence of $k-1$ binary comparisons, if needed.

- **Remark:** We do not allow any *randomization* in our programs, but similar, albeit slightly more complicated, arguments apply to randomized versions as well.
- Imagine running your program over *all possible* inputs: each execution corresponds to a root-to-leaf path in this ADT, and so we are interested in figuring out its longest path.
- To prove lower bound on a problem's complexity, we will argue that any ADT for solving that problem has a long path. The worst-case complexity of the program is (at least) proportional to the *longest path in the tree.*

2.1 Using ADT for Lower Bounds

- The central idea, which originated with Steele-Yao and Ben-Or (1982–83), is simple but abstract. (Such abstraction is necessary to be able to model all possible algorithms within the model constraints.)
- First, we will only consider Decision problems because any optimization problem is at least as hard as its decision counterpart.
- Suppose each instance of a decision problem involves n real-valued variables $x_1, \ldots, x_n$. Such an instance corresponds to a *point in the n-dimensional Euclidean space R^n.* (For instance, a set of n points in two-dimensional plane is completely specified by $2n$ coordinates, and therefore represented by a point in R^{2n}.)
- Some instances of the Decision problem evaluate to YES, others to NO; otherwise the problem is trivial.
- Let W be the subspace of R^n that contains all the YES instances—the algorithm outputs YES if and only if $(x_1, \ldots, x_n) \in W$.
- The subset $W \subset R^n$ can have complicated structure. We will be mainly interested in how many *connected components* does it have. So, let $\#W$ denote the number of *disjoint connected components of W.*

  Figure.

- Suppose T is the ADT corresponding to an algorithm that solves this problem. Each execution of T traverses a unique path $v_1, \ldots, v_l$, where v_1 is the root, and v_l the leaf node.

- Each node v_j of this path is associated with a function $f_{v_j}(x_1, \ldots, x_n)$, and any specific path in the tree is uniquely specified by the *compute-and-branch* patterns at each of its nodes, where at each node v_j we have

$$f_{v_j} = 0, \text{ or } f_{v_j} > 0, \text{ or } f_{v_j} \geq 0$$

2.2 Argument for the Linear Functions

- The intuitive part of the proof technique is best understood by restricting the ADT to only *linear functions*. This is the (easier) framework introduced in Dobkin-Lipton (1979).

- Let T be the binary linear decision tree embodying the algorithm A that solves the membership in W.

- Associated with each leaf of T is a region of R^n, and each leaf is either "accepting" or "rejecting." (This is the final node in the tree, so the algorithm must output the answer.)

- Suppose

  1. $W_1, \ldots, W_p$ are the (connected) components of W,
  2. $l_1, \ldots, l_r$ the set of leaves of T, and
  3. $D_j \subset R^n$ is the domain associated with leaf l_j.

- By definition of the algorithm's correctness, l_j is accepting if and only if $D_j \subset W$. (Either the whole D_j is accepting, or the whole D_j is rejecting.)

- The lower bound is shown by proving that

$$r \ \geq \ \#W$$

  That is, T must have as many leaves as connected components of W. That in turn implies that *the height of* T *is* $\geq \ \log_2 \#W$.

- We construct a function $\alpha : \{W_1, W_2, \ldots, W_p\} \to \{1, 2, \ldots, r\}$, defined by

$$\alpha(W_i) \ = \ \min\{j \mid j \in \{1, 2, \ldots, r\} \text{ and } D_j \cap W_i \neq \emptyset$$

  That is, $\alpha(W_i)$ to is the lowest index leave whose domain intersects W_i.

- Our lower bound argument rests on showing that two different connected components W_i, W_j have different α indices, and so the number of leaves $r \geq \#W$. The proof is as follows.

  1. Assume, for the sake of contradiction, that $\alpha(W_i) = \alpha(W_j) = h$.
  2. Since the algorithm A solves the membership problem (YES/NO) for a point $q \in W_i$, and W_i is a part of YES subset, the leaf l_h must be accepting (type YES).
  3. On the other hand, by definition of α, we have $W_i \cap D_h \neq \emptyset$. Similarly, $W_j \cap D_h \neq \emptyset$
  4. Therefore, we can pick a point $q' \in W_i \cap D_h$ and a point $q'' \in W_j \cap D_h$.
  5. But since T is a *linear* decision tree, the region D_h is the intersection of halfspaces in R^n, and therefore a *convex set*.
  6. Therefore, any convex combination of q' and q'' must also lie in D_h. In particular, the entire line segment $q'q''$ lies in D_h.
  7. But since q' and q'' lie in disjoint components W_i and W_j, there is at least one point q''' on this segment such that $q''' \notin W$.
  8. This is our contradiction: the segment cannot lie in $D_h \subset W$ and still have a point outside W.
  9. Thus, T has at least as many leaves as number of components in W.

- Since T is binary, we get that its height is at least $\log_2 \#W$.

2.3 Extension to Algebraic Functions (ADT)

- The main difficulty with the previous argument is that if the functions f are *non-linear*, the domain associated with a leaf is no longer *convex* or (most importantly) *connected*.

- The joint result of degree d polynomial inequalities can be quite complex and highly disconnected. How many pieces?

- To make progress on this question requires ideas from algebraic geometry, and builds on important results proved by Milnor and Thom (1960s).

- The intuitive idea is this: suppose we take a number of polynomial functions, each of degree at most d, in m-dimensional space: $g_i(x_1, x_2, \ldots, x_m) = 0$. Then the number of connected components in the solution set of these equations is upper bounded as

$$d(2d-1)^{m-1}$$

- In order to import this ideas to our lower bound, we need to make sure we can handle *polynomial inequalities* and not just equations, and that was done by Steele-Yap and Ben-Or.

- What SY and Ben-Or show is this: Suppose h is the depth of the ADT tree T, corresponding to our algorithm A, operating on a problem with n variables, using degree d polynomial functions. Then, T has at most 2^h leaves, and each leaf accounts for at most $d(2d-1)^{n+h-1}$ components of W.
- Therefore, following the same line of logic as before, we get

$$\#W \leq 2^h d(2d-1)^{n+h-1}$$

- In simplified form, it gives the lower bound on the height of T (running time of A):

$$h \geq \frac{\log_2 \#W}{1+\log_2(2d-1)}$$

3 Lower Bounds for Geometric Problems

3.1 Lower bound for Element Distictness

- We now prove lower bound for a concrete problem called *element distinctness*.
- Given a set of n numbers $x_1, \ldots, x_n$, decide if they are all distinct. That is, $x_i \neq x_j$, whenever $i \neq j$.
- This should be "easier" than sorting. Is it?
- Let $W \subset R^n$ denote the set of all *YES* instances of the problem, namely, instances where elements are all unique.
- How many connected components does W have?
- **Claim:** $\#W = n!$.
- **Proof.** Recall that $n!$ is the number of distinct permutations of $\{1, 2, \ldots, n\}$.

  1. Each instance $\{x_1, x_2, \ldots, x_n\} \in W$ can be identified with the unique permutation of its numbers.
  2. We claim that each connected component contains only points identified with the same permutation. If not, then let p, p' be two instances with difference permutations, but within the same connected component.
  3. Without loss of generality, suppose that permutations p, p' differ in ordering of elements i and j. In other words, $x_i < x_j$ in p but $x_i > x_j$ in p'.
  4. Since p, p' are in the same component, there is a "path" connecting them, and each point on this path is also a valid YES instance of the problem.

5. Therefore, there is a sequence of valid instances that starts at p (where $x_i < x_j$) but ends at p' (where $x_i > x_j$).
6. But in order for the order to switch, at least two elements must become equal at some intermediate point.
7. But having two equal items means the instance is not a valid YES instance, and thus not in W. Contradiction!

- Using our ADT Theorem, therefore any algebraic decision tree algorithm for Element Distinctness of n numbers must take $\Omega(n \log n)$ time.
- The lower bound assumes that the number are *reals*. What if numbers are rationals or integers?
- The ADT argument doesn't work. However, even for integer numbers, the $\Omega(n \log n)$ lower bound holds, by an extension proved by Lubiw and Racz (1991).
- By the way notice that hashing is not a valid algorithm in our ADT model, and hence the lower bound does not apply to that scheme, which is good because we can solve this problem in $O(n)$ time using hashing.

3.2 Other Geometric Problems

- The Element Distinctness Problem turns out to be key to proving similar lower bounds on many other problems.
- **Set Disjointness.** Given two sets of numbers $\{a_1, a_2, \ldots, a_n\}$ and $\{b_1, b_2, \ldots, b_n\}$, decide if $a_i = b_j$ for some i, j.
  - Element Distinctness is a special case, with a and b sequences being the same.
- **Maximum Gap.** Given a set of n (unsorted) numbers $x_1, x_2, \ldots, x_n$, what is the maximum gap between two consecutive numbers (in sorted order)?
- **Diameter of 2D Set.** Given a set of n points in $2D$ plane $p_1, p_2, \ldots, p_n$, find the maximum distance between any two points.
  - First, how difficult is this problem in $1D$?
  - In 2D, we reduce Set Disjointness to Diameter.
  - Let $\{a_1, a_2, \ldots, a_n\}$ and $\{b_1, b_2, \ldots, b_n\}$ be the input.
  - For each a_i, produce a point in the 2D plane where the line $y = a_i x$ intersects the unit circle on the right (positive x) side.
  - Specifically, each a_i maps to the point $p_i = (x_i, y_i)$ such that $y_i = a_i x_i$, $x_i > 0$, and $x_i^2 + y_i^2 = 1$.

- For each b_i, produce a point in the 2D plane where the line $y = b_i x$ intersects the unit circle on the let (negative x) side.
- Diameter of this collection of $2n$ points is 2 if and only if the sets are NOT disjoint; otherwise the diameter is strictly less than 2.
- The entire transformation (input to diameter and back) takes $O(n)$ time.
- So, diameter in $2D$ requires $\Omega(n \log n)$ time.

4 3-SUM Hardness

- Unfortunately, for most geometric problems ADT at best yields an $\Omega(n \log n)$ lower bound.
- Using very different techniques and model, Chazelle, Fredman etc. have shown lower bounds for range searching, but that's an entirely different topic.
- Within computational geometry, there are many problems where we have not been able to beat the *quadratic* $O(n^2)$ algorithmic barrier, and it seems unlikely that it's even possible. But no lower bound technique is known.
- As partial progress, we have been able to show an *equivalence* class of many problems that are mutually $\Omega(n^2)$-Hard, meaning if you can devise a sub-quadratic algorithm for any one of them, we can solve all of them in sub-quadratic time bound.

4.1 3-SUM Problem

- Given a set S of n *integers*, is there a triple $a, b, c \in S$ such that $a + b + c = 0$?
- We can solve the problem in $O(n^2)$ time. (How?)
- The ACT model gives only an $\Omega(n \log n)$ lower bound.
- In spite of significant effort, no (truly) sub-quadratic time algorithm is known, under the standard model of computing. The straightforward upper bound of $O(n^2)$ can be slightly improved, using non-trivial techniques. For instance, a result by Gronlund and Pettie achieves the bound of $O(\frac{n^2}{(\log n/\log\log n)^{2/3}}$.
- The 3SUM conjecture is that it cannot be solved in time $O(n^{2-\varepsilon})$, for any $\varepsilon > 0$.
- Imitating the NP-completeness model of problem equivalence classes, the 3-sum problem can be used to show n^2-hardness of other problems.
- *A problem is* $3SUM$*-Hard if an* $o(n^2)$ *time algorithm for the problem implies an* $o(n^2)$ *time algorithm for* $3SUM$*.*
- A generalized version is kSUM, which is conjectured to require $\Omega(n^{\lceil k/2 \rceil})$ time.

4.2 Degeneracy Testing

- We have often conveniently assumed that the points are in *non-degenerate* position. How complex is to check that condition?
- Specifically, given a set S of n points in the plane, are three of them collinear?
- **Theorem.** $2D$ degeneracy testing is 3SUM-hard.
- **Proof.** Three numbers a, b, c sum to 0 if and only if (a, a^3), (b, b^3), (c, c^3) are collinear.
- Suppose the 3 points lie on a line $y = \mu x + \gamma$. Then, for the first two points, we can infer that:

$$a^3 - b^3 = \mu(a - b) \quad \text{which implies} \quad \mu = a^2 + ab + b^2$$

- Similarly, for the 2nd and 3d point, we get

$$b^3 - c^3 = \mu(b - c) \quad \text{which implies} \quad \mu = b^2 + bc + c^2$$

- Thus, $a^2 + ab + b^2 = b^2 + bc + c^2$, which gives $a + c = -b$, or $a + b + c = 0$.

4.3 Other $3SUM$-hard Problems

- Using similar ideas, one can show that all of the following problems are $3SUM$-hard.
- Given a set of n lines in the plane, are there three that pass through the same point?
- Given a set of (non-intersecting, axis-parallel) line segments, is there a line that separates them into two non-empty subsets?
- Given a set of (infinite) strips in the plane, do they fully cover a given rectangle?
- Given a set of triangles in the plane, compute their measure.
- Given a set of horizontal triangles in space, can a particular triangle be seen from a particular viewpoint?

5 Hardness using Exponential Time Hypothesis (ETH)

- The most common hardness assumption is $P \neq NP$, which is used to show that a problem is difficult to solve *optimaly* in *polynomial time*.
- Those proofs say nothing about the complexity of problems that *can* be solved in polynomial time but improving their time complexity seems to pose a barrier, such as $\Omega(n^2)$ or $\Omega(n^3)$.

- A new line of hardness proofs has emerged in the past decade or so, enabling us to argue that breaking those barriers may be just as hard as $P \neq NP$!
- In this lecture, we will consider one such line of arguments based on the *exponential time hypothesis* (ETH).
- Informally, ETH says that 3-SAT cannot be solved in $2^{o(n)}$ time, and SETH says that k-SAT needs roughly 2^n time for large k.

5.1 Hardness of Orthogonal Vectors

- The *orthogonal vectors* problem is to decide if there is an orthogonal pair of vectors between two sets. Specifically:
  1. Let $A = \{\alpha_1, \alpha_2, \ldots, \alpha_n\}$ and $B = \{\beta_1, \beta_2, \ldots, \beta_n\}$ be two sets of d-dimensional binary vectors.
  2. Decide if there is an *orthogonal pair* α', β', namely, two vectors that satisfy $\sum_{i=1}^{d} \alpha'[i] \cdot \beta'[i] \;=\; 0$.
- For simplicity, we assume that $d = \Theta(\log^2 n)$; in fact, $d = \omega(\log n)$ is enough. Otherwise, the sets A, B will either have repetitions or be the whole space of $\{0,1\}^d$ vectors.
- A brute force algorithm for OV takes $O(n^2 d)$ time—check the inner product of all pairs. We will show that we cannot hope to do better assuming SETH.
- **OV Theorem.** Assuming SETH, the OV problem requires $\Omega(n^{2-\varepsilon})$ time, for all $\varepsilon > 0$.
- **Proof.** We will reduce k-SAT to OV. Suppose the formula has n variables $x_1, \ldots, x_n$ and m clauses $C_1, \ldots, C_m$, where we can assume that $m = O(n)$.
  1. Split the variables into two sets of $n/2$ each, calling them X and Y.
  2. For each set X and Y, list all $2^{n/2}$ (partial) assignments.
  3. For each assignment α of X, define a m-dimensional vector $v(\alpha)$ whose jth coordinate is 0 if α sets (true) any of the literals in the jth clause; otherwise, the jth coordinate is 1. That is,

     $v(\alpha)_j \;=\; 0$ if and only if clause C_j is satisfied by partial assingment α.
  4. Similarly, for each partial assignment β of Y, define a m-dim vector $w(\beta)$.
  5. We now observe that any clause C_j is *unsatisfied* by the partial assignments α, β if and only if $v_j(\alpha) \cdot w_j(\beta) = 1$. Otherwise, either $v_j(\alpha)$ or $w_j(\beta)$ is 0, in which case C_j is satisfied by either α or β.
  6. Thus, C_j is satisfied if the jth bit of of the dot product $v(\alpha) \cdot w(\beta)$ is 0.
  7. Therefore, all the clauses of the k-SAT are satisfied if the entire dot product is zero, namely, $v \cdot w = 0$.

8. Our instance of OV has $N = 2^{n/2}$, and $d \approx m \approx n \approx \log N$.
9. Therefore, if the OV problem can be solved in time $O(N^{2-\varepsilon})$, then by this reduction the k-SAT can be solved in time $2^{(2-\varepsilon)n/2} = O(2^{(1-\varepsilon/2)n})$ which violates SETH. This completes the proof.

5.2 Hardness of Nearest Neighbors

- The *Bi-chromatic Closest Pair* (BCP) problem is the following: given two sets of points A and B in some space, find $a \in A$ and $b \in B$ such that $\|a - b\|$ is minimum.
- The BCP problem is a *batched* version of the Nearest Neighbor query problem: instead of finding NN of one query, we want to find NNs of multiple queries.
- We will argue that the trivial algorithm that solves the problem by pairwise comparisons is essentially best possible in worst-case.
- We will use the L_1 norm and points from the hypercube vertices $\{0,1\}^d$. (Observe that this is the version we used in LSH: Hamming distance with binary vectors.)
- **Theorem.** Assuming SETH, solving BCP needs $\Omega(n^{2-\varepsilon})$ time, for all $\varepsilon > 0$.
- **Proof.** Reduction from OV to BCP.

1. Suppose we have two sets of vectors $A, B \subset \{0,1\}^d$, with $d = O(\log^2 n)$, and $|A| = |B| = n$.
2. For the lower bound, we will like to relate the closest pair distance between A and B to their *dot product*.
3. The idea begins with the following relation, for any $a, b \in \{0,1\}^d$:

$$\|a - b\|_1 \;=\; \|a\|_1 \;+\; \|b\|_1 \;-\; 2\langle a, b\rangle$$

   (a) To prove this, we note that a and b are 0-1 vectors. Let D be the number of coordinates at which a and b differ, and I be the number of coordinates where they both have a 1.
   (b) Then, $D = \|a - b\|_1$, and $I = \langle a, b\rangle$.
   (c) In the count $\|a\|_1 + \|b\|_1$, the D disagreement coordinates are counted once, while the I agreement coordinates are counted twice.
   (d) So, $\|a\|_1 + \|b\|_1 = D + 2I$, which implies $D = \|a\|_1 + \|b\|_1 - 2I$, proving the equality.
4. Next, define $\bar{b} = \bar{1} - b$, where $\bar{1}$ is the vector of all 1s. Then, using the preceding equality, we get

$$\begin{aligned}\|a-\bar{b}\|_1 &= \|a\|_1 + \|\bar{b}\|_1 - 2\langle a,\bar{b}\rangle \\ &= \|a\|_1 + \|\bar{b}\|_1 - 2(\|a\|_1 - \langle a,b\rangle) \\ &= \|\bar{b}\|_1 - \|a\|_1 + 2\langle a,b\rangle\end{aligned}$$

5. Therefore, if partition A and B into the following subsets, for $i \in \{0,1,\ldots,d\}$,

$$A_i = \{a \in A : \|a\|_1 = i\}, \qquad B_i = \{b \in B : \|\bar{b}\|_1 = i\}$$

and define the set

$$\bar{B}_i = \{\bar{b} : b \in B_i\}$$

Then we get that for any $a \in A_i$ and $\bar{b} \in \bar{B}_j$, we have

$$\|a-\bar{b}\|_1 = j - i + 2\langle a,b\rangle \geq j - i$$

6. In other words, since an inner prduct is always non-negative, $j-i$ is a *lower bound* on the closest pair between A_i and $\bar{B}_j$, and that the *equality holds if and only if* $\langle a,b\rangle = 0$.
7. Therefore, to find $a \in A$ and $b \in B$ with $\langle a,b\rangle = 0$, we can run BCP on the pairs of sets A_i and $\bar{B}_j$ for all $i,j \in \{0,\ldots,d\}$, and check if the inequality in (4) is ever achieved among these pairs of vectors. (If it is, it must be for the norm-minimizing, namely, BCP pair of vectors.)
8. An $o(n^{2-\varepsilon})$ time algorithm for BCP then gives us an $o((d+1)^2 n^{2-\varepsilon}) = o(n^{2-\varepsilon}\log^4 n)$ time algorithm for the OV problem, which contradicts the hardness of OV.

- **NN Hardness.** Fix $\delta, c > 0$, and suppose an algorithm is allowed $O(n^c)$ time to preprocess the set A. It still needs $\Omega(n^{1-\delta})$ time to answer each online NN query for $b \in B$.

6 References

1. On a Class of $O(n^2)$ Problems in Computational Geometry. Anka Gajentaan, Mark H. Overmars: Comput. Geom. 165-185, 1995.
2. Hadrness of approximate nearest neighbor saerch. Aviad Rubinstein. STOC 2018.
3. A new algorithm for optimal 2-constraint satisfaction and its implications. Ryan Williams. Theoretical Computer Science, 2005.

7 Extra Material

7.1 Transformations and Reductions

- The most common technique for proving lower bounds is reduction. A reduction from problem A to B is the following procedure.

  1. Input of problem A is converted to a suitable input for B.
  2. Solve problem B.
  3. Transform the output into a correct solution to problem A.

- If the transformation steps 1 and 3 take time $\tau(N)$ on input size N, then we say that A is $\tau(N)$-reducible to B.

- [**Reduction Theorem.**] If problem A is known to require $T(N)$ time to solve, and A is $\tau(N)$-reducible to B, then B requires at least $T(N) - O(\tau(N))$ time.

- In other words, hardness of A proves hardness of B.

- Similarly, if B can be solved in $T(N)$, then A can be solved in $T(N) + O(\tau(N))$.

- In the precious reduction, we only transformed in the direction from A to B. If the $\tau(N)$-reduction works in both directions, then A and B are called *equivalent.*.

- But to get started, we first need a lower bound on A. How does one prove that some problem A must require $T(N)$ time no matter what algorithm is used?